AGX Dynamics Documentation



AGX Dynamics is a modular physics simulation toolkit written in C++. It contains the necessary core functionality to perform high fidelity, stable and robust simulations of mechanical systems in real-time, and can also be configured to deliver faithful simulation for applications in science and engineering. AGX Dynamics contains a mix of solvers for the resulting linear and LCP problems, suitable for different purposes: iterative for fast approximate solutions, and direct solvers for machine precision accuracy. In addition, AGX Dynamics consists of models for simulating: wires, cables, hydrodynamics, aerodynamics, tires, discrete element modelling (DEM), drivetrain, hydraulics etc.

AGX Dynamic's API also includes event handlers for injecting user code as listeners of contact events, step events etc.

Table of Contents

Front Page

1. Legal notice

2. Scope of document

3. Overview

4. Modules overview

5. Getting started

6. License systems

7. Dimensions and units

8. Tutorials

9. Creating stable simulations

10. Creating a Simulation

11. Bodies and Shapes

12. Constraints

13. Particle systems

14. AGX Granular

15. AGX Wire

16. AGX Cable

17. AGX Cable damage

18. AGX PowerLine

19. AGX DriveTrain

20. AGX Hydraulics

21. AGX Hydrodynamics

22. AGX Vehicle

23. AGX Terrain

24. Material calibration – bulk materials

25. OpenPLX for AGX Dynamics

27. AGX Util

28. AGX Model

29. AGX Model: Deformable Beam

30. AGX Model: Tire

31. AGX Model: PID Controller

32. AGX Sensor

33. Surface Velocity Conveyor Belt

34. AGX Control

35. Robotics

36. Reinforcement Learning

37. Strong interaction

38. Performance

39. Reducing model complexity

40. Energy calculations

41. Statistics

42. Serialization

43. File formats

44. 3D Viewer and default rendering

45. Journal Viewer

46. Locating files and resources

47. Environment variables

48. Python Scripting

49. C#/.NET bindings of the AGX API

50. Matlab/Simulink plugin

51. FMI Export

52. Debugging

53. References

1. Legal notice

This document is a document owned by Algoryx Simulation AB, and is only intended for internal use, or use of the individual or entity to which it is directed. It may contain information that is privileged and exempt from disclosure under applicable law. If you are not the intended recipient, please notify us immediately. You may not copy it or disclose its contents to any other person or entity unless specifically granted by Algoryx.

2. Scope of document

The purpose of this document is to describe the current state of the AGX Dynamics API as well give an overview of how to use the API.

In addition to this user manual, there is also the generated API documentation which is located in *<agx-dir>/doc/html*

On the windows platform, there are also shortcuts from the START menu to all the documentation and all the tutorials and demonstration scripts. It is located in START\AGX Dynamics-2.37.0.0 (replace 2.37.0.0 with the version of AGX Dynamics you have installed).

- AGX Dynamics Main Page – Link to a web page containing the tutorials and demonstration scripts.
- AGX Dynamics API Documentation – Doxygen generated API documentation.
- List of publications based on AGX Dynamics

2.1. Python documentation server

On Linux the API documentation included in the AGX package can be inaccessible if your web browser is running sandboxed. The sandbox can make parts of the file system inaccessible for the browser. To work around this problem, a python-based utility is provided with AGX that is available as */opt/Algoryx/AGX-<version>/bin/doc-server.py*. The utility starts a local webserver, listening on localhost, and provides a GUI for easy access to the documentation. For window managers that automatically find .desktop-files, there will also be an item on the start menu to launch the documentation utility.

3. Overview

3.1. Platforms

AGX Dynamics is built and tested under the following platforms and compilers:

- Windows 11 & 10; 64 bit. Compilers: Visual Studio 2019, 2022
- Linux, Ubuntu 18.04 (gcc 7.3.0), Ubuntu 20.04 (gcc 9.3.0), Ubuntu 22.04 (gcc 11.2.0)
- Mac OS X (clang 11.0.0)

3.2. Libraries/namespaces

AGX consists of more than 500 classes which are separated into a number of namespaces and libraries listed in Table 1. Each namespace is accessible through the include directive:

```
#include <namespace/headerfile.h>
```

Table 3.1 Available namespaces and libraries

NAMESPACE	LIBRARY NAME	DESCRIPTION
agxSabre	agxSabre.lib	Sparse matrix solver.
agx, agxData	agxCore.lib	Core dynamics library, including basic data types: Vec3, Quat, Matrices ref_ptr, etc.
agxCollide	agxPhysics.lib	Geometric intersection system for calculating intersection points, normals, penetration depth. Classes for various primitives such as Sphere, Cylinder, Box, …
agxIO	agxPhysics.lib	Contain classes for reading / (writing) various file formats (images, models).
agxSDK	agxPhysics.lib	Simulation framework. Higher level abstraction for modeling a system, stepping collision and dynamics forward in discrete time steps. Event handling, materials.
agxStream	agxPhysics.lib	Serialization classes.
agxUtil	agxPhysics.lib	Various utilities for reading CPU information etc.
agxNet	agxPhysics.lib	Classes for communication, Sockets, Compression.
agxOSG	agxOSG.lib	Various classes for integrating AGX to OpenSceneGraph. For testing/debug purposes.
agxModel	agxModel.lib	High level modeling primitives, Tree, Terrain etc.
agxWire	agxPhysics.lib	Implementation of lumped element wire.
agxTerrain	agxTerrain.lib	Implementation of deformable Terrain.

An application using physics simulation typically requires the following functionality:

Table 3.2 Functionality in different namespaces

NAMESPACE	FUNCTIONALITY	CLASS NAME
agxCollide	Modeling API for geometry/shapes	Geometry, Shape, Sphere, Box, …
agxCollide	Intersection testing and extraction of contact data	Space, SweepAndPruneBroadPhase
agx	Modeling API for physical bodies	RigidBody, MassProperties
agx	Modeling API for constraints	Hinge, BallJoint, LockJoint, …
agx	Solving non-linear systems for constraints/forces. Integration of velocities and positions.	ZorroSolver
agxSDK	Integration of user code for event handlers	StepEventLister, ContactEventListener, …
agxIO	Reading Wave front files, Collada, images…	MeshReader, ImageReaderPNG, …
agxSensor	Reading gamepads etc.	Joystick, JoystickManager..

When building/running an application with dynamics, it is important that collision space (agxCollide::Space) and the dynamics are updated and stepped correctly. Therefore there is a class that integrates these two into a higher level functionality: **agxSDK::Simulation**. This class makes sure that both

collision space and the dynamics system are updated when objects are added or removed. Even though it is possible to use *agxCollide::Space individually*, we recommend always using an agxSDK::Simulation instance when building simulations.

3.3. Debug vs. Release build mode

In the installer for Microsoft Windows, the debug libraries are available. Library/runtime files in debug build has the letter 'd' added to the library name:

BUILD MODE	SAMPLE NAMES	FLAGS
Release	agxCore.lib, agxPhysics.dll, agxModel.lib	C++: -D_NDEBUG Linker: /MD
Debug	agxCored.lib, agxPhysicsd.dll,agxModel.lib	C++: -D_DEBUG Linker: /MDd

When building applications with Microsoft Visual Studio it is very important to never mix libraries built in debug and release mode. The reason for this is how memory allocation/STL containers work in Visual Studio. Mixing libraries breaks the ODR (One Definition Rule) (see more on the web).

3.4. Solver/Dynamics

The solver in AGX Dynamics is a hybrid solver which is very flexible in terms of specifying solve type for various constraints. The solver contains both an *iterative* solver and a *direct* solver.

3.4.1. Iterative solver

An iterative solver gradually improves the solution with the number of iterations, and after a finite number of iterations it thus delivers a solution with finite precision. The most common type of iterative solver is the Gauss-Seidel relaxational method and an extension called Projected Gauss-Seidel (PGS) which is used to solve constrained linear systems of equations. It solves the equations one-by-one and updates all the dependent variables of the system before proceeding with the next equation. One iteration corresponds to a full pass through all the equations. Since the constraint equations are treated one-by-one to solve for the constraint force and corresponding updated velocity, and the information about this local solution propagates to the coupled equations, it is often referred to as an *iterative impulse propagation method*. However, the term propagation is somewhat misleading since the ordering of the equations is not well defined in PGS, so information propagates randomly, unless specifically ordered. Relaxational solvers are moderately efficient and conceptually simple, at least when the local constraint violations are small, which is often true for systems with small mass ratios and weak couplings. Therefore they can be fast and efficient for large scale stacking problems with homogeneous types of bodies, while they are less efficient for e.g. lines and wires with large mass and force ratios, where they result in rubber band behavior unless the number of iterations is set to an extremely high value. They are also not so good at treating loop closures in physical systems, unless these problems are given special treatment.

3.4.2. Direct solver

A direct solver finds the exact solution (down to machine precision) in a limited number of operations (provided there exist a well-defined solution). For sparsely connected systems, a sparse type of solver is much preferred over a general dense solver, and can give enormous performance benefits. The chances of finding a proper physical solution are improved through regularization, which introduces an extra inverse mass/inertia term in all equations.

Our direct solver computes solutions of quadratic programming (QP) problems using a block pivot method. This is similar to Newton-Raphson iterations without damping or line search and is generally very fast and almost always returns a good solution in just a few steps. The linear algebra operations rely on our own sparse symmetric indefinite factorization code which is tailored for multi-body dynamics, and can also handle ill-conditioned systems of equations arising from joint singularities, also known as gimbal lock. The overall algorithm is generally faster than publicly available QP solvers based on active set or interior point methods.

The direct solver is very robust and can handle large multi-body systems with very high mass ratios, and yet maintain constraints accurately, to second order with respect to the time step. This is slower than the iterative solver but the latter makes larger errors and cannot process systems that have large mass ratios, or many joints. The simulation of a heavy vehicle for instance requires the direct solver for stability.

3.4.3. Split solver

This is the default solver in AGX Dynamics. It is a hybrid solver utilizing both the DIRECT and the ITERATIVE solver. By default, all ordinary constraints (hinges, prismatics, etc.) and all contacts normal forces are in a first stage solved with the direct solver. The normal forces are used as input to the iterative solver which solves for friction in the second stage. In the third and last pass, the constraints and normal forces are solved again in the direct solver given the frictional forces in the system. This results in a very fast solution, even for moderately large contact scenes with friction. For even larger systems with stacking etc., it is more efficient to select a pure iterative solution (described in Section 11.15.5).

Selected constraints can also explicitly solver solved in the iterative solver, (see Section 12.15). The DIRECT solver utilizes an in-house implementation of a sparse matrix solver: SABRE. SABRE is an optimized solver for rigid body simulations. It makes heavy use of BLAS-level3 which results in a fully utilized CPU pipeline.

The split solver is slower than the purely iterative one, but it produces much better results with residual errors that are at least two orders of magnitude smaller. This generally produces more stable stacking with much better dry friction in general. Because of a fundamental mathematical limitation however, the direct solver becomes slower and slower for piling or volume filling configurations. The slowdown - the rate of change of the execution time divided by the number of objects depends on the topology of the problem. It is worst for volume packing, moderate for planar constructions, i.e. walls, and nil for linear ones, i.e., stacks.

The execution time for the purely iterative solver grows only linearly with the number of bodies if the number of iteration is kept constant.

3.5. Multi-core support

Modern processors usually have two or more execution units, cores, which can be used to parallelize systems.

AGX is built upon a task model, which allows for a whole tree of tasks, each depending on other tasks. This task-tree can be split and run in parallel based on each tasks predecessors. For example, update bounding volumes can operate in parallel on each geometry, which makes it parallelizable. On the other hand, the broad phase tasks, require the bounding volume to be updated, so it need to wait until update bounding volume is done. The directory Components in the binary directory of AGX consists of many task files (both binary and XML) which together build the complete structure of the AGX Dynamics engine. More on parallelization in Section 38.6.

3.6. Time integration

The time stepper used in AGX is derived from a discrete variational principle applied to the discrete Lagrangian representing the constrained dynamic system. This results in a stepping equation that also is valid for non-smooth and dry frictional forces, which are represented through Rayleigh dissipation functions. In the presence of smooth forces only this stepping equation corresponds to the classical Leapfrog integration scheme. For constraint impacts, a second order term is used to correct the velocities and the corresponding impulses. The rotational degrees of freedom require special treatment since the integration of the angular momentum and the quaternions contain a non-linear coupling in the gyroscopic term. In many cases the gyroscopic term is neglected (always in game physics engines, and usually the approximation is not even known, and thought to be “exact”). Neglecting gyroscopic forces corresponds to conservation of angular velocity rather than angular momentum, which in turn can be said to correspond to an approximation where the inertia tensor is homogeneous, which is true for symmetric bodies (i.e. a sphere with homogeneous mass distribution).

The integration of rigid body motion in AGX is based on a Cartesian representation of body coordinates in world frame x and their velocities $v = \dot{x}$ Their orientations are represented with quaternions q which represent the transformation from the body frame to world coordinates, as well as the angular velocity ω in world frame. If we label f as the sum of external and constraint forces, and τ as the sum of external and constraint torques, the time integration is

(3.1)

$$\begin{aligned} v_{k+1} &= v_k + \frac{h}{m} f_k \\ \omega_{k+1} &= \omega_k + h J_k^{-1} \tau_k \\ x_{k+1} &= x_k + h v_{k+1} \\ q_{k+1} &= q \exp(\frac{h}{2}\omega_{k+1}) \end{aligned}$$

where the quantities are defined according to
m the scalar mass
J the inertia tensor in world frame
h the timestep
k the discrete time
in addition, the exponential of the angular velocity is defined as the quaternion

(3.2)

$$\exp(\frac{h}{2}\omega) = \begin{bmatrix} \cos(\frac{h}{2}) \parallel \omega \parallel \\ \frac{1}{\parallel \omega \parallel} \sin(\frac{h}{2})\omega \end{bmatrix}$$

where quaternions are defined so that the first component is the real part, and the other three are the imaginary part. For a quaternion q, we denote the real and imaginary parts as q_0 and q_v respectively. We also use the right handed quaternion product so that for two quaternions p, q we have

(3.3)

$$pq = \begin{bmatrix} p_0 \\ p_v \end{bmatrix} \begin{bmatrix} q_0 \\ q_v \end{bmatrix} = \begin{bmatrix} p_0 q_0 - p_v \cdot q_v \\ p_0 q_v + q_0 p_v + p_v \times q_v \end{bmatrix}$$

and $(\cdot)$ and $(\times)$ are the ordinary scalar and vector products, respectively. The latter is also right handed so that for two vectors $x, y \in R^3$ then

(3.4)

$$x \times y = \begin{bmatrix} -x_3 y_2 + x_2 y_3 \\ x_3 y_1 - x_1 y_3 \\ -x_2 y_1 + x_1 y_2 \end{bmatrix}$$

The time integration described in Eqn. (3.1) corresponds to the velocity version of the Störmer-Verlet stepper [1]. The equations used to solve for the constraint forces are described elsewhere[see 2, Eqn.4.31 p. 100], and this is a symmetrized variant of the Shake algorithm [1] which includes robust constraint stabilization as well as constraint relaxation to prevent ill-conditioning in the linear algebra. Contact constraints are also described in [2], and the contact forces are computed by solving a linear complementarity problem.

3.7. Contact mechanics (Elastic Foundation Model)

Contact mechanics in AGX Dynamics is governed by a dynamic set of local contact constraints. These constraints exist where contacts are detected and are constantly updated as a simulation progresses. Mechanically they represent a linear-elastic material constitution which is specified individually for every

ContactMaterial. Furthermore, the constraints dynamically assess and utilize the local contact surface, penetration and material volume in order to collectively achieve highly stable and powerful contact mechanics.

Note however that this contact mechanics model is not intended nor generally suitable for systems that are heavily influenced by detailed or global structural deformations. Such demanding contact interactions may require complementing high-resolution nonlinear FEA.

The approach used in AGX Dynamics will lead to slight interpenetration between the bodies, which models local linear deformation. The amount of interpenetration will depend on material parameters such as Young's Modulus, as well as external forces influencing the contact. For parameters influencing the contact mechanics, (see Material).

There are two different approaches for calculating the resulting stiffness between two interacting geometries/shapes: contact based and area based. Which version that is being used, can be controlled per ContactMaterial. See: ContactMaterial.

3.7.1. Area Based approach

Collision detection will provide points, which are a discretization of the contact patch. For each point, a separating normal, as well as a contact depth (in case of overlap) and a contact area (a part of the complete contact patch) will be computed. Each contact point can be considered as a spring. Assuming the ContactMaterial's Young's modulus E and a material rest length h, the spring's contact stiffness per unit area k can be defied as:

$$k = E / h$$

Note that the rest length h measures the total length of the bodies in contact direction, and it varies with the position of the contact point and the direction of the contact normal. The total length of the involved shapes belonging to each of the two bodies is measured along the point and normal, with an upper cut-off representing the maximum length of the locally elastic domain, as well as a lower cut-off for numerical reasons.

Given a depth d computed from collision detection, the local pressure P can then be computed as:

$$P = k \cdot d$$

Given the contact area A for the contact point, the force F for the contact point is computed as:

$$F = P \cdot A$$

This contact model is a simplified version of the Elastic Foundation Model, which is defined in the publications by Johnson [3] and Perez [4]. Note that the version presented here differs from the ones presented in the literature in the following ways:

- The effect of Poisson's ratio is ignored, thus corresponding to a Poisson's ratio of 0 (a Poisson's ratio of 0.3 would increase k with a factor of 1.35).
- The Young's Modulus is defined for each pair of materials, and the rest lengths of the two bodies are added, thus giving a common spring stiffness per unit area. This is a simplification of the original model, in which the stiffness per unit area is computed for each body given its Young's modulus, rest length (andPoisson's ratio), and a common spring stiffness per unit area is computed by $k = \frac{k_1 \cdot k_2}{k_1 + k_2}$

3.7.2. Contact point based approach (default)

The default contact model is an approximation of the area-based model above, where h is set to 1m and the contact area is approximated.

This model has the advantage that it is computationally faster and gives good-enough stability and fidelity in scenes with objects of sizes 1dm and larger as well as heavy masses, but it is less exact, especially when dealing with smaller and lighter objects.

3.8. Unicode

AGX uses utf-8 internally. One of the reasons is that it is backwards compatible with ascii. It will also save storage. std::string/agx::String works just as it is as a storage container for utf8 strings.

In <agx/Encoding.h> there are several utility methods for converting between Wide and utf8.

So for example, reading a string encoded in utf8 from the windows registry requires you to convert from utf8 to Wide:

```
const char *keyInUtf8; // Initialized with some utf8 string
std::wstring wKey = agx::utf8ToWide( key ); // Convert to Wide character string

agx::String& value; // Here is where we store the result

dwType=0;
DWORD dwLen=0;
status = RegQueryValueExW(hkey, wKey.c_str(), 0,&dwType,nullptr, &dwLen);

if (status== ERROR_SUCCESS &&  dwType == REG_SZ)
{
agx::Vector<wchar_t> data;

// Since we use wchar, this will overallocate, but that's ok.
data.resize(dwLen+1);

// Now get the value
```

```
status = RegQueryValueExW(hkey, wKey.c_str(), nullptr,nullptr,(PBYTE)(data.ptr()), &dwLen);
if (status== ERROR_SUCCESS)
{
  // If we use RegQueryValueEx and UNICODE is not set, then the API call
  // will be RegQueryValueExA that will convert from wstring to ANSI if
  // the data in the registry key is written as unicode.
  //
  // We don't want that conversion since it might not be safe and having to mess
  // around with code pages is not fun.

  value = agx::wideToUtf8( std::wstring(data.ptr() ) );
  found = true;
}
RegCloseKey(hkey);
}
```

3.8.1. Limitations

Applications such as agxViewer will not be able to read files in a path containing utf8 characters.

C++ streams do not handle unicode paths in windows, and there are no open methods that handle wstring. Hence basic_ios and classes such as ifstream must not be used for accessing files in paths with non-ascii characters.

4. Modules overview

The named modules below extend the core functionality of AGX Dynamics to cover a wide range of simulation needs.

- AGX Cable
- AGX Cable Damage
- AGX Control
- AGX ROS2
- AGX DriveTrain
- AGX Hydraulics
- AGX Granular
- AGX Hydrodynamics & Aerodynamics
- AGX Sensor
- AGX Terrain
- AGX Tire
- AGX Tracks
- AGX Wire

4.1. AGX Cable

License	AgX-Cable
Library	agxCable.lib
Namespace	agxCable

The AGX Cable module can be used to simulate hoses, ropes, dress packs, and of course, cables. These are long structures with a circular cross section that can be bent, stretched and twisted. Modeled with discrete elements that can be subjected to large displacement. AGX Cables can interact with all other AGX geometry shapes, including itself and other cables. Robot dress packs analysis is a particularly relevant application.



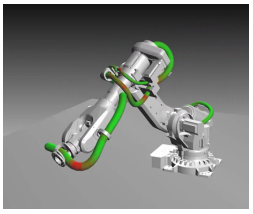
Features:

- High-fidelity self interaction
- Arbitrary stiffness and damping for torsion, bend, and stretch
- Support for plastic deformation
- Internal force reporting
- Fully integrated with other modules including hydrodynamics
- Easy routing along list of points in space
- Initial state can be configured with any shape

4.2. AGX Cable Damage

License	AgX-CableDamage
Library	agxCable.lib
Namespace	agxCable

Building upon AGX Cable, the AGX Cable Damage module provides computations to estimate wear and damage caused on a cable during the simulation and provide an aggregate report. All types of deformations are considered such as stretch, bend, and twist, along with deformation rates and what forces caused them. These are aggregated according to a configurable set of weights. This greatly assists the optimization of controllers and dresspack routing during development, or analyze problems and identify their causes after deployment.



Features:

- Helps tune controllers to minimize damage
- Accounts for stretch, bend, twist, and their rates
- Accounts for internal and external forces
- Accounts for contacts and friction
- Configurable damage metric
- Live and postmortem analysis
- Perfect for robot programming and design

4.3. AGX Control

License	AgX-Control
Library	agxModel.lib
Namespace	agxControl, agxModel

The AGX Control module provides functionality related to controlling mechanical systems such as vehicles and machines. Specifically, it contains forward kinematics, inverse kinematics and inverse dynamics. In addition it also contains functionality for Pure Pursuit path tracking for vehicles.

Related to this is also the library for PID controllers.

agxModel::ReconfigureRequest is another related functionality that allows for reconfiguring a complex mechanical system via a dynamic simulation. A wheel loader for example, you might want to transform the bucket to a specific transform, the rest of the machine should then adopt to this new configuration in a physically correct way, including ranges etc.

Features:

- Inverse kinematics for robotic systems
- Inverse dynamics for robotic systems
- PID Controllers
- Pure Pursuit path tracking for vehicles
- Dynamic reconfiguration of mechanical systems

4.4. AGX ROS2

License	None
Library	agxROS2.lib

Namespace	agxROS2

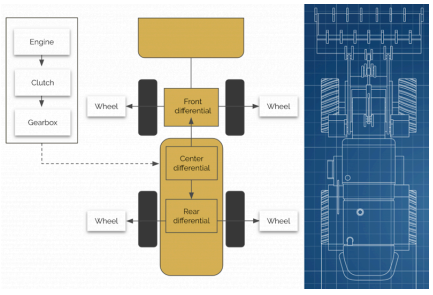
The ROS2 module for AGX Dynamics provides seamless integration between AGX Dynamics simulations and the Robot Operating System 2 (ROS2) framework. This module enables users to leverage ROS2's powerful communication and middleware capabilities to control and monitor AGX Dynamics simulations in real-time.

- Subscribers and Publishers for standard messages.
- Support for Sensor (Lidar) data publishing.
- Self contained: Does not depend on any ROS2 installation.

4.5. AGX DriveTrain

License	AgX-DriveTrain
Library	agxModel.lib
Namespace	agxDriveTrain, agxPowerLine

The AGX DriveTrain module simulates the 1D dynamics of mechanical components such as shafts, gears, differentials, and clutches in tight coupling with full 3D multi-body dynamics. AGX DriveTrain offers engine models via torque versus RPM lookup table or with a more accurate, first-principle diesel engine model. The module can be used with any AGX Dynamics model and in conjunction with any other modules to power excavators, trucks, winches, and robots, even to power a hydraulic pump.



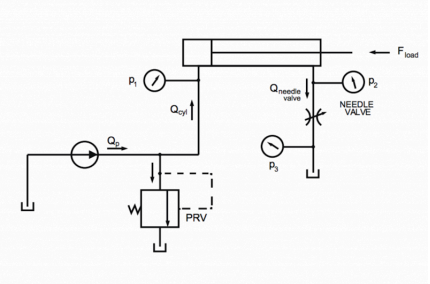
Features:

- Components for building a complete drivetrain: shaft, clutch, gear box, brake, differential and more
- Combustion engine model
- Torque/RPM engine model
- DC electric motor
- Deep integration with multi-body dynamics: Connect drivetrain directly to rotational or linear constraints
- Wire winch support
- Integrated with AGX Hydraulics

4.6. AGX Hydraulics

License	AgX-Hydraulics
Library	agxModel.lib
Namespace	agxHydraulics

AGX Hydraulics is a 1D simulation of hydraulic systems tightly coupled to the multi-body simulation in 3D. Components for building hydraulic systems integrated with mechanics include: accumulator, binary valves, check valves, motors, pumps (both fixed and variable displacement), stop valves, needle valve, and relief valve. It integrates with the AGX Drivetrain module, allowing for modeling of mixed mechanical and hydraulic systems. Coupling to mechanical simulation is done through linear and rotational actuators, enabling simulation of complete hydraulic-powered systems such as hydraulic winches, excavators, cranes, and wheel loaders.



Features:

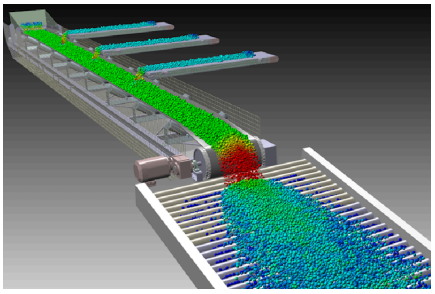
- Integrated simulation/coupling of 1D hydraulics and 3D multi-body dynamics using the same solver: allows for large time steps and robust simulation
- Handles stiff interaction between hydraulics and mechanical simulation
- Fast and efficient
- Library of basic hydraulic components: pump, motor, valves, accumulator, actuator

4.7. AGX Granular

License	AgX-Particles
Library	agxPhysics.lib
Namespace	agx, agx::Physics

With the AGX Granular module, you can study and optimize the design and control of systems that transport and process granules in bulk—rocks, pellets, pills, or grains. Material flow speed and evenness are fundamental to efficiency, leading to significant savings in time, energy, and improved quality. AGX Granular uses the Non-smooth Discrete Element Method (NDEM) instead of conventional Discrete Element Method (DEM), meaning it is deeply integrated with core models and numerical methods at the heart of AGX Dynamics. The core multi-body and specialized granular solvers are strongly coupled, leading to accurate coupling forces between granules and mechanical systems and very stable simulations. Simulating a pelleting drum with hundreds of thousands of pellets or a fully loaded dumper truck can be done at large time steps with full confidence.

The AGX Granular module is also available in a separate product in a full CAD environment. For more information see Algoryx Momentum.



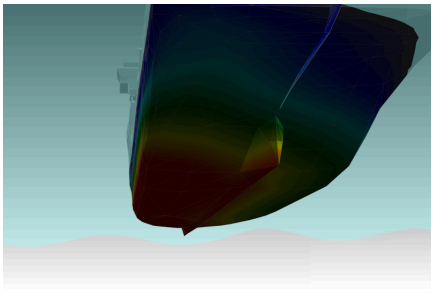
Features:

- Large time-step integration
- Stable, accurate integration with multibody dynamics
- High-performance parallel solver
- Linear and Hertzian contact models
- Rolling resistance for stable piles
- Statistics output and sensors
- Support for recording and playback of simulations (Journal files).
- Journal Viewer with UI for analysis of recorded simulations
- Force visualization
- Python scripts for analysis and data export

4.8. AGX Hydrodynamics & Aerodynamics

License	AgX-Hydrodynamics
Library	agxModel.lib
Namespace	agxModel

The AGX Hydrodynamics module computes drag and lift forces on any geometry due to water or air directly from the 3D tessellation and accounting for water level. As every triangle is accounted for, the forces are as accurate as the geometric representation. This is essential for simulating ROVs, submarines, anchors and oil rigs, as well as cranes in heavy wind conditions and wind turbines. It is fully integrated with AGX Wire and AGX Cable modules.



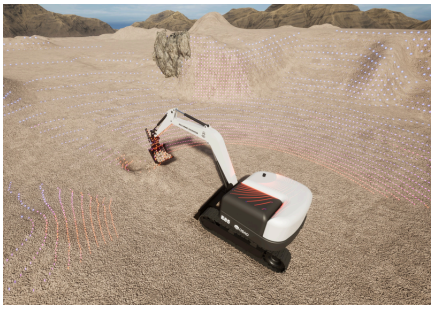
Features:

- Computes lift, drag, and buoyancy forces
- Includes hydrodynamic added mass
- Works directly on the geometry
- Accounts for dynamic waterline at each simulation step
- User-defined wind or water currents

4.9. AGX Sensor

License	AgX-Sensor
Library	agxSensor.lib
Namespace	agxSensor

AGX Sensor is a module of sensors and utilities for simulating physically accurate real-time sensor readings coupled to the multi-body simulation. It contains a set of convenient-to-use sensor types, which can easily be configured directly from real-world sensor data sheets. These sensors reside in a sensor environment that contains all objects detectable by the sensors and all environmental parameters, such as fog, rain, dust, surface materials, and magnetic fields, in a way that does not interfere with the main multi-body simulation. This module integrates with the AGX Terrain module, allowing sensors to detect terrain changes in real-time.



Sensor Types:

- LiDAR
- Inertial Measurement Unit (IMU)
- Accelerometer
- Gyroscope
- Magnetometer

4.10. AGX Terrain

License	AgX-Terrain
Library	agxTerrain.lib
Namespace	agxTerrain

The AGX Terrain module simulates soil deformation via interactions with wheels, tracks, blades, buckets, and shovels. Useful for bulldozers, excavators, wheel loaders, and track vehicles, it supports excavation, grading, and all forms of contacts, including all sides of any tool. It rigorously conserves mass and accounts for compaction and soil failure. It includes a calibrated, configurable library of soil models for gravel, sand, and dirt. Most importantly, this module computes accurate bi-directional forces between contacting bodies, tools, and terrain—crucially relevant for machine learning applications when developing autonomous machinery.

With Terrain Paging, only the proximity of the terrain near a vehicle is active at one time, allowing for unlimited working areas.



Features:

- Real-time performance
- Accurate soil mechanics with compaction
- Accurate calculation of interaction forces
- Strong coupling with all of AGX Dynamics
- Support for dozer blades, buckets, shovels, tires, and tracks
- Excavation, grading, and compaction capabilities
- Extensible and configurable terrain material library
- Supports terrain paging for unlimited terrain size

4.11. AGX Tire

License	AgX-Tires
Library	agxModel.lib, agxVehicle.lib
Namespace	agxModel, agxVehicle

The Two body Tire module adds flexibility to otherwise rigid wheels to represent tires. The model introduces elasticity, slip, and anisotropic friction in forward and transversal directions. AGX Tire works for on and off-road scenarios. Related to this is also the agxVehicle::WheelJoint and the various steering models that can be used to simulate realistic suspension and steering behavior.



Features:

- Independent forward and sideways friction

- Elasticity and damping in radial, torsional, and lateral directions
- Fast and efficient

4.12. AGX Tracks

License	AgX-Tracks
Library	agxVehicle.lib
Namespace	agxVehicle

The AGX Tracks module allows for simulation of crawler/tracked vehicles or dynamic conveyor belts. The model contains a routing algorithm that automatically sets up the geometrical configuration of the tracks, including initial tension. An automatic model reduction system ensures performance is retained even for a large number of treads or shoes, and it allows for varying geometry setup over the length of the track. Suitable for bulldozers, wheel loaders, and any tracked vehicle.



Features:

- Automatic routing for easy setup
- Works with AGX Terrain
- Variable pad configuration
- Two track models, one for slow moving vehicles with high fidelity and one for fast moving vehicles with high performance.
- Can also be used to simulate conveyor belts

4.13. AGX Wire

License	AgX-Wire
Library	agxPhysics.lib
Namespace	agxWire

Like cables, wires have stretch and bend resistance but differ in having no resistance to torsion. They also have dynamic resolution, allowing them to be kilometers long and subjected to arbitrarily high tension. The resolution is adapted according to contact geometry so they can be wrapped around other objects, including other wires. They are used for cranes, anchor handling, winching, tethers, and umbilicals for ROVs and other sub-sea operations. Wires can be tuned to represent ropes, wires, and chains. AGX Wire has proven itself in mission-critical simulators around the globe for almost two decades.



Features:

- Detailed material specifications
- Capable of handling extreme mass ratios

- Tension and bend force reporting
- Responds to wind and hydrodynamics
- Winches with speed and tension control
- DriveTrain and Hydraulics can power winches
- Wires make contact with all other objects
- Dynamic configuration: cut, merge, split, and attach objects during runtime

5. Getting started

5.1. Installation

This chapter describes the process of installing a binary version of AGX Dynamics.

The name of the installation executable informs you of the build environment used when building the libraries:

agx-setup-<*version*>-<*platform*>-<*compiler*>-<*precision*>.exe

For example:

agx-setup-2.30.0.0-x64-VS2019-double.exe

AGX version 2.30.0.0
Platform: 64bit
Compiler: Visual Studio 2019

5.1.1. Install AGX in Windows

Start the installation executable.

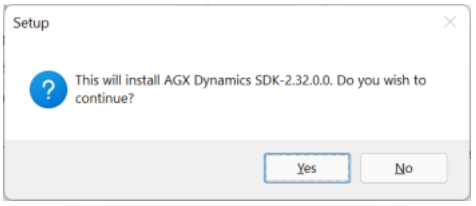


Fig. 5.1 After the splash screen you will get a verification question (if you have UAC enabled) which will show you a verified installer from Algoryx Simulation.

If you have a prior installation of AGX you will get a query regarding uninstalling that version. It is recommended that you do so.

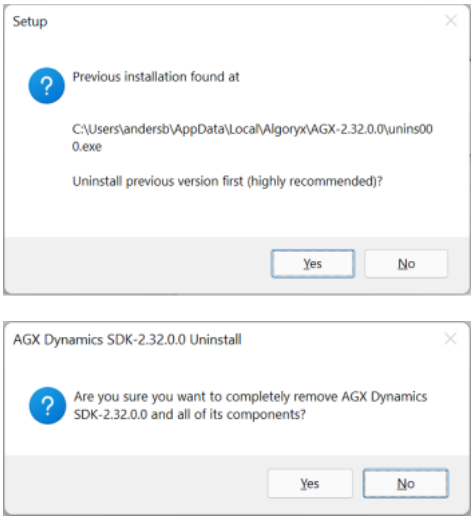


Fig. 5.2 Next is the uninstall window, just press Yes to remove the previous installation.

If you have created additional files, such as CMake, visual studio project files etc, they will not be removed. You will have to manually remove these files.

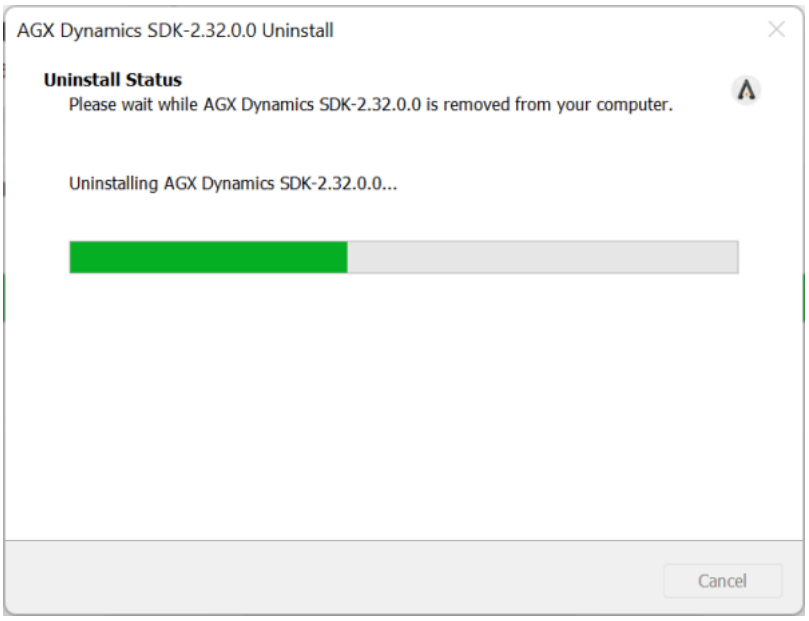


Fig. 5.3 Uninstall window.

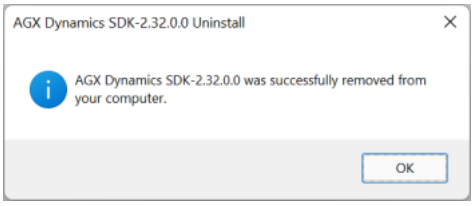


Fig. 5.4 Message indicating uninstall is performed. When the uninstall is finished the installer will continue with the installation of AGX.

To continue the installation of AGX Dynamics, you need to read and accept the License Agreement.

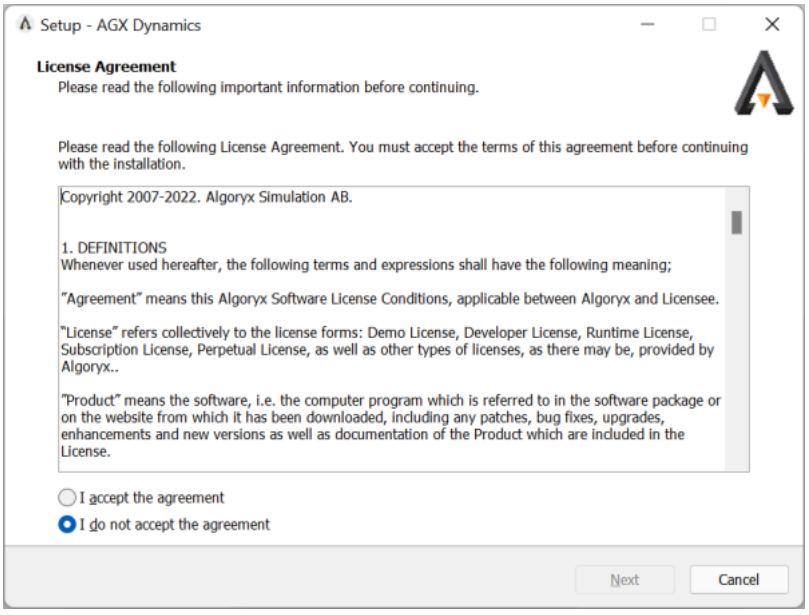


Fig. 5.5 License agreement.

Next is the installation options. Here various options for the installation are listed. If you are not going to utilize the Python API, you can uncheck the “Install python libraries” and leave the download of ROS2 unchecked. If you need to authorize a license, then check the box for “Run the license authorization program?”.

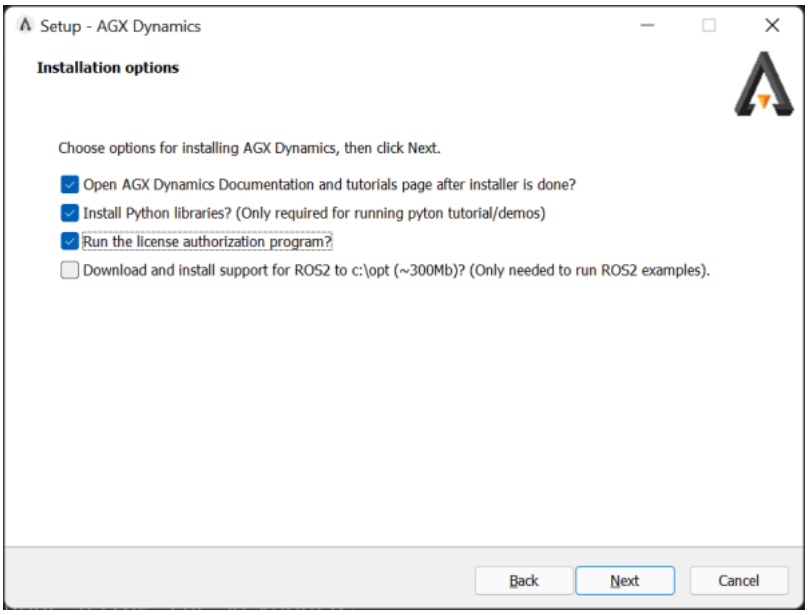


Fig. 5.6 Installation options.

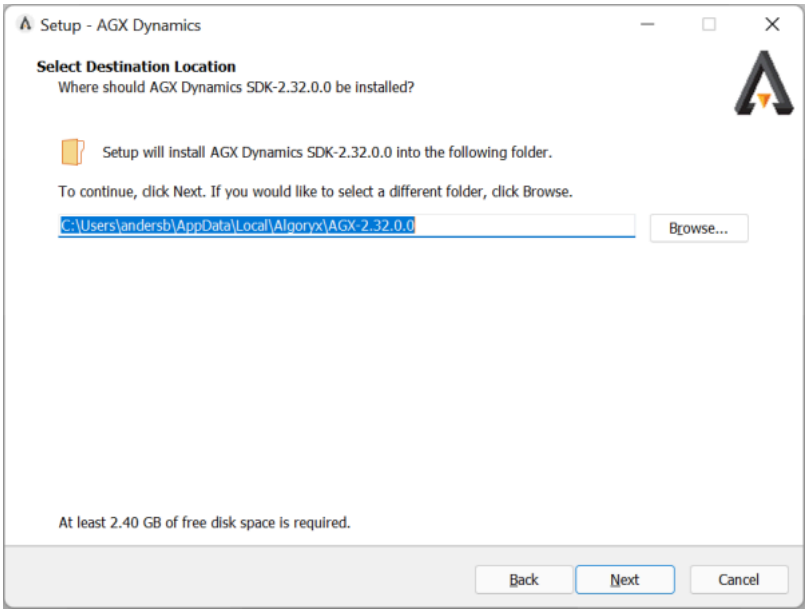


Fig. 5.7 Install destination

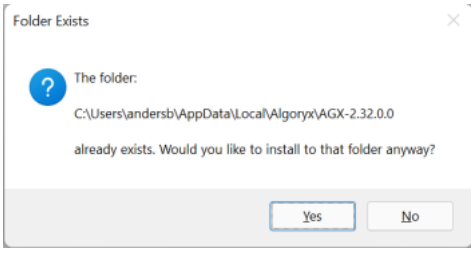


Fig. 5.8 If the target directory exists, you might get a final overwrite verification.

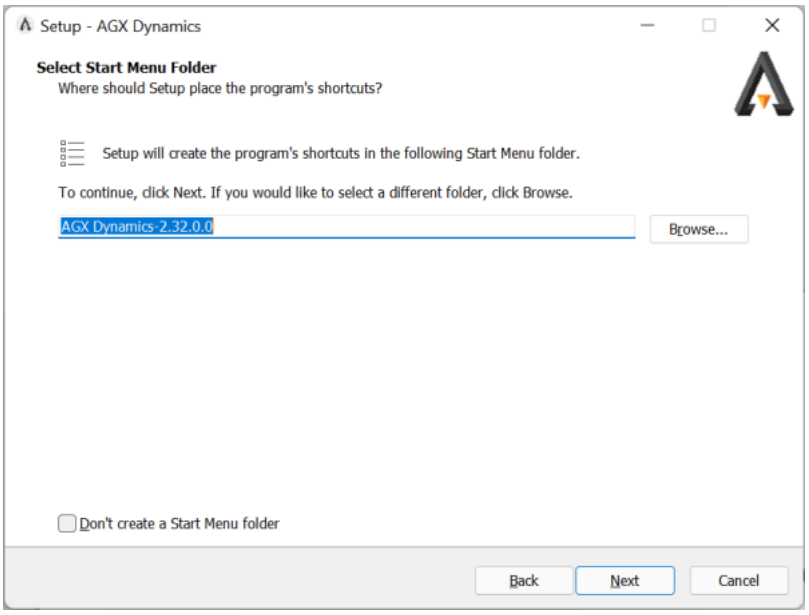


Fig. 5.9 Name of entry in Start menu

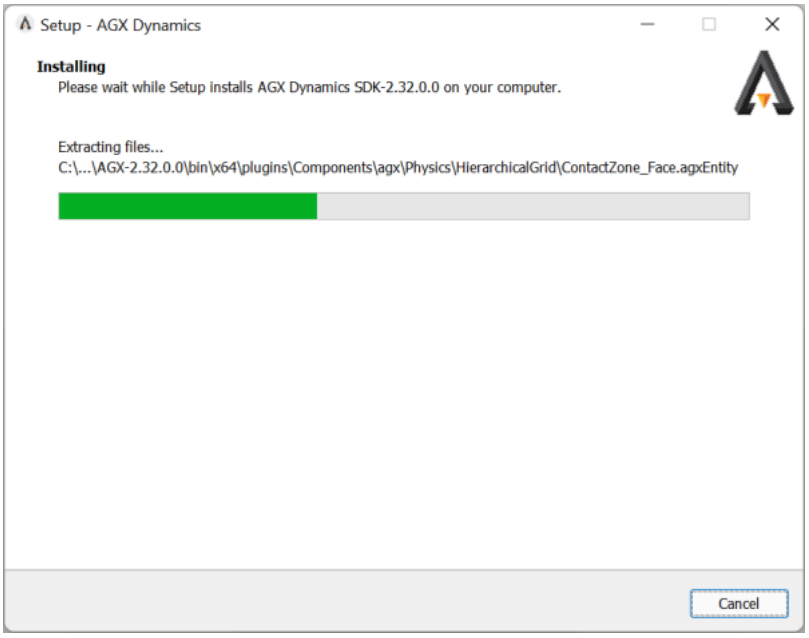


Fig. 5.10 Install in progress.

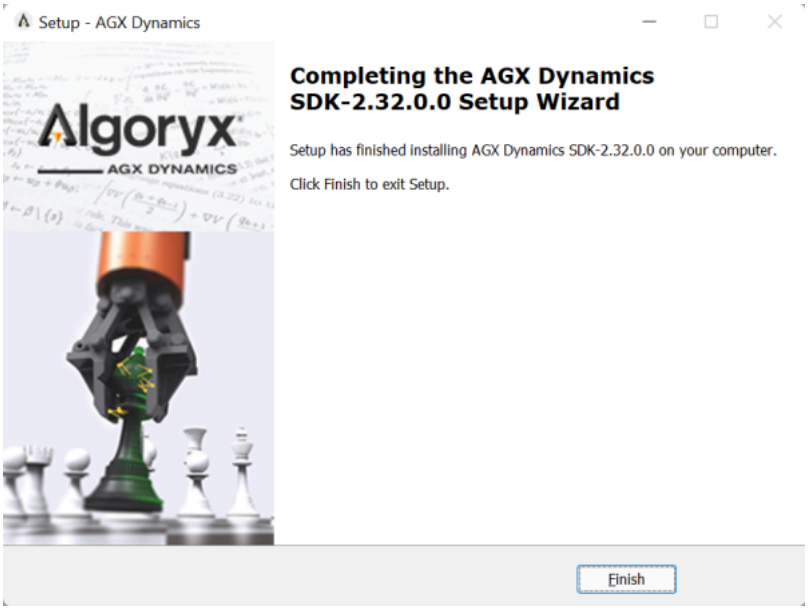


Fig. 5.11 AGX Dynamics successfully installed.

5.1.1.1. Post install

If the option for downloading ROS2 was checked, a progress bar for downloading the complete ROS2 package is shown. After the download is complete, a command window where the files are unzipped will be started.

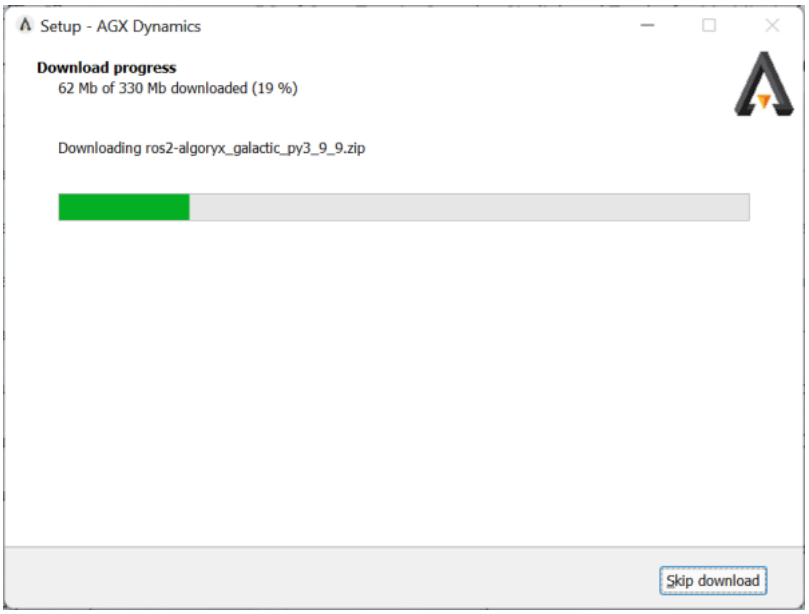


Fig. 5.12 Install in progress.

If the option for the license authorization was checked, a window that shows the license manager will be displayed.

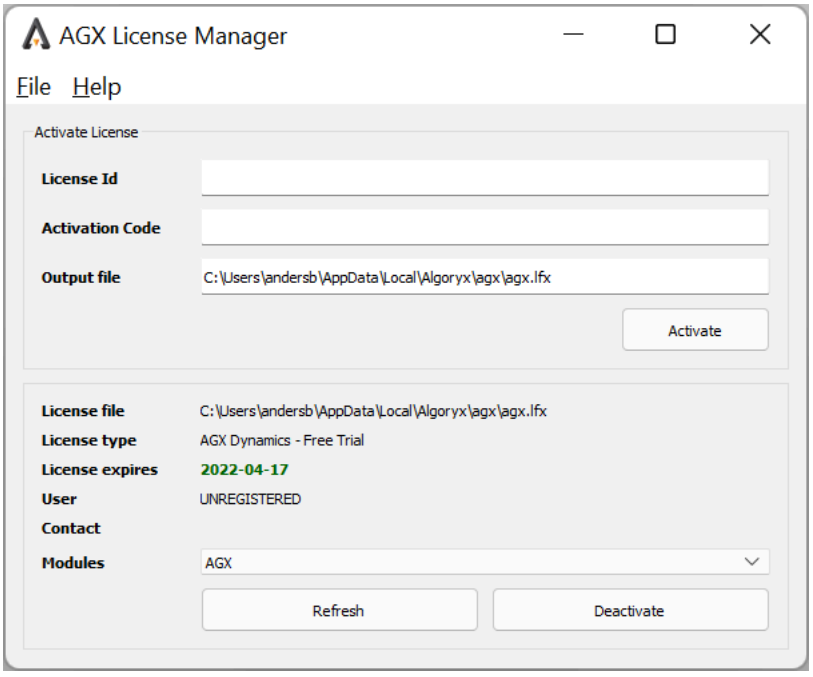


Fig. 5.13 License authorization tool.

If the option for installing python libraries was checked, a command window with the installation will be shown.

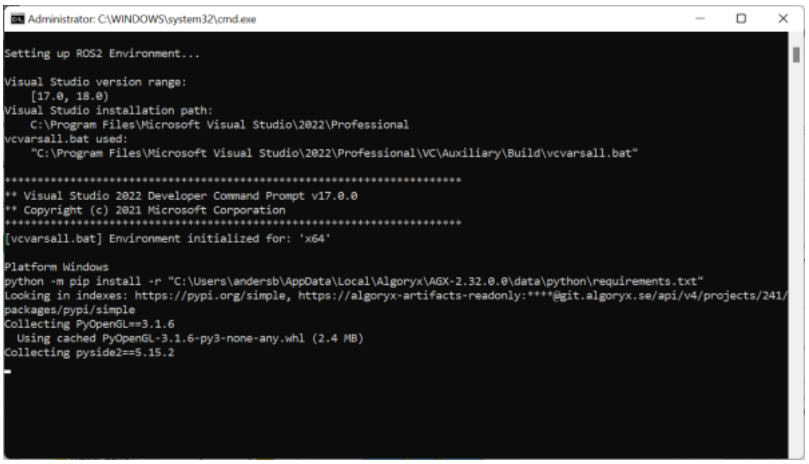
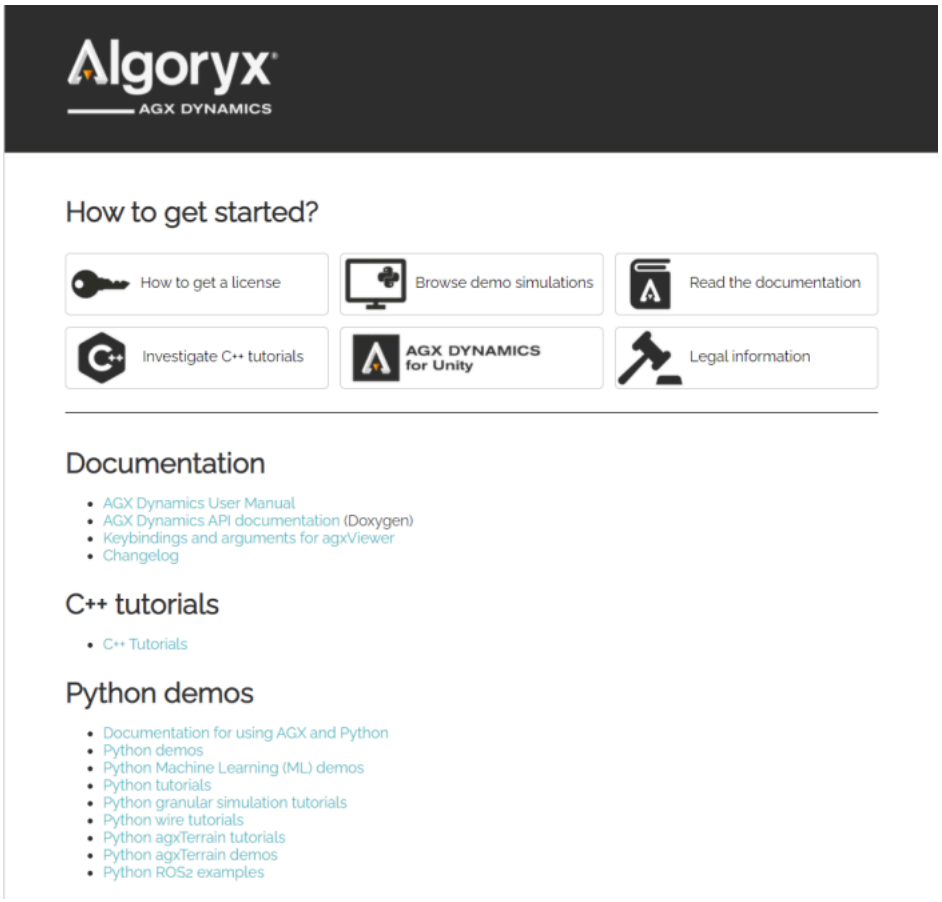


Fig. 5.14 Installing python libraries.

After the installation is done and the option for opening documentation page was checked, the AGX Dynamics startup page will appear with documentation, tutorials and examples.



5.1.2. Install on Ubuntu

Different Linux distributions and versions might have different installation procedures. The procedure described here matches the one used for installing a deb-file on Ubuntu 20.04.

By default, AGX Dynamics is installed into /opt/Algoryx/AGX-<version>

```
$ sudo apt-get update  # Refresh package lists if needed
  ...
$ sudo apt install ./agx-${AGX_VERSION}-amd64-ubuntu_20.04.deb
      Reading package lists... Done
      Building dependency tree
      Reading state information... Done
      Note, selecting 'agx' instead of 'agx-${AGX_VERSION}-amd64-ubuntu_20.04.deb'
      The following additional packages will be installed:
  ...
```

```
      Suggested packages:
  ...
      The following NEW packages will be installed:
  ...
      0 upgraded, 56 newly installed, 0 to remove and 0 not upgraded.
      Need to get 36.8 MB/517 MB of archives.
      After this operation, 2847 MB of additional disk space will be used.
      Do you want to continue? [Y/n]

$ ls /opt/Algoryx/AGX-${AGX_VERSION}/
      AGX_build_settings.txt  LICENSE.TXT  applications     conanfile.txt  include         swig
      CMakeLists.txt          README.TXT   bin              data           lib             tutorials
      CMakeModules            agxOSG       build-info.json  doc            setup_env.bash
```

You need to manually place your AGX license in the installation directory. Also, before running any AGX simulation the shell environment must be updated with some environment variables. Do this by starting a bash shell and execute the following command:

```
$ source /opt/Algoryx/AGX-<version>/setup_env.bash
```

AGX binaries and libraries use embedded rpath information to help locate needed shared libraries. The environment script previously set LD_LIBRARY_PATH but has been updated to no longer do so. If your usecase has trouble finding the needed libraries, the following can be used after sourcing the environment script:

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/Algoryx/AGX-<version>/lib
```

You can then start the tutorial applications or load Python scripts using agxViewer:

```
$ agxViewer data/python/cables.agxPy
```

The Linux package also install desktop files which most window managers will find automatically. Via these desktop files, the window manager will show menu options in the development section which help locate AGX API documentation, the user manual and the tutorials/demos.

5.1.3. Install on macOS

To install AGX on macOS, right click the downloaded installer file, select Open and follow the instructions. The installed AGX environment will be available in */opt/Algoryx/AGX-<version>*.

To use AGX with python, you must have Python3.9 installed on your machine, using the homebrew package manager

```
$ brew install python@3.12
```

You need to manually place your AGX license in the installation directory. Also, before running any AGX simulation the shell environment must be updated with some environment variables.

Do this by starting a bash shell and execute the following command:

```
$ source /opt/Algoryx/AGX-<version>/setup_env.bash
```

You can then start the tutorial applications or load Python scripts using agxViewer:

```
$ agxViewer data/python/cables.agxPy
```

5.1.4. Installing license

To learn how to use the license system, please read chapter Section 6.1.

5.2. Building your first application

AGX utilizes CMake for setting up a portable build environment. In the directory tutorials, there is a sample CMakeLists.txt which can be used as a template for building applications based on AGX. This chapter explains how to build the tutorials from a binary release of AGX on various platforms. It describes how to setup the include and library paths and which libraries are necessary to build an application based on AGX. First, make sure you have the correct libraries for your target platform (32/64 bit).

Note

AGX Dynamics is built with RTTI and exception enabled. Integrating AGX Dynamics into an application without having features enabled might cause runtime problems such as crashes or undefined behaviour.

5.2.1. Setting up a runtime environment

Note

To be able to execute an application using AGX Dynamics you need make sure you have the correct setup environment variables set. The AGX Dynamics runtime directories are not added to the PATH by default.

The AGX Dynamics installation comes with script/bat files for setting up the runtime environment. In Windows the file is named **setup_env.bat** and in Linux/Mac: **setup_env.bash**. These setup files will initialize environment variables needed for AGX Dynamics to run properly, including PATH, AGX_FILE_PATH etc.

5.2.1.1. Windows

For the Windows platform there are a few ways of getting a correct environment:

In the START menu, there is a **AGX Dynamics Command line** shortcut. This will open up a cmd prompt with the correct setup. By default it will utilize the internal Python interpretator. See Section 48 for more information on how to use a pre-installed Python together with AGX Dynamics.

If you want to start an application such as Matlab, Unreal Editor or Unity with a correct environment for AGX Dynamics, you can use the **agx_cmd.bat** which is located in the AGX install directory. The example below will properly initialize AGX Dynamics environment and start the specified application.

```
"c:\Users\username\AppData\Local\Algoryx\AGX-2.33.0.0\agx_cmd" "c:\Program
Files\Unity\Hub\Editor\2019.3.5f1\Editor\Unity.exe"
```

5.2.1.2. Linux/macOS

To setup a working environment for the Linux or macOS platform you need to execute the script **setup_env.bash** that comes with the AGX Dynamics installation. You will find this in the installation directory.

```
$ source /opt/Algoryx/AGX-2.33.0.0/setup_env.bash
$ agxViewer data/python/cables.agxPy
```

5.2.2. Windows/Visual Studio

AGX for the Windows platform is built using Visual Studio with runtime libraries from *MultiThreaded DLL (MultiThreaded Debug DLL* in the debug version). When you build your application, it is also important to link to the correct version of the libraries (debug/release), as it is not recommended to mix the two in the same application. AGX for windows is delivered with both debug and release libraries (see below).

Attention

By default, the installer will install AGX Dynamics into a user directory, *c:\Users\username\AppData\Local\Algoryx\AGX-2.33.0.0*. This results in that you can change/add/remove files from the installation directory as you like. If you choose to install AGX Dynamics into another folder, for example *c:\program files\Algoryx* you might need to have administrative rights to change files in the folder. The below instructions assume you have full access right to the AGX Dynamics installation path.

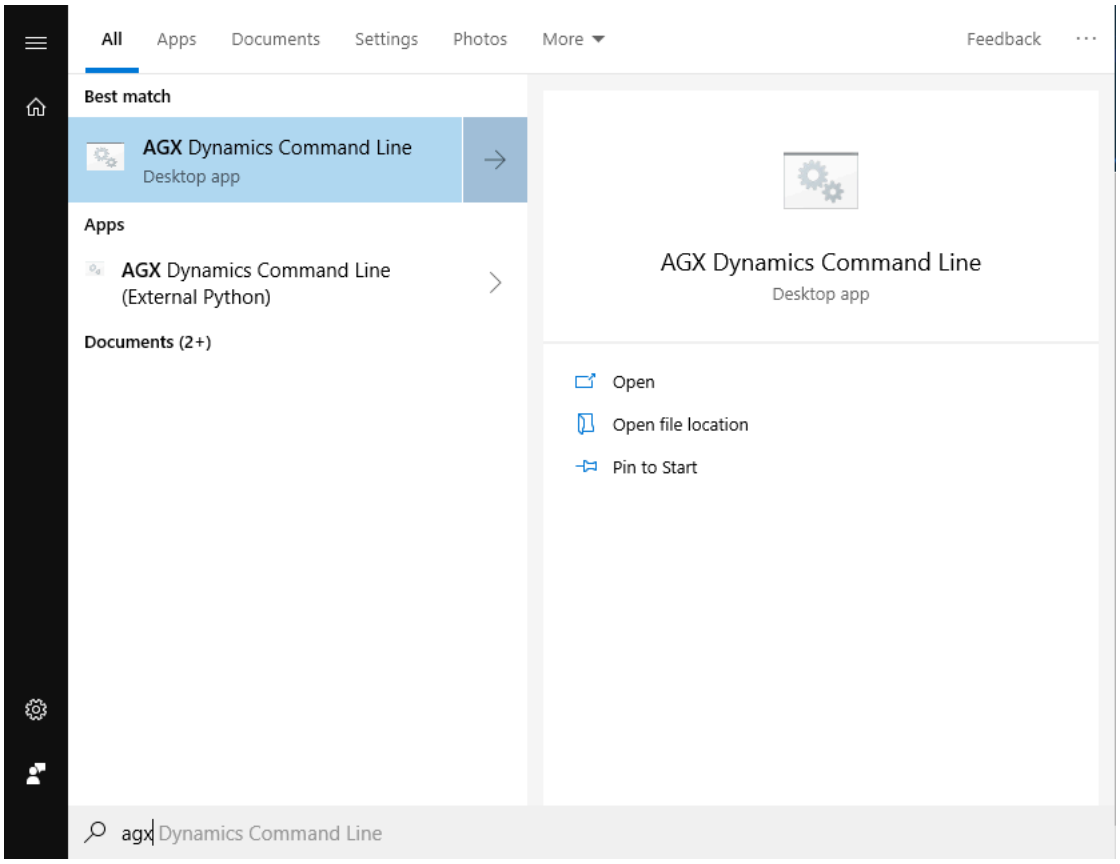
When building the tutorials which come with AGX, the following steps have to be followed to build in-source (in the installation folder):

1. Install cmake (https://cmake.org) Make sure you have CMake in your PATH (there might be an option for that during the installation).
2. Start an AGX Dynamics Command Line: Press the START button and type AGX (and it should appear if you are running Windows 10 or later).
3. Create an empty directory somewhere on the disk and make it the current directory.
4. Execute the command *cmake-gui* (make sure you have CMake in your path) and supply the AGX installation directory as an argument: `cmake-gui %AGX_DIR%`
5. Press configure and *select the correct generator* (VS2019, VS2022, VS2026; *64* bits) *that matches your installed AGX Dynamics binary distribution*. If not, you will get linking or runtime problems.
6. Change CMAKE_INSTALL_PREFIX to “.” (just the dot).
7. Press Generate and close cmake-gui.
8. Start Visual Studio with environment variables from setup_env: `devenv /useenv agxBuild.sln`
9. Build the solution
10. Building the project INSTALL will copy the files to the folder earlier given by CMAKE_INSTALL_PREFIX.

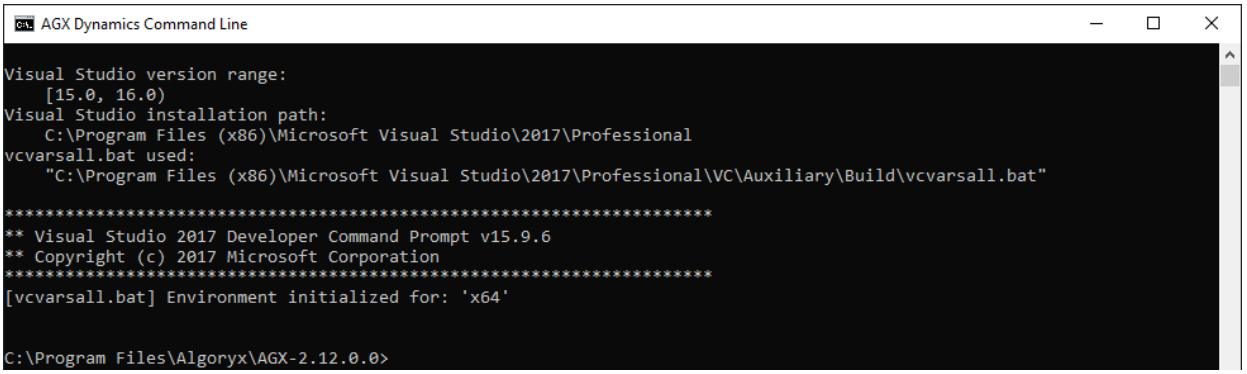
11. Now you should be able to run and debug the tutorials. Select one of them (e.g. tutorial_cable) as a StartUp-project by right-clicking on it in the solution explorer, and choosing “set as StartUp-project”.
12. Run externally with ctrl-F5, or debug it with F5.
13. Additional options can be passed via right-click in solution explorer *–>properties–>debugging–>Command arguments*. One option would be -p in order to start the simulation in paused state (press ‘e’ to toggle paused state while running).

Below is a session where the tutorials are prepared for building:

First, start a cmd window with the AGX Dynamics environment setup. You can either start your own cmd window and execute the setup_env.bat that exists in the AGX installation path. Or you can start one from the START menu:



Which of the two you choose, depends whether you want to use the embedded Python environment that comes with AGX, or your own (version compatible) Python installation.




```
c:\Program Files\Algoryx\AGX-2.33.0.0> mkdir d:\temp\buildAGXTutorials
c:\Program Files\Algoryx\AGX-2.33.0.0> pushd d:\temp\buildAGXTutorials
d:\temp\buildAGXTutorials> cmake-gui %AGX_DIR%
```

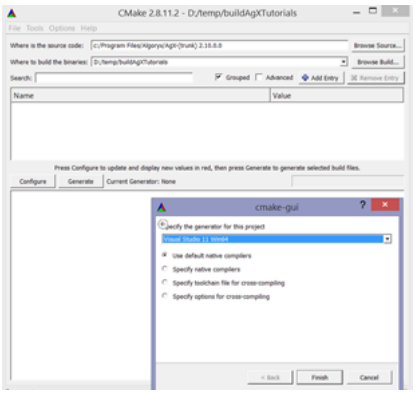


Fig. 5.15 Press Configure and select generator

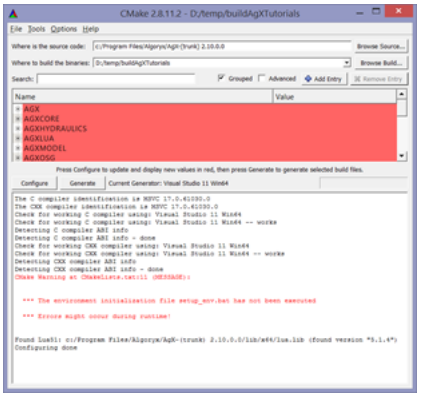


Fig. 5.16 Warnings might occur if setup_env.bat is not executed.

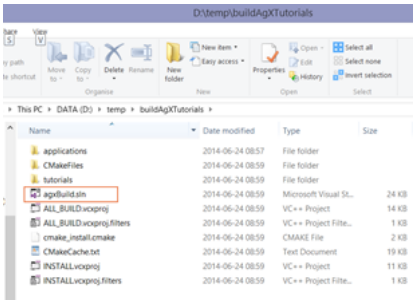


Fig. 5.17 agxBuild.sln is the resulting solution file.

```
d:\temp\buildAGXTutorials>
d:\temp\buildAGXTutorials>dir Release\tutorial_bodies.exe
d:\temp\buildAGXTutorials>Release\tutorial_bodies.exe
d:\temp\buildAGXTutorials>set PATH=%PATH%;c:\Program Files\Algoryx\AGX-2.33.0.0\bin\x64
d:\temp\buildAGXTutorials>Release\tutorial_bodies.exe

d:\temp\buildAGXTutorials>Release\tutorial_basicSimulation.exe
      AGX Library 64Bit AGX-2.33.0.0 Algoryx(C)
      Basic simulation tutorial
      --------------------------------


Creating a simulation
Creating a body
Adding body to the simulation
Step system 3 seconds forward in time
Shutting down AGX

d:\temp\buildAGXTutorials>
```

If an AGX Dynamics application is started without the correct path you might get an error message regarding missing runtime libraries:

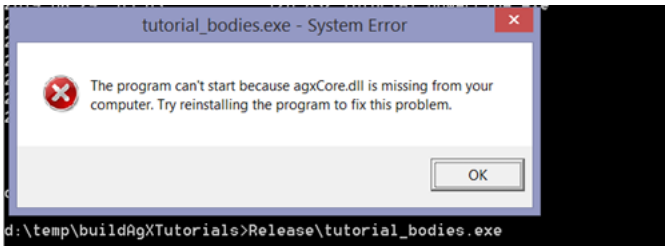


Fig. 5.18 Starting an AGX application without correct PATH.

5.2.2.1. Building HelloWorld in Windows

The directory HelloWorld contains a small code example which can be used as a starting point for a standalone project using AGX Dynamics. To build this example, just follow the steps below:

```
# create an empty directory
> mkdir c:\temp\myHelloWorld
> cd c:\temp\HelloWorld
:: Configure cmake and create makefiles for the tutorials etc.
> cmake -DCMAKE_INSTALL_PREFIX=%CD% c:\users\username\Algoryx\AGX-2.37.0.0\HelloWorld -A x64
:: Now run make using the created makefiles.
> msbuild HelloWorld.sln /p:configuration=Release /target:BUILD /p:Platform=x64 /p:useEnv=true /m
> Release\HelloWorld
*** Initializing AGX Dynamics...
*** Creating a simulation
*** Creating a RigidBody
*** Simulating
  0.0166667:   0 0 -0.00272407
  0.0333333:   0 0 -0.00817221
  0.05:        0 0 -0.0163444
  0.0666667:   0 0 -0.0272407

  ...

  0.466667:    0 0 -1.10597
  0.483333:    0 0 -1.18497
  0.5:         0 0 -1.26669
  0.516667:    0 0 -1.35114
*** Un-initializing AGX Dynamics...
```

5.2.2.2. Compiling (without CMake)

If you want to build an application without using the CMake build tool you need to specify a few pre-processor directives:

- Path to AGX headers and dependencies (usually <agx-install-dir>/include)
- Make sure that the WIN32 macro is defined: /D WIN32

The header file <agx/config.h> contains various settings that were used when AGX was built.

For Visual Studio you need to specify the include path to where the AGX headers are. Select your project and go to Project->Properties (Alt+F7). Specify the path to the include directory of AGX:

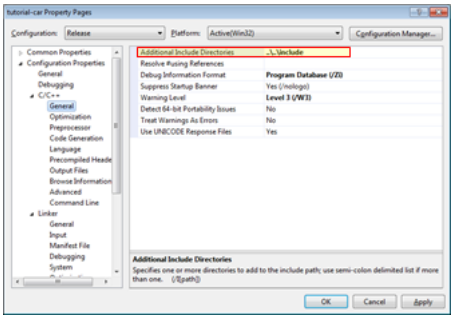


Fig. 5.19 Specify include path for AGX.

5.2.2.3. Linking (without CMake)

First you need to specify the path to where the libraries can be found. Make sure you choose the correct platform (x86/x64):

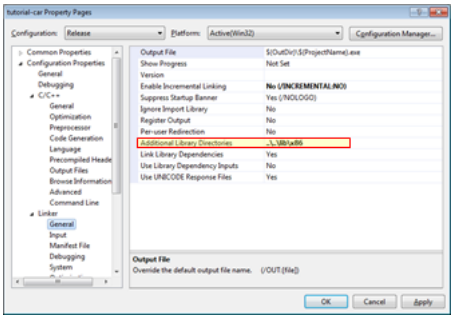


Fig. 5.20 Specify library path.

To be able to link an application to AGX you will at least need the following libraries: **agxSabre.lib, agxCore.lib** and **agxPhysics.lib**. The tutorials also require **agxOSG.lib**, and osg.lib, osgViewer.lib, osgDB.lib (OpenSceneGraph for rendering). Debug version of the libraries has a suffix 'd', for example **agxSabred.lib**. Additional functionality might require additional libraries. Such as Hydraulics, TireModel etc.

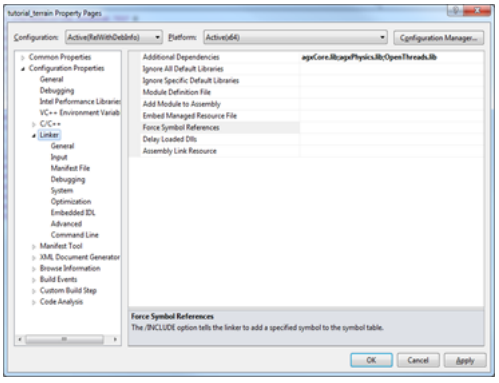


Fig. 5.21 Specifying AGX libraries.

5.2.3. Linux/gcc

To build the tutorials from source code, you need a compiler, development headers and optionally CMake.

The names for required and optional packages depends on which Linux distribution that is used. The lists here are for Ubuntu 20.04 and should be adapted to the Linux distribution/version you are using.

5.2.3.1. Linux tools for building AGX based applications

Package name	Purpose
build-essential	Required. Compiler related dependencies.
cmake	Recommended. Simplify generation of project/build files
cmake-curses-gui	Optional
cmake-qt-gui	Optional
ninja-build	Optional. Alternative to make
clang	Optional. Alternative compiler instead of GCC

5.2.3.2. Libraries for building/running AGX binaries

The AGX package bundles some dependencies and expects some to exist with the operating system. Note that some Linux distributions have separate packages for the shared library and the header files needed to compile and use said library.

Buildtime requirements:

Package name	Purpose
libgl-dev	OpenGL Development files
libglu-dev	OpenGL Development files

Runtime requirements:

Package name	Purpose
libglu1-mesa	OpenGL Releated
libssl1.1	Needed for websockets. Some earlier Ubuntu versions use libssl1.0.0
libpython3.8	Needed for python support. Use libpython3.6 instead on Ubuntu-18.04.
ttf-mscorefonts-installer	Recommended. Needed for in-application text to render properly.

5.2.3.3. Building tutorials on Linux

```
# create an empty directory
$ mkdir /tmp/agxBuild
$ cd /tmp/agxBuild
# Configure cmake and create makefiles for the tutorials etc.
$ cmake -DCMAKE_INSTALL_PREFIX=/tmp/agxBuild /opt/Algoryx/AGX-2.33.0.0
# Now run make using the created makefiles.
$ make
```

5.2.3.4. Building HelloWorld on Linux

```
# create an empty directory
$ mkdir /tmp/HelloWorld
$ cd /tmp/HelloWorld
# Configure cmake and create makefiles for the tutorials etc.
$ cmake -DCMAKE_INSTALL_PREFIX=/tmp/HelloWorld /opt/Algoryx/AGX-2.33.0.0/HelloWorld
# Now run make using the created makefiles.
$ make
```

5.2.4. MacOS

To build the tutorials from source code, you need a compiler, development headers and optionally CMake.

5.2.4.1. Tools for macOS for building AGX based Applications

Package name	Purpose
cmake	Recommended. Simplify generation of project/build files. Can be installed with Homebrew
ninja	Optional. Alternative to make. Can be installed with Homebrew
clang	Required. Installed with Xcode Command Line Tools or Homebrew

5.2.4.2. Building tutorials on macOS

```
# create an empty directory
$ mkdir /tmp/agxBuild
$ cd /tmp/agxBuild
# Configure cmake and create makefiles for the tutorials etc.
$ cmake -DCMAKE_INSTALL_PREFIX=/tmp/agxBuild /opt/Algoryx/AGX-2.33.0.0
# Now run make using the created makefiles.
$ make
```

5.2.4.3. Building HelloWorld on macOS

```
# create an empty directory
$ mkdir /tmp/HelloWorld
$ cd /tmp/HelloWorld
# Configure cmake and create makefiles for the tutorials etc.
$ cmake -DCMAKE_INSTALL_PREFIX=/tmp/HelloWorld /opt/Algoryx/AGX-2.33.0.0/HelloWorld
# Now run make using the created makefiles.
$ make
```

5.2.5. Executing an AGX application

To be able to run an application build with AGX, you need to specify the path to runtime libraries (PATH, LD_LIBRARY_PATH etc.). See the documentation for your specific platform.

You also need to specify where the components and other resource files, including the license file are located.

This can either be done using environment variables (see Section 47), or through the `agxIO::Environment` API.

For the Windows operating system, registry entries will be added by the AGX installer. These registry entries will be used whenever an AGX based application is executed.

The initialization of the AGX Dynamics API must be done before creation of any simulation related objects. This is because separate thread storages has to be initialized together with the underlying simulation framework. Initialization is done by using the function `agx::init()` and for shutdown `agx::shutdown()`.

5.2.6. HelloWorld.cpp

Below is a small example that demonstrates how to properly initialize AGX and create a simulation with one body.

```
#include <agxSDK/Simulation.h>
#include <agx/RigidBody.h>

int main()
{
  // ==========================================
  // 1. Specify path to resources
  // ==========================================

  // Where AGX is placed/installed:
  std::string pathToAGX = "c:\\Program Files\\Algoryx\\AGX-2.33.0.0\\";

  // Probably where the license file is
  agxIO::Environment::instance()->getFilePath(
    agxIO::Environment::RESOURCE_PATH).pushbackPath(pathToAGX);

  // Text files for plugins
  agxIO::Environment::instance()->getFilePath(
    agxIO::Environment::RESOURCE_PATH).addFilePath(pathToAGX);

  // binary plugin files
  agxIO::Environment::instance()->getFilePath(
    agxIO::Environment::RUNTIME_PATH).addFilePath(pathToAGX + "/bin/x64/plugins");

  // resource files (textures, models, python scripts etc.)
  agxIO::Environment::instance()->getFilePath(
    agxIO::Environment::RESOURCE_PATH).addFilePath(pathToAGX + "/data");

  // ==========================================
  // 2. Initialization of the AGX Dynamics API
  // ==========================================

  // AutoInit will call agx::init() which must be called before
  // using the AGX API creating resources such as bodies, geometries etc.
  agx::AutoInit init;
  {

    // ==========================================
    // 3. Create simulation content
    // ==========================================

    // Create a Simulation which holds the DynamicsSystem and Space.
    agxSDK::SimulationRef sim = new agxSDK::Simulation;

    // Create a rigid body (no geometry) with default mass etc.
    agx::RigidBodyRef body = new agx::RigidBody;

    // Add the body to the simulation.
    sim->add(body);

    // Set the time step to 1/100 (0.01 s or 100hz):
    sim->setTimeStep(1.0 / 100.0);
```

```
    // Simulate for 2 seconds
    while (sim->getTimeStamp() < 2.0)
      sim->stepForward();     // Take one time step.
  }

  // ========================================
  // 4. De-initialization (shutdown) of AGX
  // ========================================

  // The destructor for AutoInit will call agx::shutdown() automatically.
  // Unloads plugins, destroys threads etc.

  return 0;
}
```

5.2.6.1. Specify path to resources

Here we use the `agxIO::Environment` class to tell AGX where to locate resources. You are free to use '\' or '/').

Where to locate text/resource files for the AGX Dynamics simulation frameworks (Components):

```
std::string pathToAGX = "c:/Program Files/Algoryx/AGX-2.33.0.0\";
agxIO::Environment::instance().getFilePath(
      agxIO::Environment::RESOURCE_PATH).push_back(pathToAGX+"/bin/x64/plugins");
```

Where to locate binary files (.dll/.so):

```
agxIO::Environment::instance().getFilePath(
      agxIO::Environment::RUNTIME_PATH).push_back(pathToAGX+"/bin/x64/plugins");
```

Where to find data files such as models, textures, .cfg configuration files etc. Even python scripts will be located here:

```
agxIO::Environment::instance().getFilePath(
      agxIO::Environment::RESOURCE_PATH).push_back(pathToAGX+"/data");
```

You need to make sure that the license file (agx.lfx/agx.lic) exists in one of the directories specified for the RESOURCE_PATH. For more information regarding license files can be found in Section 6.1.

5.2.6.2. Initialization of the AGX Dynamics API

The function `agx::init()` will initialize the AGX API making it possible to create and destroy rigid bodies, constraints etc. One of the important tasks for agx::init() is it initialize the threading system. This will allow for fast local access of various data per thread.

The number of calls to agx::init() must match the number of calls to shutdown. I.e. if you call init three times, you also need to call shutdown three times before shutdown is actually performed.

Therefore it might be easier to use the `agx::AutoInit` class that will use the scope of the object variable to call init and a mathing shutdown.

5.2.6.3. Create simulation content

Then the API is initialized it is possible to create objects to build a simulation. See Section 11.

5.2.6.4. De-initialization (shutdown) of AGX

A call to `agx::shutdown()` should be the last AGX API call that is done. AGX threads will be destroyed together with the possibility to create new objects. Make sure that all RigidBodies, Constraints, Shapes etc. are all deallocated before a call to shutdown. Again, using `agx::AutoInit()` is a good way of getting a structured init/shutdown pair.

5.2.7. Dependencies

AGX Dynamics has several dependencies built as runtime libraries (.dll in windows). Table 5.1 below list the required runtime libraries for executing an application linked against AGX Dynamics under the Windows platform. Depending on which modules are used, more dll-files might be required.

Table 5.1 Required runtime libraries

RUNTIME LIBRARY	DESCRIPTION

RUNTIME LIBRARY	DESCRIPTION
AGXCORE.DLL	AGX Dynamics core library
AGXPHYSICS.DLL	AGX Dynamics core physics library
AGXSABRE.DLL	AGX Dynamics sparse blocked solver
COLAMD.DLL	Approximate minimum degree column ordering algorithm
LIBPNG16.DLL	PNG Image reader
MSVCP140.DLL*	Microsoft runtime libraries
MSVCR140.DLL*	Microsoft runtime libraries
VCRUNTIME140.DLL*	Microsoft runtime libraries
ZLIB.DLL	ZLib compression library
LIBPNG.DLL	PNG image library

*Depends on which version of Visual Studio that is being used.

Table 5.2 lists a few sample optional runtime libraries that might be used depending on the selected functionality in your application. If you use any classes from the agxModel name space, you will for example need to include the agxmodel.dll runtime library. The sample rendering framework utilizes OpenSceneGraph, which will require the files names osg*.dll. The table below is *not* a complete list of required runtime libraries. For a full list, please see the python script for copying a full runtime environment: Section 5.2.8.

Table 5.2 Optional runtime libraries

RUNTIME LIBRARY	DESCRIPTION	
AGXMODEL.DLL	AGX Dynamics model library	
AGXOSG.DLL	AGX Dynamics OpenSceneGraph wrapper	
AGXPYTHON.DLL	AGX Dynamics Python wrapper	
OSG*-OSG.DLL	OpenSceneGraph runtime library	
OSG*-OSGDB.DLL	OpenSceneGraph runtime library	
OSG*-OSGGA.DLL	OpenSceneGraph runtime library	
OSG*-OSGSHADOW.DLL	OpenSceneGraph runtime library	
OSG*-OSGSIM.DLL	OpenSceneGraph runtime library	
OSG*-OSGTEXT.DLL	OpenSceneGraph runtime library	
OSG*-OSGUTIL.DLL	OpenSceneGraph runtime library	
OSG*-OSGVIEWER.DLL	OpenSceneGraph runtime library	

Additionally, there is a list of data files *required* to run AGX Dynamics. These files contains *entity and kernel descriptions*. Entity is a core part of AGX and describes how data at a low level is stored and handled. Kernels describe execution units which operate on entities.

This data is located in the <agx-dir>/bin/<platform>/plugins/components.

5.2.8. Distributing an AGX Application

If you want to distribute a minimal runtime environment which contain AGX, you need to at least supply the following files:

- All files from Table 5.1 (located in `bin/x64/`)
- Files in `bin/x64/plugins/Components/` except for `bin/x64/plugins/Components/Web`
- `data/cfg/settings*.*`

Note

To distribute an application built with Visual Studio you need to make sure that the Visual Studio runtime libraries are available on the target machine. This can be managed by installing the Visual Studio redistributable package on the target machine. The redistributable package (vcredist_x64_*.exe) can be found either at the visual studio website or in the bin/x64 directory of the AGX Dynamics installation.

There is also a python script that can help you copy the required files (except for the Visual Studio runtime libraries) for running an AGX Dynamics based application. It is located at: `<agx-dir>/data/python/utilities/copy_runtimes.py*`

Depending on your specific use of AGX Dynamics, some of the runtime libraries will not be needed. It is up to you to choose only the ones that are required for your use.

The python script uses one argument: *a non-existing directory that will be created and filled with the required runtime files.* In the example below, the directory c:\temp\agx_runtimes will be created and all the required runtime files will be copied to that directory.

```
python c:\Users\username\AppData\Local\Algoryx\AGX-2.33.0.0\data\python\utilities\copy_runtimes.py c:\temp\agx_runtimes
```

5.3. Threads

AGX Dynamics is inherently multi-threaded. During runtime a pool of execution threads are utilized for various tasks. The number of threads are specified using the function `agx::setNumThreads(n)`. The number of threads can be changed during simulation. By default 1 thread will be used. A general recommendation is to use more than one thread, but not more threads than there are hardware cores.

One approach is to read out the number of cores:

```
/**
Limit core usage to number of physical cores.Assume that HT / SMT is active
and divide max threads with 2.
*/
const agx::UInt numCores = agx::SystemInformation::getNumCpu();
agx::setNumThreads(numCores / 2 - 1);
```

It is worth noticing that the call to `agx::SystemInformation::getNumCpu` might return the number of hyper-threads (if enabled in BIOS) which can be twice as many as the actual number of hardware cores.

Depending on the type of simulation the scalability of using more cores will vary. A tightly coupled simulation of one vehicle with many constraints/rigid bodies will not scale with more threads as well as a granular simulation with hundreds of thousands of granular bodies. AGX Dynamics has many tasks that can run in parallel, such as collision detection, integration, solvers etc. So one should usually see performance improvement using two threads over one, but at some point adding more threads will not improve performance.

If you are going to use the AGX Dynamics SDK in threads not created by AGX you need to make sure that you promote the thread to become an AGX thread. Reason is resource management. Some API calls will assume that the local thread has data that can be utilized. This is for performance reasons.

Therefore if you create a thread with `std::thread` or any other threading library you must promote the thread by registering it as an AGX thread using `agx::Thread::registerAsAgxThread()`. It is important that you also unregister the thread before the thread is destroyed. Below is an example that will use AGX API to load a mesh from disk in a separate thread:

```
#include "agx/agx.h"
#include <thread>
#include <agx/Thread.h>
#include <agxUtil/TrimeshReaderWriter/TrimeshReaderWriter.h>

/// Class that loads a mesh from file
class CreateTrimesh
{
  public:
    CreateTrimesh(const std::string& path) : m_path(path) {}
    void execute()
    {
      m_mesh = agxUtil::TrimeshReaderWriter::createTrimesh(m_path);
    }

    agxCollide::TrimeshRef getMesh() { return m_mesh; }

  private:
    agxCollide::TrimeshRef m_mesh;
    std::string m_path;
};

void execute(CreateTrimesh* creater)
{
  // Promote the thread to become an AGX thread
  agx::Thread::registerAsAgxThread();
  creater->execute();

  // Before we leave, unregister the thread!
  agx::Thread::unregisterAsAgxThread();
```

```
}


int main(int argc, char *argv[])
{
  agx::AutoInit agxInit;
  CreateTrimesh creator("models/box.obj");

  // Start the thread that loads the mesh.
  std::thread t1(execute, &creator);
  t1.join();

  std::cerr << "Num vertices: " << creator.getMesh()->getMeshData()->getVertices().size() << std::endl;
}
```

5.4. Reference pointers

AGX use a reference pointer system for all objects allocated on the heap. Reference pointers are a special kind of pointer that can only point to objects derived from agx::Referenced. A Referenced object keeps track of the number of reference pointers pointing to it. Whenever an instantiated class derived from Reference discovers that the number of referencing points is zero, it will automatically be de-allocated.

There are two types of reference pointers in AGX, ref_ptr and observer_ptr. ref_ptr is a *strong* pointer, which will during its lifetime keep the referenced object from being deleted. An observer_ptr on the other hand, will be notified when the referenced object is de-allocated. ref_ptr and observer_ptr both has a method isValid() that return false if it is pointing to a deallocated object. The type-cast operator is overloaded, so a ref_ptr<A> and observer_ptr<A> can be used anywhere in the code, as if it was of type A *. Example:

```
struct A : public agx::Referenced
{
};
agx::ref_ptr<A> a;
A* aPtr = a; // Ok, as the reference pointer will be cast to a A*
```

There are a few pitfalls when using reference pointers, an example:

```
// Problem #1: Local ref_ptr going out of scope
A* createA()
{
agx::ref_ptr<A> aRef = new A;

// When this function returns, aRef will go out if scope and the
// memory be deallocated. The number of reference pointers pointing to
// this object is decremented to zero, and it will be deallocated.
// So we are returning an invalid object!
return aRef;
}
```

In the above example one should either change the return type of the function to return an agx::ref_ptr<A>, or avoid using a ref_ptr inside the function. The latter solution will however be a potential memory leak, because memory will not be freed if an exception is thrown inside createA().

5.5. Math classes and conventions in AGX

There are numerous classes related to math operations and conversions in AGX. This chapter tries to describe the conventions used for these classes. **agx::Real** is the data type used for all floating point operations in AGX. Common for all rotations is that radians are used for all methods. **agx:degreesToRadians()** and **agx::radiansToDegrees()** can be used to convert between degrees and radians. The header file <agx/Math.h> contains utility methods related to math operations such as:

absolute(a)	Return absolute value of argument.
equivalent(a, b)	Return true if the two arguments are equal.
equalsZero(a)	Return true if argument is equal to zero.
minimum(a, b), maximum(a, b)	Return minimum or maximum of the two arguments.
clamp(a, min, max)	Return clamped argument a below min and max.
square(a)	Return square of a.

signedSqare(a)	Return squared argument a.
clampAbove(a, minimum)	Return clamped argument above minimum.
clampBelow(a, maximum)	Return clamped argument below maximum.
sign(a)	Return sign of argument.

5.5.1. Small vectors

Vec2, Vec3 and Vec4 are vectors for storing points and vectors in 2, 3 and 4 dimensions. Each method has methods for dot product, cross products and overloaded operators for +,-, scalar operations such as *, / etc.

5.5.2. Transformations: agx::AffineMatrix4x4

This is a 4x4 matrix that can store a concatenated rigid rotation and translation. It is stored in row order:

$R_x 0$	$R_x 1$	$R_x 2$	0
$R_y 0$	$R_y 1$	$R_y 2$	0
$R_z 0$	$R_z 1$	$R_z 2$	0
T_x	T_y	T_z	1

A vector-matrix multiplication is performed as: vM= M*v and its transpose is calculated as vM'=v*M.

The AffineMatrix4x4 is always considered to be orthonormalized, which means that the call to AffineMatrix4x4::inverse() will perform a transpose, as the transpose of a orthonormalized matrix is the same as the full inverse of the same. An AffineMatrix4x4 should never contain scaling or shearing, only rigid transformations.

5.5.3. Representing rotations

5.5.3.1. agx::OrthoMatrix3x3

This is a rotation matrix with the same conventions and methods (except everything related to translation) as the AffineMatrix4x4.

$R_x 0$	$R_x 1$	$R_x 2$
$R_y 0$	$R_y 1$	$R_y 2$
$R_z 0$	$R_z 1$	$R_z 2$

5.5.3.2. agx::Quat

Quaternion is a compressed representation of a rotation based on a vector and a scalar: q = [<x, y, z>, w], where <x, y, z> is a three dimensional vector and w is a scalar. Quaternions can be interpolated using linear interpolation: agx::lerp(t, q1, q2) or spherical linear interpolation: Quat::slerp(t, q1, q2);

5.5.3.3. agx::EulerAngles

EulerAngles is a more intuitive way of specifying rotations, as rotations can be specified per axis. There are 24 different conventions for representing rotations in Euler. This class can convert back and forth between all representations. Most classes can convert between the different rotation specifications.

5.6. Evaluation of AGX Dynamics

5.6.1. Running tutorials

The pre-built tutorials reside in the **bin** directory. They can be started from windows explorer just by clicking on them. Paths are setup so that the executables will look for data in the **data** directory.

Running applications linked to AGX without having setup the runtime environment (PATH) will lead to an application crash.

If you suspect that one more dll files are missing, use the **depends** tool from Microsoft (freely available on their website):

```
depends tutorial_basicSimulation.exe
```

It will give you a list of all the required runtime files necessary for running the application. It can also be run on a dll file:

```
depends agxPhysics.dll
```

which lists all the runtime dependencies for AGX main library.

5.6.2. Testing performance

When testing performance of a physics engine, it is important to remember the following things:

- Do not run with graphics. The agxOSG library will not run AGX in a thread separate from the graphics. To achieve this use the —agxOnly (-a) argument with agxViewer. For example:

  ```
  > agxViewer tutorial1_graphics.agxPy --agxOnly --stopAfter 10
  ```

  The above command would load and run tutorial1_graphics.agxPy without graphics for 10 seconds and stop and report the time it took to run the simulation.

- Do not use more threads than you have actual execution units on your CPU (cores). On some systems *hyperthreading* is enabled. This means that you will see more cores than what the hardware actually have. Hyperthreading works well with “light threads”, which does not do intensive float/calculations. You will notice a very uneven performance in AGX if you enable too many threads. What will happen is that you will spend a lot of time scheduling threads in and out of the cores leaving less time for actual calculations. This means that calling agxViewer –numThreads 10 will certainly NOT run faster than –numThreads 2 on a two core machine.

- Make sure that the machine is idle before starting your testing.

5.7. agxOSG – Binding to a rendering engine

Section 44.1.4 documents keybindings for the applications built using the agxOSG::ExampleApplication framework. The ExampleApplication is a collection of utilities found in the **agxOSG** directory as source. This can work as an example of an integration of AGX into a rendering engine (www.openscenegraph.org). agxOSG is mostly for internal use at Algoryx and is not very well documented.

Worth noticing is that rendering and simulation is executed from the same thread This means that if you want to measure the performance of AGX Dynamics only, you should run the tutorial/application without interactive graphics. This can be done for example through the arguments through the agxOSG::ExampleApplication class which all of the tutorials and AGXViewer is based on:

```
cmd> tutorial_customConstraints --scene 1 --stopAfter 10 --agxOnly --timeStep 0.01 --numThreads 2
```

The above command would do the following:

- start the application tutorial_customConstraints
- load scene 1 as defined in the application (–scene 1)
- run the simulation for 10 seconds (–stopAfter 10)
- Run as fast as possible with no graphics/window (–agxOnly)
- Use a time step of 100hz (–timeStep 0.01)
- Allocate and use two threads for AGX solver etc. (–numThreads 2)

6. License systems

To use AGX Dynamics, a valid license must be present.

6.1. License system

A license is acquired via an activation process where a *license id* and a *activation password* are used to perform an online activation that will result in a local license file.

The activation request will encode hardware information from the current computer. Hence a license is only valid on the machine where the activation was performed.

To help with the activation, a License Manager utility using Python is included with AGX. If PySide and Qt are available locally, the utility provides a GUI. If not, the utility can still be used as a command-line-interface (CLI) tool, see: Command line interface.

The utility, `LicenseManager.py` is located in `<agx-install-dir>/bin/${ARCH}` where `ARCH` is either `x64`, `x86` or empty - depending on the operating system.

Note

If you are using Windows, the license manager utility can also be launched via the Start menu. Just press the Windows button and write: `license` and you will find the AGX License Manager.

Note

To be able to perform license activation without having the install the full AGX SDK, there is also a standalone activation utility which has limited functionality compared to the license manager. The standalone utility is text based utility which can be used in a Windows command prompt or in a Unix shell.

Versions for all supported platforms is available in a zip at this location.

Running the utility without arguments or with the argument `--help` will provide usage information.

6.1.1. License Activation and Deactivation

To activate a license:

- Launch the LicenseManager utility
- Fill in the license id
- Fill in the activation password
- Change the output path if needed
- Click activate to generate a license file that unlocks AGX Dynamics on the computer/hardware where the activation took place.

The license file will be named *agx.lfx* and will be placed in the current users home directory.



Fig. 6.1 License manager running in a Unix operating system.

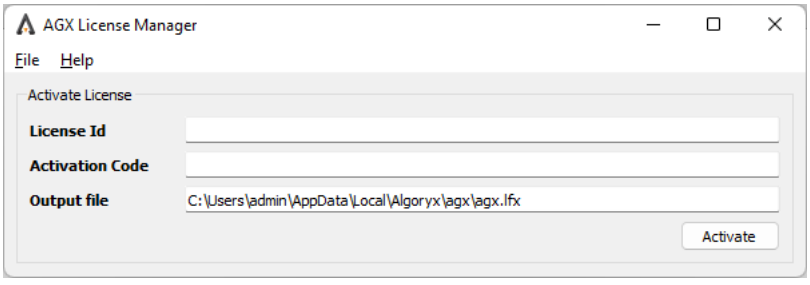


Fig. 6.2 License manager running in a Windows operating system.

After a successful license activation the License Manager will show information for the license including which modules that are enabled and how long the license is valid.

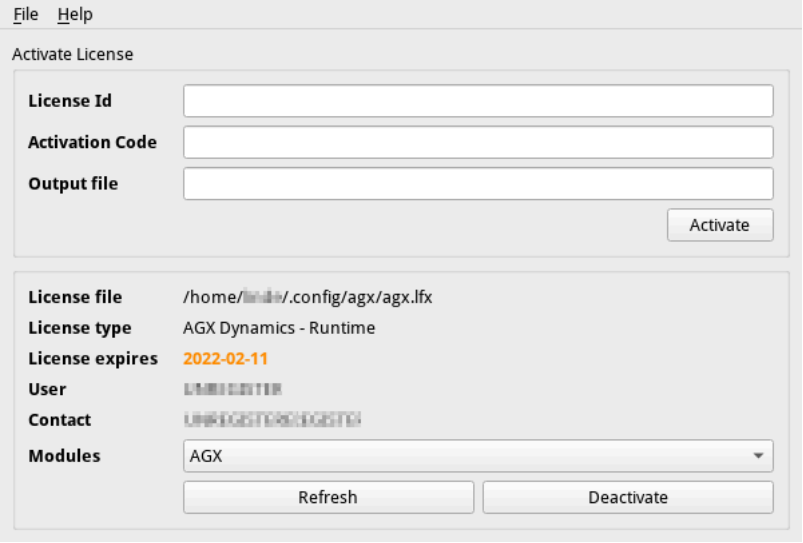


Fig. 6.3 Successful activation of the license.

6.1.1.1. Policies

Different license types have different policies.

6.1.1.1.1. Runtime license

A Runtime License cannot be deactivated by the user and normally does not have to be validated again online after the first activation.

6.1.1.1.2. Developer license

A personal developer license can be deactivated on one machine and activated on another machine i.e transferred. A license can only be activated a limited number of times (10). When the maximum number of activations are reached you need to contact Algoryx Simulation.

Developer licenses will also require an online heartbeat that will periodically refresh the license from the server.

6.1.2. License file location

By default, AGX will look for a license file named `agx.lfx`. AGX will try to locate the file automatically via the `RESOURCE_PATH` and those directories are initialized from the environment variable `AGX_FILE_PATH`. This setup is done via the provided `setup_env` scripts.

AGX could be installed where the current user does not have write access. This could be for example under the `c:\Program Files` directory in Windows or under `/opt/Algoryx/` in Linux. Therefore, AGX also searches in the current users home-directory for license files.

In Windows this is the `Algoryx\agx` subdirectory to the current users `LocalAppData` folder (`%USERPROFILE%\AppData\Local`). Full path example: `C:\Users\<MyUsername>\AppData\Local\Algoryx\agx\`.

In Linux we are adhering to XDG specification and the directory `${XDG_CONFIG_HOME}/agx/` is used as the default storage directory for the license file. If that environment variable is not defined, `~/.config/` is used in its place resulting in e.g. `/home/<MyUsername>/.config/agx/`.

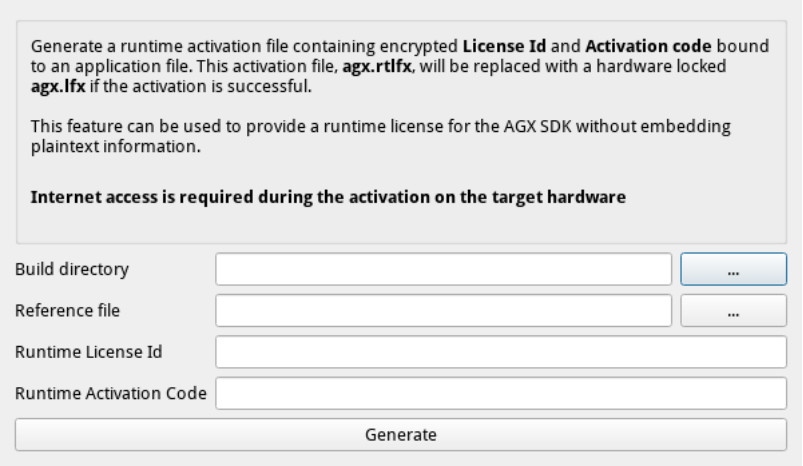
Read more on locating files in Locating files and resources.

6.1.3. Runtime deployment

This section describes the workflow for providing a Runtime license to an end-user.

When deploying an application using AGX Dynamics, activation must be performed on the end-user machine. We do not recommend embedding of any plain text information in your application. We therefore provide a way to encrypt the runtime activation information.

Selecting *File* > *Encrypt runtime…* opens the following dialog:



The *build directory* is where your built application resides. The reference file should be a relative path to a static file that is part of your application. Using the button … to browse for directory and file will fill in a path relative the build directory.

The static file is used during the activation. If the static file is changed, the encrypted runtime has to be regenerated to be usable. When clicking generate, the file *agx.rtlfx* will be saved to the build directory.

Note

The runtime activation id and activation password are not tested online when generating the encrypted runtime information. Make sure to enter the information correctly.

This file can then be used with the `agx::Runtime::activateEncryptedRuntime( rtflx_path, output_license_path )` method.

Example code:

```
auto runtime = agx::Runtime::instance();

if ( runtime->isValid() ) {
  // We have a valid license already.
  return;
}

agx::String rtflxPath = .... ;
agx::String outputLicensePath = ... ;

bool status = runtime->activateEncryptedRuntime( rtflxPath, outputLicensePath );

if ( !status ) {
  LOGGER_WARNING() << "Could not activate runtime: " << rt->getStatus() << LOGGER_ENDL();
}
else {
  LOGGER_INFO() << "Runtime license: " << outputLicensePath << LOGGER_ENDL();
}
```

6.1.4. Offline Activation

Activating AGX and receiving a license is an online process. It is still possible to activate without internet.

Part one:

- Start the license manager
- Fill in license id
- Fill in activation password
- Select File > Offline activation > Generate activation request…
- Fill in a filename for the activation request that will be stored to file.

Part two:

- Take the generated request file on a USB stick to another computer that has internet access.
- Upload the file or its contents to the manual request activation site.

- Save the response to a new file on the USB stick.

Part three:

- Fill in or change the *Output file* field in the License Manager
- Select File > Offline activation > Process activation response…
- Select the file with the activation response on the USB stick.
- A license file should be created locally for futher use.
- The license will be tied to the machine that created the offline activation request.

6.1.5. Command line interface

If the required python libraries are not installed, including PySide2, the python script LicenseManager.py can be used with a command line interface. Below is the available command line options:

```
AGX-2.36.0.0> python bin\x64\LicenseManager.py

Error: Could not import PySide2 for Qt GUI. Only command line usage is available:

usage: LicenseManager [-h] [--list] [--activate ActivationId ActivationPass OutputLicense] [--deactivate LicenseFile] [--
refresh LicenseFile]
                      [--create-activation-request ActivationId ActivationPass OutputFile] [--process-activation-response
ReponseFile LicenseFile]


Utility to manage licenses for AGX Dynamics

optional arguments:
  -h, --help            show this help message and exit
  --list                Search for license files and show license information
  --activate ActivationId ActivationPass OutputLicense
                        Activate a license and store the license to file
  --deactivate LicenseFile
                        Deactivate a license and remove file
  --refresh LicenseFile
                        Perform a manual refresh (online check) of license file
  --create-activation-request ActivationId ActivationPass OutputFile
                        Generate a license activation request on a disconnected machine.
  --process-activation-response ReponseFile LicenseFile
                        Process the response from an activation request.
```

To activate a license using the license id and an activation code:

```
AGX-2.36.0.0> python bin\x64\LicenseManager.py --activate 25556071 2x3D6Y6C agx.lfx


Performing license activation
License file:  agx.lfx
        User:         John Doe
        Contact:      john.doe@email.com
        Valid until:  2023-08-07
        Modules:      AgX
                      AgX-Particles
                      AgX-Cable
                      AgX-CableDamage
                      AgX-Wires
                      AgX-WireLink
                      AgX-Hydraulics
                      AgX-Hydrodynamics
                      AgX-DriveTrain
                      AgX-Tires
                      AgX-Tracks
                      AgX-Terrain
                      AgX-Granular
```

```
                    AgX-Simulink
                    AgX-MultiWires
```

Available licenses can be listed using the *–list* option which give the same output as above.

6.1.6. Using AGX in Docker containers

To use AGX, a license is required and this license is acquired via the activation process. As mentioned in an earlier section, the activation request will contain hashed hardware information and the issued license will contain the same identifiers.

If AGX is being activated from within a docker container, then container information will be part of the hashed hardware information. One can therefore not move license files between different docker containers since the hardware information will not match. It is also not possible to move a license while from the host machine into the container or vice versa.

6.2. Old License system

This chapter describes the previous license system utilizing agx.lic files.

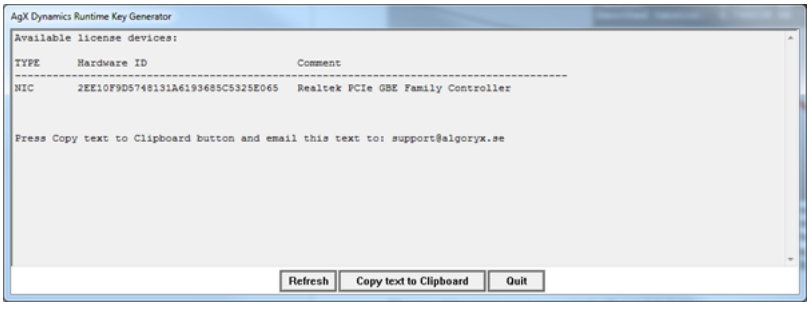
An application built with AGX requires a valid license to run. AGX can either be licensed with a *runtime*-license, or a *developer license*. A runtime-license usually does not have a time limit, but needs to be tied to a network adapter in your computer.

License information for a node-locked license is generated through a four step process:

1. Download the license executable available for all platforms here.
2. Next, execute the **RuntimeKeyGenerator** that correspond to your platform. This will generate a number of Hardware ID:s based on the available network adaptors (NIC).
3. Copy the list of the Hardware ID:s and send the following information to support@algoryx.com:
    - User/company name
    - Contact/email
    - HardwareID (select one whole line from the output of RuntimeKeyGenerator)
    - Url (Personal or your company's)
4. You will receive a file in email which should be named agx.lic that you should place in the installation directory of AGX Dynamics, or one of the directories specified by the environment variable AGX_FILE_PATH, or with the AGX API call `agxIO::Environment::getFilePath(agx::Environment::RESOURCES)` (see Section 46).

6.2.1. Sample Windows session

Below is output from the RuntimeKeyGenerator.exe:



6.2.2. Sample Linux session

Output from a typical Linux session:

```
johnd@ubuntu:~$  ~/Downloads/RuntimeKeyGenerator-amd64-ubuntu-20.04

Available unique ids:

TYPE  Hardware ID                      Comment
-------------------------------------------------------------------------------
NIC   36A55A36955110F55967514527F8FFD3 wlo1

One of these Hardware Id value(s) should be used when acquiring
a license from Algoryx
```

```
johnd@ubuntu:~$
```

6.2.3. License deployment

This section describes the workflow for shipping the license with your end-user application.

When deploying an application based on AGX Dynamics, the application must be accompanied by a valid license issued from Algoryx. This license must be a commercial valid runtime license intended for the application as written in the AGX Dynamics license agreement. A Subscription or Developer license must NOT be used for deploying applications to third-party customers.

6.2.3.1. License file

To be able to run an application with AGX Dynamics a valid license is required. This license file is distributed as a text file agx.lic as part of the licensing procedure. During internal development, an OEM/Developer license can/should be used. Usually the license file (agx.lic) is placed into a directory where AGX can locate it (see Section 5.2.5). A license file contains a number of fields:

- User: Name of the user/company/scope
- Contact: email address to the licensee
- EndDate: Last date when this license is valid
- HwKey: List of hardware keys for which this license is valid
- Modules: List of AGX modules
- Version: For which version of AGX this license is valid
- EndOfUpdate: Compared to the build date of AGX Dynamics and specifies the last date for which this license is valid.

6.2.3.2. Runtime license deployment

When an application is being deployed, we recommend *that you do not ship the agx.lic on disk*. The main reason for this is to avoid leaking the license file in clear text on the internet. A runtime license usually has the following properties:

- EndDate: None - meaning it will be valid forever
- EndOfUpdate - If set, it indicates that this is a runtime license with maintenance/update period.
- A specified AGX version: indicates which version of AGX is being utilized in the application. For example: “2.25.0.3”

Note

To reduce the risk of having a license file distributed over the internet, we strongly recommends that an obfuscated (non-readable text) license file is used.

6.2.3.3. Validate using obfuscated license file

The process of obfscating a license is where you take the original content of a license file and run it through a tool which generates a long string which represents the license information.

To convert a license file into an obfuscated license string you use the *RuntimeKeyGenerator* executable which is available for all platforms at the following link. For version 2.26.0.0 and later it is also available in your AGX installation. To obfuscate a license file you use the command:

```
> RuntimeKeyGenerator --obfuscate agx.lic obfuscated.txt
```

The content of obfuscated.txt can now be copy-pasted into your source code and used as an argument to the verifyAndUnlock method of the Runtime class:

```
const char* licenseStr =
“rF3pUP9LX+m0lP6IFoYOMBaLIq8RvYmPFro0vjjdCjP6WK+rFoXVB6k0cMb0kw2iZa9NoA2irw7PgcwAN1YF46P/Wh3623cjEuXZ8/coslPalc5e/p+Z8G/DAPz
// Unlock AGX dynamics with the obfuscated license
bool isValid = agx::Runtime::instance()->verifyAndUnlock(licenseStr);
```

6.2.4. Validating using license in clear text

The second way of validating a license file is to use the text in the license file without obfuscating. **However we do not recommend using this as the license text will be embedded in the binary as text strings.**

```
// Read the content of the license file into a string variable
std::string licenseFileContent = read_license_file_content();
```

```
// Unlock using the text string containing the license file
bool valid = agx::Runtime::instance()->unlock( licenseFileContent.c_str() );
if (valid)
  // We got a valid license file
```

7. Dimensions and units

Below is a table showing the dimensions and its associated units in AGX Dynamics. Where possible, SI units are being used. Many of the units are derived from the gravitational acceleration g. It has a default value in AGX Dynamics of 9.80665 m/s^2 and can be found in *agx/Math.h*.

Table 7.1 Dimensions and units used in AGX Dynamics.

Dimension	Unit
Acceleration	m/s^2
Angle	rad
Angular velocity	rad/s
Angular acceleration	rad/s^2
Cohesion	Pa
Compliance (rotational)	rad/Nm
Compliance (translational)	m/N
Damping [*]	s
Damping (rotational)	Nms/rad
Damping (translational)	Ns/m
Surface viscosity [†]	m/s/N or s/kg
Density	kg/m^3
Electric potential	V
Electric resistance	Ohm
Force	N
Length	m
Mass	kg
Moment of inertia	kg m^2
Pressure	Pa
Speed	m/s
Stiffness (rotational)	Nm/rad
Stiffness (translational)	N/m
Torque	Nm
Young's Modulus	Pa

[*]

Damping in AGX Dynamics refers to the *damping time constant*. It is a holonomic constraint-stabilization parameter for the time integration, expressed in seconds. It indicates roughly how quickly a holonomic constraint takes to dissipate energy and get back to the rest state; the damping effect usually scales with the simulation time step Δt.

[†]

Surface viscosity is a regulerization parameter for nonholonomic constraint that quantifies how easily a force generates velocity; the unit $m / s / N = s / kg$ denotes velocity per unit force (mechanical mobility).

8. Tutorials

8.1. C++

All source code (C++) tutorials can be found in the *tutorial* subdirectory. The tutorials are usually associated to a namespace and are to be found in the various subdirectories agx, agxCollide, agxOSG, agxSDK,…

A new tutorial is added by creating a source code file named tutorial_X.cpp/(tutorial_X.h) which will automatically be picked up by the build system and added to the target build.

Common to all tutorials in the agxOSG directory is the **ExampleApplication** class which encapsulates functionality for capturing keyboard events, creating example scenes etc.

These tutorials are all built around different “scenes”. The current scene can be selected at run-time with keyboard presses (0…9, F1-F10) or via a command-line argument: –scene [1…]

Building your first application is a good starting point for learning how to building the existing tutorials from source code.

8.2. Python

Many examples and tutorials are available also as Python code. This is a good way of learning how to use the AGX Dynamics API as it allows for quick changes/reloads without needing to compile/link C++ code.

- Python tutorials
- Python AGX Terrain tutorials
- Python AGX Granular tutorials
- Python AGX Wire tutorials
- Python Demos and examples
- Python AGX Terrain Demos
- Python AGX Gymnasium and Reinforcement Learning (RL)
- Python AGX ROS2 Examples

8.3. OpenPLX

OpenPLX is a high-level open source modeling framework. It is built by Algoryx and integrated in AGX Dynamics, designed to simplify the creation of physics-based simulations. It allows you to define simulation components using a structured and human-readable syntax, making it easier to build, extend, and control complex mechanical systems.

Explore the following tutorials to learn how to set up and extend simulations using OpenPLX:

- All OpenPLX Tutorials
- Introduction to OpenPLX: Boxtruck
- Introduction to Automatic Assembly
- Bulldozer D10 (Tracks, Drivetrain, Deformable Terrain)
- Build a Heavy Machine Bundle
- URDF Robot
- Signal Interface
- Lidar Autobrake
- Wheeled Vehicle

9. Creating stable simulations

Creating stable simulations is an art. It requires the juggling of a large set of parameters and attributes. This chapter will try to summarize a few of the most important things to keep in mind.

9.1. Masses

Depending on the selected solver (iterative or direct), the mass ratio between interacting bodies can become a problem. The default solver setting in AGX is a hybrid scheme where all constraints including ordinary constraints and contacts are handled with the direct solver, except for the friction of contacts, which are handled by the iterative solver. The hybrid solver is able to handle large mass ratios in stacking and constrained systems.

For the iterative solver (when specifying constraints or contacts to be solved iteratively) it is very important to keep the masses in the system in the same range of order of magnitude. For example a rigid body of mass 0.1 which interacts (collides with, is constrained to) with one that weighs 100 might cause instabilities due to large mass ratios. In general (but especially when using the iterative solver), try to keep mass ratios as low as possible, or at least for interacting objects. Having objects with mass 1gram in the same simulation as objects with 100E3Kg is ok, as long as they don't interact with each other.

9.2. Damping

In AGX Dynamics, contacts and constraints have an amount of damping, just like in the real world, due to friction, elasticity and other effects. Apart from modeling damped systems, damping has also an effect on the numerics and thus on the stability of the simulation.

Damping for contacts can either be set through the material properties;

```
Material::getBulkMaterial()::setDamping( )
```

or more preferably through the use of explicit ContactMaterial.

Damping is also important when using constraints. Assume for example that you have created a cylinder which is attached to the world with a hinge. If the hinge does not have a target speed-, lock- or friction controller enabled, nothing will stop the cylinder from rotating once it starts (unless the body itself has velocity damping enabled).

In real life, there is always friction in bearings, air-resistance etc. that will add "damping" to a system. This is something to keep in mind when you want to create stable simulations.

The damping coefficient set in AGX Dynamics for contacts and constraints is given in the unit of time, and differs from the typically used viscous damping coefficient. Use the functions in `agxUtil::convert` in order to convert from the viscous damping coefficient to AGX' damping coefficient. See 12.1.1 for more information.

A separate topic is the concept of *body velocity damping*, which is sometimes applied in simulations for numerical reasons, or to mimic aerodynamic or hydrodynamic effects. In AGX Dynamics, body velocity damping is disabled by default, since aerodynamic and hydrodynamic effects are better modeled with the dedicated library in AGX (see chapter 21).

However, velocity damping of a rigid body can be done the following way:

```
RigidBody::setVelocityDamping(double)
RigidBody::setLinearVelocityDamping(double)
RigidBody::setAngularVelocityDamping(double)
```

Valid values are between 0 and any real value, 0 is no damping (default). The velocity damping is a viscous damping.

9.3. Time step

The *time step* or *dt* used when stepping a simulation forward is very important. To get stable simulations one should use consistent time steps, that is, always stepping the simulation forward with same *dt*. Stepping a simulation with different *dt* might introduce energy and cause instabilities. Always strive to use the shortest possible *dt*, the shorter, the more stable simulations. Collision detection relies on that all collisions can be caught during one step. So if a geometry is moving fast relative to its size, or collides with a relative small geometry, penetration, or *tunneling* effects can occur. This will result in large penetrations or even missed collisions. Large penetrations will create large forces that will try to restore the penetrated geometry outwards.

Changing the time step is done via the `agxSDK::Simulation` object:

```
#include <agxSDK/Simulation.h>

// Create a Simulation which holds the DynamicsSystem and Space.
agxSDK::SimulationRef sim = new agxSDK::Simulation;
```

```
// Set the time step to 0.01s (100hz)
sim->setTimeStep( 1.0/100.0 );
```

Consider the simple case of two objects colliding below.

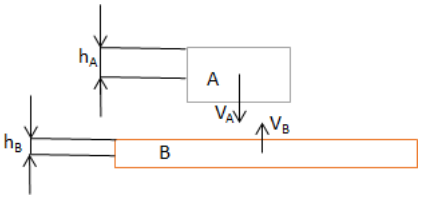


Fig. 9.1 Two objects moving towards each other.

In Fig. 9.1 we have two objects *A* and *B* that are moving towards each other and will at some point collide. Collision detection is done after moving the objects, so they will overlap to a certain degree. The object's relative velocity $V_{rel} = V_A - V_B$ will determine the total impact speed at collision. The distance these two objects will cover during one time step is $S_{dt} = V_{rel} * \Delta t$. We need to make sure that the time step is small enough so the following expression is fulfilled:

$$\Delta t: [S_{dt} << min(h_A, h_B)]$$

I.e. Δt is chosen small enough so that the distance the two objects cover during one time step is much smaller than the half extent of the smallest object.

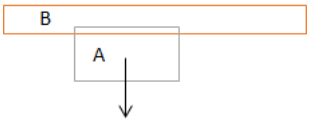


Fig. 9.2 Δt is too large. Object *A* will pass through object *B*.

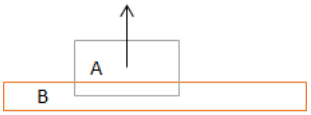


Fig. 9.3 Δt is small enough. Object *A* will bounce out of object *B* in a correct manner.

9.4. Size of objects

As described in the previous section: If objects are small in comparison to their velocity (and the used time step), they can cause deep penetrations and missed collisions. Try to keep an eye on the size of geometries so they are not too small. It's all relative to the time step and the velocity, so there is no right or wrong here.

9.5. Material attributes

Try to reduce the number of materials so that you have control over which combinations that can occur.

For example, if you have 100 objects (geometries) in your simulation and each geometry have different materials. This will lead to that you have ~100^2 combinations of materials. This is because for each pair of colliding geometries, a *ContactMaterial* is created from the two geometries materials. If you don't have control over your Materials, then you can get undesired behavior.

9.6. Velocities

Try to keep the velocities of objects moderate. Too high velocity in relation to time step and geometry sizes can cause penetrations.

9.7. Valid initial states

When creating a simulation, it is important that the initial state, which is how geometries are transformed relative to each other, is valid. An example of an invalid state is when two geometries are positioned initially in contact. This can result in an "exploding" simulation, where the solver will try to push the two interacting geometries/bodies out of each other.

Another example of an initial state that can cause unstable simulations is when using constraints. Assume you have created a rigid body, and this rigid body is also attached to a constraint. If the body is moved in such a way that it violates the constraint, after the constraint has been initialized, the solver will try to move the body so that the constraint stops being violated. For example, translating a body after it has been attached to a hinge and the world (only one rigid body used in the constraint).

Examples of invalid initial states:

1. Deep overlaps between geometries: Results in large forces -> high velocities

2. Unrealistic forces (on motors etc.)
3. Unrealistic velocities
4. Large violation on constraints

9.8. Using correct solve type

Fig. 9.4 show a scene from one of the tutorials (tutorial_direct_iterative_solver.agxPy) which illustrate the use of various solver configurations. AGX Dynamics has a high performance DIRECT solver which is capable of delivering very accurate and stable simulations. However, to achieve real time simulation for larger contact systems there might be a need to switch to the ITERATIVE contact/friction solver. By default AGX Dynamics use the SPLIT contact solver Section 3.4.3 which balances stability and performance for real time simulations in most situations.

When your simulation contains constraints which are solved with either the ITERATIVE or DIRECT solver, it is important to understand how to make the best out of the different variations.

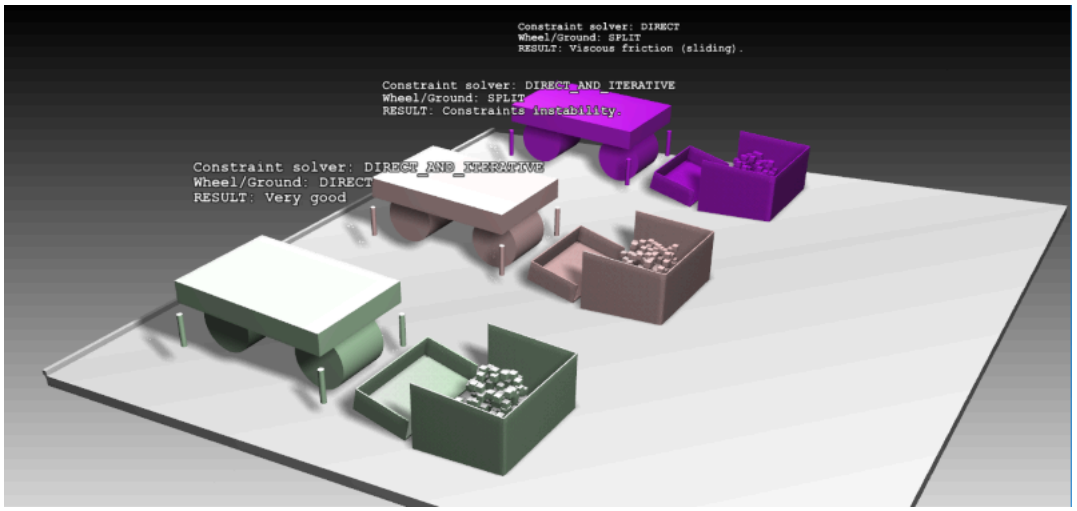


Fig. 9.4 Scene with three different solver configurations.

In the scene above there are three different set of vehicle/rocks. Each vehicle will drive forward, scoop up the “rocks” and back up and finally lift up the whole vehicle using the lifters in each corner.

Common for all of the three scenes are:

- ITERATIVE contact solver is used between rocks (for scalability)
- SPLIT contact solver is used between rocks and ground
- SPLIT contact solver is used between rocks and the shovel

Differences between the scenes are:

Scene 1: (purple)

- DIRECT solver is used for all ordinary constraints (Hinge, Prismatic etc.)
- SPLIT contact solver is used between the lifter/wheels and the ground

Result: *This scene demonstrates how the SPLIT solver will result in viscous friction due to the fact that the ITERATIVE solve for the friction is not accurate enough to resolve the frictional contact constraints.*

Scene 2: (brown)

- DIRECT_AND_ITERATIVE solver is used for all ordinary constraints.
- SPLIT contact solver is used between the lifter/wheels and the ground.

Result: *This scene demonstrate how the SPLIT solver in combination with DIRECT_AND_ITERATIVE constraint solver will result in unstable simulation. The error in the ITERATIVE part of contact solver will propagate to the ordinary constraints.*

Scene 3: (green):

- DIRECT_AND_ITERATIVE solver is used for all ordinary constraints.
- DIRECT contact solver is used between the lifter/wheels and the ground.

Result: *Here we get a stable simulation where all solvers are used in the best possible way. The ITERATIVE contact solver between rocks give good performance. The SPLIT solver between rocks and shovel and rocks and ground give good stability and acceptable performance. The*

DIRECT_AND_ITERATIVE solver for ordinary constraint will allow forces from the ITERATIVE system to propagate correctly to the DIRECT solver. The DIRECT solver between wheel/lifter and ground allow for stable and dry friction.

It would also be possible to use the ITERATIVE contact solver between rocks and the ground to improve performance. However, one possible artifact would be that the shovel could *push* the rocks into the ground/boxes due to large normal forces to which the ITERATIVE solver cannot handle correctly.

10. Creating a Simulation

The namespace agxSDK contains classes that bridges agx (dynamics) and agxCollide (collision). All insertion/deletion of objects such as rigid bodies, geometries, and materials should go through the main class agxSDK::Simulation to make sure that that everything is registered/executed in the correct manner.

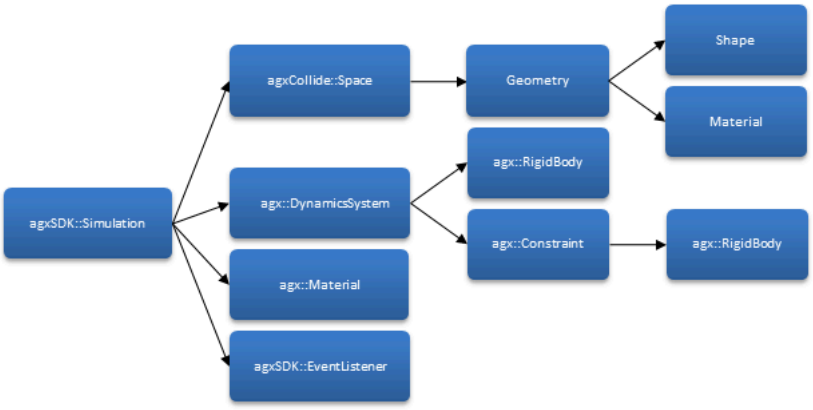


Fig. 10.1 Structural description of a Simulation

Fig. 10.1 show the structure of a simulation with the n-ary relationships between the various classes. An agxSDK::Simulation contain one agxCollide::Space and one agx::DynamicsSystem, it also contains any number of agx::Material and any number of agxSDK::EventListeners etc.

10.1. Simulation

The class **agxSDK::Simulation** is the class encapsulating most aspects regarding building and executing a simulation. This class holds a reference to an **agx::DynamicsSystem** that is responsible for integrating physical bodies and solving constraint equations, and an **agxCollide::Space** which generate contacts between geometries. All entities that should be part of a running simulation, such as Geometry, Constraint, and EventListener have to be added to the simulation. There can be several instances of the class Simulation, and they will each have their own Space and DynamicsSystem.

Attention

Always add/remove objects through the agxSDK::Simulation interface, not via DynamicsSystem or Space. However Space/DynamicsSystem can be used to access objects, such as getting a vector of all Geometries etc.

10.1.1. Timeline for Simulation::stepForward()

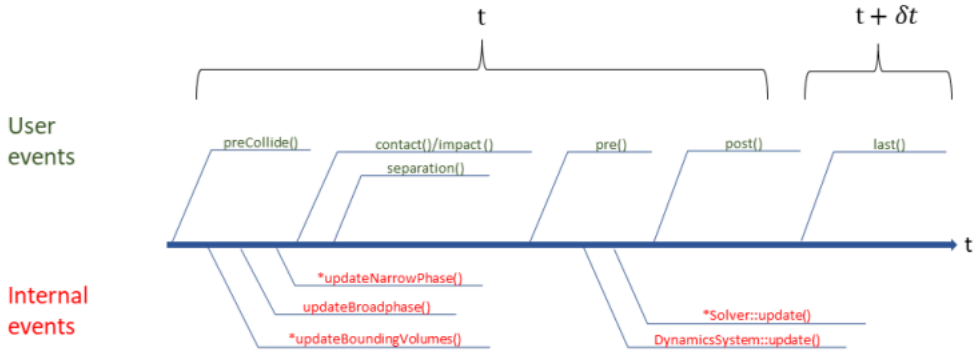


Fig. 10.2 Timeline illustrated for a Simulation::stepForward() call. (* indicates parallelizable steps).

Fig. 10.2 show a general picture of flow of events during a call to Simulation::stepForward(). An asterisk (*) indicate that the step is possible to parallelize. *User events* mean that it is possible to inject user code using EventListeners. In reality, there are many, many more tasks executed, many of them parallelizable.

Note

During the call to stepForward, the *simulation time* will be kept unmodified, except for the call to the `last()` callback. This means that the simulation time t will be incremented to $d + \delta t$ when it is time to call the `StepEventListener::last()` callbacks.

10.1.2. Initialization and shutdown of AGX

To properly initialize AGX, an initialization method needs to be called before creating an AGX object (rigid bodies, geometries, constraints etc.):

```
agx::init();
```

Various resources in AGX are handled by Singletons, classes that ensure that only one instance exists. AGX depends on various plugins (components etc.) which are all initialized at the call to agx::init. To properly call the destructors and shut down any threads the last call to AGX should be a call to agx::shutdown():

```
agx::shutdown();
```

The shutdown method is used to release any resources currently allocated by AGX. This includes loaded plugins, running threads, search paths (Environment) etc. **New in 2.3.0.0**: agx:init() can be called again after a call to agx::shutdown() to re-initialize AGX for use.

Attention

To properly shutdown any running threads, a call to **agx::shutdown()** should be done *before* the end of scope of main(). You should not register an atExit() callback and execute the call there, as this callback will be executed after the scope of main has ended. So always end your application with a call to the shutdown() method.

agx::AutoInit is a class that will upon construction call agx::init(), and upon its destruction call agx::shutdown(). This class can be used for handling init/shutdown within a scope. Hence, if an exception is thrown and not caught, the method agx::shutdown will still be called.

Just remember that any paths specified for agxIO::Environment will be lost after a call to the shutdown method, this goes for other resources too. You will need to re-initialize these to whatever value you want them to have, just as you did when you first started the application (initialized AGX).

10.1.3. Shutting down threads

Having running threads when leaving the scope of main can on the Windows platform cause hanging processes. Therefore, always make sure you either call agx::shutdown() or call the static method **agx::Thread::shutdown()** *before end of the scope of main*.

10.2. agx::DynamicsSystem

The DynamicsSystem is responsible for all rigid bodies and constraints in a simulation. It sets up a toolchain with solver, integration, etc. The DynamicsSystem stores an instance of an **agx::TimeGovernor** which is responsible for supplying a time step.

10.3. agxCollide::Space

Space handles geometric overlap tests such as broad phase tests for testing bounding volume overlaps, and near phase tests which result in detailed contact information: contact point, normal and penetration depth.

10.4. Events

Events are used for adding user code into the simulation. Types of Events implemented in AGX: *add/remove events, collision events, step events* and *gui* events. Event listeners are classes that can be implemented by the user through specialization of base classes supplied by the agxSDK namespace. All listeners are registered to the Simulation class and are triggered by the system whenever appropriate. All instances of the class EventListener have the methods:

- addNotification
- removeNotification

These methods are called whenever an event listener is added/removed from a simulation and can be used for initialization etc. Those methods cannot be filtered away, they will always be executed.

The method **EventListener::getSimulation()** is used to get a reference to the Simulation that the listener is currently registered to.

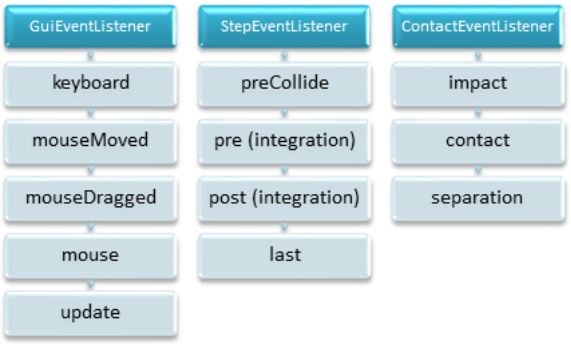


Fig. 10.3 Available event listeners and methods.

10.4.1. Filter events

Event listeners can be masked/filtered, so that they only are invoked for selected events. Each sub-class of agxSDK::EventListener has a method setMask() and a number of enums specifying for which events the class will be activated For example, specifying that a ContactEventListener should only react to IMPACT and SEPARATION events can be done as follows:

```
agx::ref_ptr<MyContactEventListener> l = new MyContactEventListener;
unsigned int mask = agxSDK::ContactEventListener::IMPACT|
       agxSDK::ContactEventListener::SEPARATION;

// Only react to IMPACT and SEPARATION events.
l->setMask( mask );
```

Some event listeners have more detailed filters. For ContactEventListener, see the subchapter “Filters” below.

10.4.2. Order of execution

The order in which event listeners are executed can be controlled using a priority. This is available for ContactEventListeners and StepEventListener, but not currently for GuiEventListeners. For EventListeners with the same priority, the order should be considered undefined.

```
simulation->add( listenerA );
simulation->add( listenerB );
```

In the above example, listenerA *can* be executed before listenerB, but the opposite is equally true.

Attention

The order in which Contact/Step event listeners are triggered should be considered undefined when they all have the same priority. This can affect for example a ContactEventListener that removes a contact.

There is no guarantee that other ContactListeners will not see that contact.

```
simulation->add( listenerA, 10 ) ;
simulation->add( listenerB, 20 ) ;
```

In the example above, listener B will be executed before listener A. Priority is only valid within the range:

```
[EventManager::LOWEST_PRIORITY, EventManager::HIGHEST_PRIORITY]
```

10.4.3. Collision Events

During the discrete simulation of a dynamic system, geometric interference calculations are done. They are done in two different steps, one for simpler bounding volumes (currently Axis-Aligned-Bounding-Boxes, AABB) testing for overlap, a process called *broad phase* test.

For each pair of AABBs that overlap, a more detailed *narrow phase* test is performed, which can have two outcomes: intersect or disjoint. Disjoint means that even though the bounding volumes overlap, there is still no intersection between the underlying geometries. If the result is intersect, contact data such as contact points, normals and penetration depth are calculated in a class named **agxCollide::GeometryContact**. Each intersection is stored in a list for later use (events and simulation).

The events generated by the *narrow phase* test are: *IMPACT, CONTACT* and *SEPARATION*. IMPACT occurs when two geometries intersect for the first time. CONTACT occurs when two geometries that in the previous time step caused an IMPACT event to occur. SEPARATION occurs when two geometries

that have intersected (and generated either an IMPACT or CONTACT event) are now disjoint.

Examples of event orders are:

IMPACT->SEPARATION->IMPACT->SEPARATION
IMPACT->CONTACT->CONTACT->SEPARATION

Collision events are caught by implementing a specialization of a class named **agxSDK::ContactEventListener** and overriding virtual methods with names corresponding to the events.

A ContactEventListener can also be told to listen only to a few of the above events by using the **setState()** method.

Table 10.1 Available virtual methods for ContactEventListener

EVENT	INTERFACE	DESCRIPTION
IMPACT	KeepContactPolicy impact(const agx::TimeStamp& agxCollide::GeometryContact *gc)	Called upon first impact between two geometries. No prior contact.
CONTACT	KeepContactPolicy contact(const agx::TimeStamp&, agxCollide::GeometryContact *)	Called when two geometries have narrow phase contact. Called after impact.
SEPARATION	void separation(const agx::TimeStamp& t, agxCollide::GeometryPair& cd);	Called when two previously overlapping geometries separates from narrow phase test.

```
class MyContactListener : public agxSDK::ContactEventListener
{
public:
  KeepContactPolicy impact( const agx::TimeStamp&,
                            agxCollide::GeometryContact *gc);
};
```

Attention

EventListeners only operate together with an agxSDK::Simulation. Explicitly calling Space::update() will NOT generate any callbacks to event listeners and might result in missing events.

10.4.3.1. Filters

A *ContactEventListener* is activated during contact between two geometries. The question is how to specify for which two geometries the listener should be activated?

agxSDK::ExecuteFilter is a base class which is specialized by a number of classes:

- **agxSDK::GeometryFilter** - can be used to specify which Geometry or pair of Geometries should trigger a listener.
- **agxSDK::PropertyFilter** (String, Int, Float, …) - can be used to specify a property or pair of properties that should trigger a listener.
- **agxSDK::RigidBodyFilter** - can be used to specify which RigidBody or pair of RigidBodies should trigger a listener.
- **agxSDK::RigidBodyGeometryFilter** - can be used to specify which combination of RigidBody and Geometry should trigger a listener.
- **agxSDK::CollisionGroupFilter** - can be used to specify which combination of collision groups should trigger a listener.
- **Your own** – by inheriting from **agxSDK::ExecuteFilter** and overriding some of its functions, you can specify your own criteria.

Examples of *ContactEventListeners* can be found in the examples/agxSDK directory. A filter is bound to the ContactEventListener:

```
agx::ref_ptr<MyContactListener> listener = new MyContactListener();
listener->setFilter( new agxSDK::GeometryFilter(geom1) );

simulation->addEventListener( listener ):
```

10.4.4. Modifying contacts

The methods impact and contact have a return type of **KeepContactPolicy**. This is an enum with the following values:

Table 10.2 Values for the enum KeepContactPolicy

KEEP_CONTACT	This contact will be kept and used in the contact solver. (if all other ContactEventListener also return KEEP_CONTACT for this contact)
REMOVE_CONTACT	This contact will be removed AFTER all other ContactEventListener have operated on it.

REMOVE_CONTACT_IMMEDIATELY	This contact will be removed directly after the return of this method, and no other ContactEventListener executed after this listener will see the contact.

So by using the values in Table 10.2 one can specify whether a contact should be kept or not after the callbacks has been executed.

Any data in an **agxCollide::GeometryContact** structure can be modified in a contact callback. If the methods *impact* and *contact* is to return REMOVE or REMOVE_IMMEDIATELY, the contact will be removed from the list of contacts.

10.4.5. Calculating relative velocity

Given that two geometries overlap in space, the result can be one or more intersection points.

The class **agxCollide::GeometryContact** contain all the information necessary for the solver to calculate the required impulses/forces to restore the overlap. This information can also be of use for the user of the toolkit. At a contact event, such as IMPACT, or CONTACT, the contact information is readily available as an argument to the event methods:

```
KeepContactPolicy impact( const agx::TimeStamp&,
            agxCollide::GeometryContact* gc);

KeepContactPolicy contact( const agx::TimeStamp&,
              agxCollide::GeometryContact* gc);
```

The argument *gc* is a pointer to a buffer containing all overlaps in the system after the last update to the collision detection system.

The actual overlap data is available as:

```
for( size_t i =0; i < gc->points().size(); i++)
{
  const agxCollide::ContactPoint& p = gc->points()[i];

  p.normal(); // contact normal
  p.point();  // point of contact
  p.depth();  // penetration depth
  agx::Vec3 contactVel = gc->calculateRelativeVelocity( i );
}
```

The last call to the method **calculateRelativeVelocity(i)** calculates the relative velocity between two colliding rigid bodies. The argument i specifies the index in the points vector. This method will return a velocity in world coordinates that can be used for calculating sound etc.

After the solver is done, the ContactPointBuffer will also be updated with state and force information for each contact:

```
const agxCollide::ContactPoint& p = gc->points()[i];
agx::Vec3 normalForce             = p.getNormalForce()();
agx::Vec3 frictionForce           = p.getTangentialForce();
agx::Bool isImpactingPoint        = p.hasState( agxCollide::ContactPoint::IMPACTING );
```

However, this information is not available in a contact event listener, as this event occurs before the solver is done. So to access this information, you need to get a reference to the vector of contacts directly from agxCollide::Space, or by storing a pointer to the ContactPoint which you are interested in.

Attention

It's not valid to keep references to contact data (`agxCollide::GeometryContact` and `agxCollide::ContactPoint`) between calls to `stepForward`. The objects are only views into data buffers and becomes invalid when the buffers are resized during collision detection.

Below is an example of how to access the actual vector containing points from within an event listener Remember that all event listeners has a method for accessing the simulation in which they exist:

```
const agxCollide::GeometryContactPtrVector& contacts = simulation->getSpace()->getGeometryContacts();
```

10.4.6. Step Events

A step event is generated before and after a discrete step is taken in the simulation. The two events generated are *PRE* and *POST*. **PRE** is generated after the collision detection phase but just before a step is taken in the dynamic simulation. POST is generated directly after a step is taken in the dynamic simulation.

agxSDK::StepEventListener is a base class which can be specialized by the user by inheritance.

Table 10.3 Available StepEventListener events

EVENT	INTERFACE	DESCRIPTION
PRE_COLLIDE	void preCollide(const agx::TimeStamp&)	Called before collision detection is performed. Last chance to add Geometries/Shapes to Space.
PRE_STEP	void pre(const agx::TimeStamp&)	Called before updating the dynamics information in the system (solver, integrating velocity/position).
POST_STEP	void post(const agx::TimeStamp&)	Called after the update of dynamics information. Bodies/Geometries have updated transformations.
LAST_STEP	void last(const agx::TimeStamp&)	Called last in Simulation::stepForward(). Simulation time is now incremented to $time + \delta t$. Operations executed in this stage will not be included into the timing for agxSDK::Simulation::stepForward().

10.4.7. GUI Events

GUI events are keyboard presses and mouse events. The Simulation class does not know of how to read of the mouse/keyboard for any specific system Instead an abstract class agxSDK::GuiEventAdapter can be specialized and implemented for a specific system which will inject appropriate events into the Simulation object An example of an implementation can be found in agxOSG::GuiEventAdapter which can be used together with OpenSceneGraph to generate GUI events.

To be able to receive GUI events a class named **agxSDK::GuiEventListener** is specialized and implemented. Examples of this can be found in *tutorials/tutorial_basic1.cpp*

10.5. agx::Frame

The class agx::Frame is used to handle multiple coordinate systems. This class contains a transformation (AffineMatrix4x4), velocities, parent/child relations and methods for converting points and vectors between local and world coordinate systems. Parent/child relationship can be constructed with the **Frame::setParent(Frame *)** method. This makes it possible to build up a scene of transformations.

Many classes in AGX such as RigidBody, Geometry, Assembly, etc. use an agx::Frame internally to store transformations and velocities.

An example of how to build a structure with parent/child relationship using Frames:

```
using namespace agx;
FrameRef parent = new Frame;
FrameRef child = new Frame;

child->setParent(parent);

parent->setTranslate( Vec3( 1, 0, 0) );

child->getTranslate(); // Now returns (1,0,0) due to its parent

parent->setRotate( EulerAngles(0, M_PI/2, 0) ); // Rotate 90 deg around Y.

Vec3 localVelocity( 1, 0, 0 ); // Local velocity, 1 in X.
Vec3 worldVelocity = child->transformVectorToWorld( localVelocity );
// World velocity is now (0,0,-1) as it is rotated around Y 90 deg.

child->getTranslate(); // Now returns (0,0,-1) due to its parents transformation.
```

10.6. Assembly

An agxSDK::Assembly is a class for logically (and coordinate wise) grouping objects together. Conceptually it could be viewed as a “bag” of things: You can fill the bag with objects put the bag somewhere and empty the objects onto the “floor”. When the simulation starts, a rigid body can move away from the “bag”. So if you ask the bag (Assembly) for its position, it is still where you left it. The rigid body however can move freely.

It can store a set of:

- agx::Constraint
- agx::RigidBody
- agxCollide::Geometry
- agxSDK::Assembly
- agxSDK::EventListener
- agx::ContactMaterial

- agx::ParticleSystem
- agx::Emitter
- agx::ObserverFrame
- agx::MergedBody
- agxSDK::TerrainInstance (baseclass for AGX Terrain)

  This can be useful when building larger constructions where a logical grouping is needed. An Assembly also contains a Frame which holds a transformation.

  Each RigidBody, Geometry, Constraint etc. will derive the transformations specified in the Assembly's Frame. So if an Assembly is moved `Assembly::setPosition()` all contained objects will also be moved accordingly.

  Schematic example of a function creating a car, returning a pointer to an assembly containing the whole car:

```
agxSDK::Assembly* buildCar()
{
  agxSDK::Assembly* parent = new agxSDK::Assembly();
  agx::RigidBodyRef chassis = new agx::RigidBody();
  // ...
  parent->add(chassis);
  parent->add(new agx::Material("rubber"));
  // ...
  return parent;
}
```

10.6.1. Adding/Removing an Assembly

An assembly can contain various objects such as RigidBody, Geometry, StepEventListener, GuiEventListener, ContactEventListener and Constraints.

When an Assembly is added to the simulation, all its contained items will also be added to the simulation. The same rule is applied when removing an Assembly from a simulation. This can however be controlled with the second argument to the Simulation::remove method:

```
bool Simulation::remove(agxSDK::Assembly *assembly, bool removeAllEntries = true );
```

An Assembly also has virtual methods that will be called when the assembly is added to or removed from a simulation `Assembly::addNotification`/`Assembly::removeNotification`. The python example below illustrates the add/remove notification that is used to execute some code as part of an assembly being added or removed.

```
class MyThing(agxSDK.Assembly):
    def __init__(self):
        super().__init__()

    def addNotification(self, simulation):
        print("The assembly is being added to simulation")

    def removeNotification(self, simulation):
        print("The assembly is being removed from simulation")
```

10.6.2. Derived transformation

When a RigidBody is added to an Assembly, the RigidBody derives the Assembly's transformation/velocity through a Frame/Parent interface. This means that a transformation made on the Assembly, will affect all rigid bodies contained in the Assembly. For example:

```
body->getFrame()->setLocalTranslate( agx::Vec3(1,0,0) ); // Set the transformation of the rigidbody

agxSDK::AssemblyRef assembly = new agxSDK::Assembly();
assembly->add( body );

// Move the assembly up in z 1 unit.
assembly->setPosition( agx::Vec3(0,0,1) );

// The line below will be true as body will inherit the assembly's transformation
assert(body->getPosition() == agx::Vec3(1,0,1));
```

10.7. Collection

The class **agxSDK::Collection** is derived from Assembly. It has the same functionality except for the transformation. A Collection does not contain any transformation and it will not take ownership of added objects Frame. Changing the transformation of a Collection has no effect on its children.

The Collection class is useful for collecting objects when reading from files, building entities where it is important to keep previous frame/transformation hierarchies.

11. Bodies and Shapes

11.1. RigidBody

An `agx::RigidBody` is an entity in 3D space with physical properties such as mass and inertia tensor and dynamic properties such as position, orientation and velocity. A rigid body is created as follows:

```
agx::RigidBodyRef body = new agx::RigidBody();
simulation->add( body ); // Add body to the simulation
```

There are three motion control schemes for a RigidBody:

Table 11.1 Motion control modes for a rigid body

DYNAMICS	Affected by forces, position/velocity is integrated by the *DynamicsSystem*.
STATIC	Not affected by forces, considered to have infinite mass, will not be affected by the integrator.
KINEMATICS	Controlled by the user. By supplying a target transformation and a delta time, a linear and an angular velocity are calculated. This velocity will be used by DynamicsSystem to interpolate the position/orientation of the kinematic RigidBody. The angular velocity is considered to be constant during the interpolation.

The type of motion control is set with the method:

```
RigidBody::setMotionControl(RigidBody::MotionControl);
```

11.1.1. Velocities

The angular and linear velocity of a rigid body is always understood as *the velocity of the center of mass*. Velocities of a rigid body are expressed in the world coordinate system.

```
agx::Vec3 worldVelocity(0,1,0);

// Specify the linear velocity in world coordinates for a rigid body
body->setVelocity( worldVelocity );

// Specify angular velocity for a body in LOCAL coordinate system
agx::Vec3 localAngular(0,0,1);

// Convert from local to world coordinate system
agx::Vec3 worldAngular = body->getFrame()->transformVectorToWorld( localAngular);

// Set the angular velocity in World coordinates
body->setAngularVelocity( worldAngular );
```

11.2. Velocity damping

AGX has a damping term (angular and linear) one for each coordinate axis (in the body's local coordinate system). It can be specified through the API as:

```
rigidBody->setLinearVelocityDamping( const Vec3f& damping );
rigidBody->setAngularVelocityDamping( const Vec3f& damping );
```

With this damping term *D* and the gravitational mass *m*, we will get a new mass matrix M' such that:

$$M^{'} = m\begin{bmatrix} (1 + D_x h) & & \\ & (1 + D_y h) & \\ & & (1 + D_z h) \end{bmatrix}$$

Analogous for the moment of inertia tensor.

Now given this effective mass matrix M' and the original mass matrix M we get the velocity integration equation:

$$V_{k+1} = M^{'-1} M V_k + M^{'-1} G^T \lambda + M^{'-1} h f_e$$

This means that if damping > 0, the term $M^{'}M$ will be < 0 and the velocity will be damped with a viscous damping term. The effective mass (including damping) will enter the system of equation on the left side, illustrated in the simplified linearized stepping equation below:

$$\begin{bmatrix} M^{'} & -G^{T} \\ G & \epsilon \end{bmatrix} \begin{bmatrix} V_{k+1} \\ \lambda \end{bmatrix} = \begin{bmatrix} -hf_{e} \\ g + G^{'}v \end{bmatrix}$$

For a more complete description see equation 4.31 in [2].

11.3. Effective Mass

When for example simulating a ship in water, the effective mass can be substantially larger than the actual weight of the ship. This is due to the water that is added by hydrodynamic effects. The mass varies depending on depth of water and is also different in different directions. The class `agx::MassProperties` has the capability of handling this added mass through the methods:

```
void setMassCoefficients( const Vec3& coefficients );
void setInertiaTensorCoefficients(const Vec3& coefficients);
```

This also means that AGX can handle objects that have different masses along each axis, so called *non-symmetric mass matrices*.

11.4. Modelling coordinates/Center of Mass coordinates

Center of mass is the point in the world coordinate system where the collected mass of a rigid body is located in space. This point can either automatically be calculated from the geometries/shapes that are associated to the rigid body, or it can also be explicitly specified to any point in space.

Since the center of mass moves when geometries are added (`if RigidBody::getMassProperties()::getAutoGenerate()==true`), it is not a practical reference when building a simulation. For example, if a constraint is attached to a body relative to the body center of mass, adding another geometry and thereby moving the center of mass would change to attachment point for the constraint. This is usually not the expected/wanted behavior.

Therefore AGX provides two coordinate frames, the model frame and the center of mass(CM) frame.

Fig. 11.1: shows one box with the model origin (axes) and the center of mass (yellow sphere) at the same positions. Fig. 11.2: illustrates the situation after another box geometry is added to the body, and thus the center of mass (yellow sphere) is shifted along the y-axis. The model origin (axes) is still at the same position.

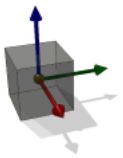


Fig. 11.1 Model and Center of mass at the same position.

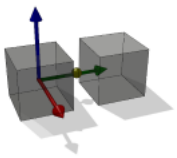


Fig. 11.2 Center of mass translated due to added geometry.

Note

AGX only supports relative translation of the center-of-mass-frame with regard to the model frame, but not relative rotation.

The translation, rotation and velocity of the center of mass for a rigid body can be accessed via the methods:

```
agx::RigidBodyRef body = new agx::RigidBody;

// Get position of Center of Mass in world coordinate system
Vec3 pos = body->getCmPosition();

// Get orientation of Center of Mass in world coordinate system
Quat rot = body->getCmRotation();

// Get rotation and translation of Center of Mass in world
```

```
// coordinate system
AffineMatrix4x4 m = body->getCmTransform();

// Or the complete frame at one call
Frame *cmFrame = body->getCmFrame();

// Translation relative to model coordinate system
Vec3 pos = body->getCmFrame()->getLocalTranslate();

// In most situation, it is the model coordinate system that is interesting. This can be accessed using the calls:
// Get position of Rigid Body in world coordinate system
Vec3 pos = body->getPosition();

// Get orientation of Rigid Body in world coordinate system
Quat rot = body->getRotation();

// Get rotation and translation of Rigid Body in world coordinate system
AffineMatrix4x4 m = body->getTransform();

// Or the complete frame at one call
Frame *cmFrame = body->getFrame();

// Get linear velocity of body
Vec3 v = body->getVelocity()

/**
\return the local offset of the center of mass position to the model origin (in model frame coordinates).
*/
agx::Vec3 getCmLocalTranslate() const;

/**
Sets the local offset of the center of mass position to the model origin (in model frame coordinates).
\param translate The new offset.
*/
void setCmLocalTranslate(const agx::Vec3& translate);
```

11.5. Kinematic rigid body

By specifying that a rigid body should be *kinematic*, we are telling the DynamicsSystem to stop simulating the dynamics of the body. This means that it will get an infinite mass, will not be affected by forces etc.

```
rigidbody->setMotionControl( agx::RigidBody::KINEMATICS );
```

It is also possible to interpolate the position/orientation based on a target transformation specified by the user. The body can then be specified to move from the current transform to a new during a specified time step with the call to:

```
agx::RigidBody::moveTo(AffineMatrix4x4& transform, agx::Real dt);
```

The above method will make a body move in a velocity calculated from its current transformation frame, to the specified during a time interval of *dt*.

When the interval *dt* is reached (current time + dt) the body will continue to move according to its velocity until next call to moveTo().

If you explicitly set a linear/angular velocity on a body, and specify it to be KINEMATICS, then that velocity will be used for integrating the transformation (position and orientation).

11.6. Geometry

A *Geometry* can be assigned to a rigid body to give it a geometrical representation. The geometry is also the entity that can intersect other geometries and create contacts. A Geometry contains one or more shapes. A *Shape* is specialized by *Box*, *Sphere*, *Capsule* etc. see table 11.4 for a complete list of available shapes.

11.6.1. Enable/disable

A Geometry can be enabled/disabled. A disabled geometry will not be able to collide with any other geometries. This is true for all the shapes in the geometry. *A disabled geometry will not contribute to the mass properties of a rigid body*.

```
void Geometry::setEnable( bool enable );
bool Geometry::getEnable() const;
bool Geometry::isEnabled() const;
```

Calling `geometry->setEnable(false)` will effectively remove the geometry from the mass calculation and collision detection.

11.6.2. Mass property calculations

A Geometry can also be disabled for mass property calculation. This means that it can still collide, but will not contribute to the mass properties of a rigidbody.

```
agx::RigidBodyRef body = new agx::RigidBody();
agxCollide::Geometry sphere1 = new agxCollide::Geometry(new agxCollide::Sphere(0.5));

agxCollide::Geometry sphere2 = new agxCollide::Geometry(new agxCollide::Sphere(0.5));
// This sphere will not be part of the mass property calculation for the rigid body
sphere2->setEnableMassProperties(false);

body->add(sphere1);
body->add(sphere2, agx::AffineMatrix4x4::translate(-1, 0, 0));

// Mass will still be the mass of one sphere.
// Same for Inertia
// cm_translate will be (0, 0, 0)
auto cm_translate = body->getCmLocalTranslate();

// Enable the second sphere for mass property calculation
sphere2->setEnableMassProperties(true);

// Update the mass propertices for the rigidbody
// Mass will now be 2x the mass of one sphere.
// Inertia and cm will also be updated
body->updateMassProperties();
```

11.6.3. Sensors

A geometry can be specified to be a *sensor*. This means that the collision system will generate contact information, such as contact points, normals and penetration, but no *physical* contact (contact constraint) will be created, hence the solver will not see this contact. A sensor can therefore penetrate any other geometry as a “ghost”. This can be useful for realizing certain functionality such as proxy tests, vision etc. A geometry is specified to be a sensor through the call:

```
agxCollide::GeometryRef geometry = new agxCollide::Geometry;
geometry->setSensor( true ); // Tell the geometry to become a Sensor
```

A sensor geometry will not add any physical properties such as mass, center of mass, or inertia tensor to the body it is part of. Calling **setSensor(true)** will effectively remove the geometry from the mass calculation.

In some cases it is enough to know that a sensor overlapped with some other geometry, without any need for the contact point data. Performance can be improved by not computing that data and instead letting the collider do an early out as soon as possible when an overlap has been detected. Contact data generation is disabled for a sensor geometry by passing **false** as the second argument to **setSensor**:

```
geometry->setSensor(true, false);
```

Disabling contact generation on a geometry that isn’t a sensor has no effect, contact points will always be generated for geometries that aren’t sensors.

Support for collider early out is added on a shape-type-pair basis and not all pairs are handled yet. Currently supported are:

- box-trimesh
- cylinder-trimesh
- sphere-trimesh

11.7. Enabling/disabling Contacts

There are several different alternatives to control whether pairwise contacts should occur or not.

- Disable all collisions for a geometry
- Enable/disable pair of geometries
- Geometry groups
- User supplied contact listeners

It should be noted that there is not direct priority between these different possibilities, but rather, if any of them disables the contact (even if others enable it), the contact generation will be treated as disabled.

11.7.1. Disable collisions for a geometry

To explicitly disable all collision generation for a geometry:

```
// Disable all contact generation for the geometry:
geometry1->setEnableCollisions( false );
```

11.7.2. Enable/disable pair

To disable/enable contacts for a specific geometry pair one can call the method `Geometry::setEnableCollisions( Geometry*, bool )`

```
// Disable contact generation between geometry1 and geometry2
geometry1->setEnableCollisions( geometry2, false );

// Enable contact generation between geometry1 and geometry2
geometry1->setEnableCollisions( geometry2, true );
```

The state regarding geometries being disabled/enabled for collision against each other will be stored in the geometries. So removing/adding them from/to a simulation, the result will still be the same.

11.7.3. GroupID

The class Geometry has a *groupID* attribute. This attribute can be set and read using the following methods:

```
void agxCollide::Geometry::addGroup( unsigned );
bool agxCollide::Geometry::hasGroup( unsigned ) const;
bool agxCollide::Geometry::removeGroup( unsigned );
```

With the method addGroup() one can assign a geometry to a specified group of geometries.

Together with the following method in agxCollide::Space:

```
void agxCollide::Space::setEnablePair(unsigned id1,
                                      unsigned id2,
                                      bool enable)
```

This functionality can be used to disable collisions between groups of geometries. To disable collisions between geometries of the same group:

```
space->setEnablePair( id1, id1, false );
```

After the above call, geometries belonging to *id1* cannot collide with each other. t is worth to notice that the group ids are stored in a vector, so searching and comparing has a time complexity of O(n). So before adding a large number of group ids to a geometry, make sure there isn't another alternative. This is because the geometry groups will be compared to each other during the collision detection process.

11.7.4. Named collision groups

New in 2.3.0.0 is that there is also named collision groups. This has the same functionality as GroupID, but it is defined using a string instead.

```
void agxCollide::Geometry::addGroup( const agx::Name& );
bool agxCollide::Geometry::hasGroup(const agx::Name& ) const;
bool agxCollide::Geometry::removeGroup(const agx::Name& );
```

This functionality can be used to disable collisions between groups of geometries. To disable collisions between geometries of the same group:

```
space->setEnablePair( name1, name1, false );
```

After the above call, geometries belonging to *name1* cannot collide with each other.

11.7.5. User supplied contact listener

A *ContactEventListener* (see 10.4.3) can be registered to listen to specific collision events, between specified geometries, geometries with specific properties etc. By implementing a collision event listener that for a given pair of geometries return REMOVE_CONTACT or KEEP_CONTACT immediately in the impact() method, one can achieve the same result as the previous two alternatives, namely, no contact is transferred to the contact solver. This is a very general way of specifying for which pairs contacts should be generated.

```
class MyContactListener : public agxSDK::ContactEventListener
{
public:
  MyContactListener()
  {
    // Only activate upon impact event
    setMask(agxSDK::ContactEventListener::IMPACT);
  }

  bool impact(const agx::TimeStamp& t, agxCollide::GeometryContact *cd) {

    RigidBody* rb1 = cd->geomA->getBody();
    RigidBody* rb2 = cd->geomB->getBody();

    // If both geometries have a body, and the mass of the first is over 10
    // then remove the contact
    if (rb1 && rb2 && rb1->getMassProperties()->getMass() > 10 )
      return REMOVE_CONTACT; // remove contact.

    return KEEP_CONTACT; // keep the contact
  }
};
```

Which geometries should trigger the ContactEventListener can be specified using an ExecuteFilter. This filter can also be user specified. By deriving from the class **agxCollide::ExecuteFilter** and implementing two methods, any user supplied functionality can be implemented for filtering out contacts:

```
class MyFilter : public agxCollide::ExecuteFilter
{
public:

  MyFilter() {}

  /**
  Called when narrow phase tests have determined overlap (when we have
  detailed intersection between the actual geometries).
  Return true if GeometryContact matches this filter:

  if both geometries have a body.
  */
  virtual  bool operator==(const agxCollide::GeometryContact& cd) const
  {
    bool f =
      (cd.geomA->getBody() && cd.geomB->getBody());
    return f;
  }

  /**
  Called when broad phase tests have determined overlap between the two
  bounding volumes.
  */
  virtual  bool operator==(const GeometryPair& geometryPair) const
  {
    bool f =
      (cd.geomA->getBody() && cd.geomB->getBody());
    return f;
```

```
  }
};
```

11.7.6. Surface velocity

A geometry can be given a surface velocity. It is specified in the local coordinate system of the geometry and can be used for simulating the surface of a conveyor belt. In its general form, it is independent of the contact point position relative to the geometry. If more control over that behavior is desired, it is possible to create a class inheriting from agxCollide::Geometry, and to override the method

```
agx::Vec3f Geometry::calculateSurfaceVelocity( const agxCollide::LocalContactPoint& point ) const.
```

One such class is `agx::SurfaceVelocityConveyorBelt`, which is described in Section 33.

11.8. Primitives for collision detection (Shapes)

AGX allows for different shapes for geometric representation. Simpler shapes such as Box, Sphere or Cylinder allow for exact representation of simple geometry, as well as fast compute times.

Mesh types such as Trimesh, Convex or HeightField allow for more complex geometry, at the cost of higher compute times.

11.8.1. Box

A box is defined by three half extents with its center of mass at origin.

```
Box::Box( const agx::Vec3& halfExtents )
```

11.8.2. Sphere

A sphere is defined by its radius and is created with its center of mass at origin.

```
Sphere::Sphere( const agx::Real& radius )
```

11.8.3. Capsule

A capsule is a cylinder, capped with two spheres at the end. It is defined by its height and its radius. The capsule will be created with its height along the y-axis and its center of mass at origin.

```
Capsule::Capsule( agx::Real radius, agx::Real height )
```

11.8.4. Cylinder

```
Cylinder::Cylinder( const agx::Real& radius, const agx::Real& height )
```

11.8.5. HollowCylinder

A hollow cylinder is similar to a cylinder but with a circular hole.

```
HollowCylinder::HollowCylinder( agx::Real innerRadius, agx::Real height, agx:Real thickness )
```

This shape is suitable for modeling peg-in-a-hole kind of scenarios in combination with other cylinders, hollowcylinders or different kinds of cones.

Note

A current limitation is that the hollow cylinder does not have full collision support against all other shape types. Until full support is available according to table 11.4, a fallback is used which treats the HollowCylinder as a Cylinder.

To give an example, the pair HollowCylinder - Sphere does not have full support and hence the sphere can not be sent through the hollow part of the cylinder due to the hollow cylinder being treated as a cylinder shape.

11.8.6. Line

A line is defined by two points, begin and end points:

```
Line::Line( const agx::Vec3& start, const agx::Vec3& end )
```

11.8.7. Plane

A plane can be defined in a number of different ways: either by a point and a normal, or the four plane coefficients:

```
Plane::Plane( agx::Real const a, agx::Real const b,
                 agx::Real const c, agx::Real d )
Plane::Plane( const agx::Vec3& normal , agx::Real d )
Plane::Plane( const agx::Vec3& normal, const agx::Vec3& point )
Plane::Plane( const agx::Vec4& plane)
Plane::Plane( const agx::Vec3& p1, const agx::Vec3& p2,
              const agx::Vec3& p3)
```

11.8.8. Cone / HollowCone

```
Cone::Cone( agx::Real topRadius, agx::Real bottomRadius, agx::Real height )

HollowCone::HollowCone( agx::Real topRadius, agx::Real bottomRadius, agx::Real height, agx::Real thickness )
```

A Cone and a HollowCone are very similar. The main difference is that the hollow cone has a conical hole. Both cone types support a non-zero topRadius which creates a truncated cone where the top part is cut off.

The cones are defined alony the Y-axis and the bottom disc is positioned at y=0 and the top point (or top disc in case of truncation) is placed at y=height.

Note

HollowCone and Cone does not currently support all other shapes. Until each collider is implemented to take into account the actual cone shape, a fallback treating them as a Convex will be used. See table 11.4.

11.8.9. Triangle mesh

11.8.9.1. Triangle mesh construction

The triangle mesh shape in AGX supports general concave triangle meshes. The internal storage of triangle meshes in AGX is based on *vertices* (array of 3D points) *and indices* (indexes into the array of vertices), where three concurrent indices define a triangle each (a *triangle list*). Trimeshes share a common base class called `agxCollide::Mesh` with another mesh-based collision shape called `agxCollide::HeightField`.

A trimesh can be created using code or by reading from a support mesh file format (see Mesh reader). Below is an example of how a trimesh can be created from code or from a file

```
agx::Vec3Vector vertices;
agx::UIntVector indices;

// Add vertices
vertices.push_back( agx::Vec3( -0.4, -0.4, 0.0 ) );
// ...

// Add indices
indices.push_back( 1 ); // starting first triangle
indices.push_back( 0 );
indices.push_back( 2 ); // finished first triangle
indices.push_back( 0 ); // starting second triangle, reusing one vertex
// ...

// Create the collision shape based on the vertices and indices
agxCollide::TrimeshRef triangleMesh = new agxCollide::Trimesh( &vertices,
&indices, "handmade pyramid" );
```

And from a file:

```
// Create the trimesh shape from a WaveFront OBJ file:
agxCollide::TrimeshRef trimeshShape =
  agxUtil::TrimeshReaderWriter::createTrimesh(
    "torusRoundForCollision.obj");
```

The data will be copied by the trimesh constructor.

Furthermore, the triangles are expected to be given in counterclockwise order (*counterclockwise winding/orientation*). This is important since the order of the vertices defines which face of the triangle will point outwards and which inwards. This orientation is used within calculation of mass properties and collision detection. If the data is given in clockwise winding, this can be indicated by setting the according flag in `agxCollide::Trimesh::TrimeshOptionsFlags`. Mixed winding should be avoided since it will lead to unexpected behavior. The flag `agxCollide::Trimesh::RECALCULATE_NORMALS_GIVEN_FIRST_TRIANGLE` can be used to adapt the winding to the first triangle, given that the mesh is watertight otherwise.

Internally, the triangle data will be stored in counterclockwise winding.

For good collision detection results, each trimesh should be a watertight (a geometrical manifold). Non-manifold trimeshes can be used, but the results may vary in quality. Non-manifold trimeshes may furthermore result in an invalid mass and inertia when used in dynamic rigid bodies for which mass properties are calculated automatically.

11.8.9.2. Triangle meshes as terrain

Trimeshes can also be used to model terrains. Here, they do not have to be closed on all sides, but may have an open direction. This direction is assumed to be upward in z direction in the trimesh's local coordinate system. The corresponding normal (0, 0, 1) is used as a tunneling safeguard, and the trimesh's bounding box is enlarged in opposite direction of this normal by a value called the terrain's "bottom margin".

It is recommended to avoid collisions at terrain borders unless they are met by a corresponding neighboring terrain, since this might give suboptimal results. The trimesh class will give out warnings during construction if a terrain does not have a rectangular shape (in the trimesh's local x and y coordinates), or contain holes. The usage as terrain should be indicated in the optionsMask.

One example for a terrain with a bottom margin of 5.0 below the mesh's deepest point in Z:

```
agx::Vec3Vector vertices;
agx::UIntVector indices;

// Add vertices
vertices.push_back( agx::Vec3( -0.4, -0.4, 0.0 ) );
...
// Add indices
indices.push_back( 1 ); // starting first triangle

uint32_t optionsMask = agxCollide::Trimesh::TERRAIN;
...
TrimeshRef terrain = new Trimesh( &vertices, &indices, "handmade terrain", optionsMask, 5.0 );
```

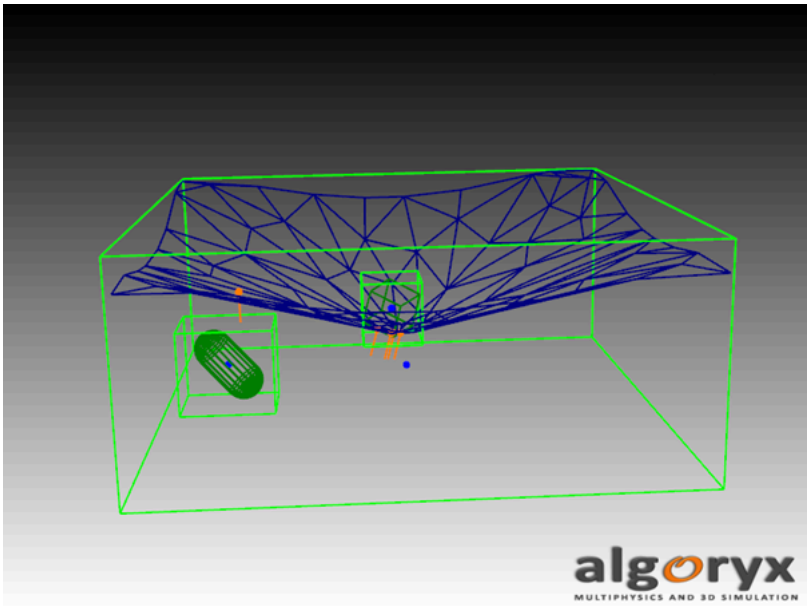


Fig. 11.3 Terrain/trimesh intersecting a box, and a capsule which is prevented from tunneling. Note the enlarged bounding box in opposite direction of the terrain normal.

Using a height-field for collision detection as a terrain has the advantage of being able to modify the terrain during run-time. Trimeshes as terrains, on the other hand, can be modeled with fewer triangles by adapting the level of detail to the local change in terrain height, so that large plain parts can be covered by few large triangles.

11.8.9.3. Triangle meshes modeling

As mentioned above, special attention should be paid when using triangle meshes as collision primitives.

It is important that the models are adapted for the special needs arising from physics simulation, and thus are either manifolds or used as terrain.

Potential problems (which should be fixed externally before using the trimesh) are:

- Unmerged identical vertices
- Holes
- Wrong/inconsistent winding
- Triangles with zero area
- Unreferenced vertices
- Self-intersection.

Trimeshes used in collision detection create potentially very many contact points. The use of *contact reduction* is therefore recommended.

The meshes in AGX normally have a half-edge structure. See further in Triangle mesh traversal. If the half-edge structure has edges with no neighbor, then it can be a sign of one or more of the problems listed above. As a help, there is a python utility which can be used to visualize missing half-edges.

It is either used with python such as `python data/python/trimesh-visualization.py path/to/mesh.obj` or used in combination with e.g. `agxViewer` by using `--attachScript data/python/trimesh-visualization.py`. The script will look for all trimeshes in the simulation, check for edges with no half-edge neighbor and debug-render these in red.

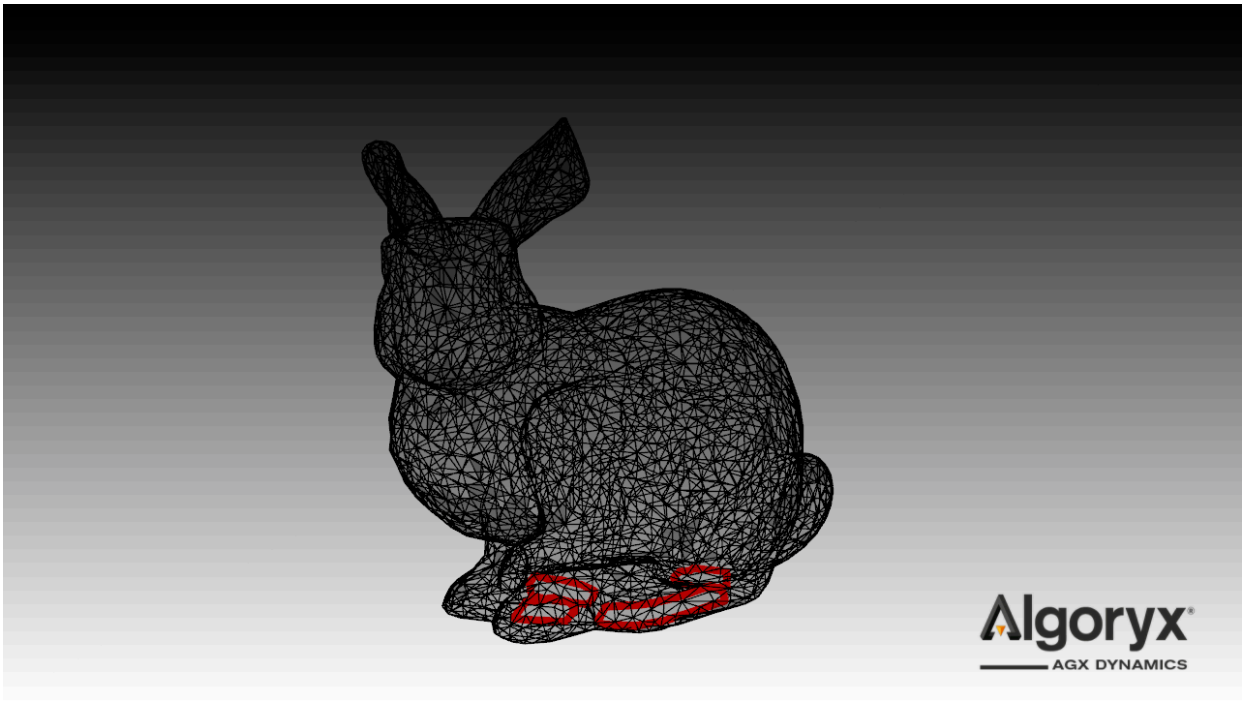


Fig. 11.4 Example visualization of a loaded trimesh where there are some regions which do not have neighbor half-edges.

11.8.9.4. Mesh reduction

The time for computing geometry contacts when using a triangle mesh model can be related to the complexity of the triangle mesh. I.e. the number of triangles.

`agxUtil::reduceMesh` is a function that can reduce the complexity of a triangle mesh by algorithmically reduce the number of edges in a process called collapsing. The goal of the mesh reduction is to reduce the number of triangles but at the same time keep the overall shape of the mesh.

The function operates on arrays of vertices and indices to be as general as possible. Below is a code snippet that demonstrates the functionality of the mesh reduction utility. The arguments are the set of vertices and indices that should be reduced, the target reduction (in percent, 1==no reduction, 0.5==50% reduction) and an aggressiveness (>0). The higher value, the more aggressive the algorithm will be, and the longer time it will take.

```
agxIO::MeshReaderRef reader = new agxIO::MeshReader();
reader->readFile("data/models/bunny.obj");


agx::Vec3Vector out_vertices;
agx::UInt32Vector out_indices;
```

```
double target_reduction = 0.5; // % triangles that should be left in the model after reduction
double aggressiveness = 7.0;    // A value that indicates how aggressive the reduction should be
agxUtil::reduceMesh(reader->getVertices(), reader->getIndices(),
                    out_vertices, out_indices,
                    target_reduction,
                    aggressiveness);

// Create a Trimesh with the reduced data
agxCollide::TrimeshRef mesh = new agxCollide::Trimesh(&out_vertices, &out_indices,
                                                      "Mesh",
                                                      agxCollide::Trimesh::REMOVE_DUPLICATE_VERTICES);
```

11.8.9.5. Triangle mesh traversal

If a trimesh is modeled in the above recommended way and thus has a half edge structure, it can be traversed via this neighborhood graph: each triangle consists of three half edges, where each half edge has a neighboring half edge in a bordering triangle, and where one half edge's starting vertex is the other one's ending vertex.

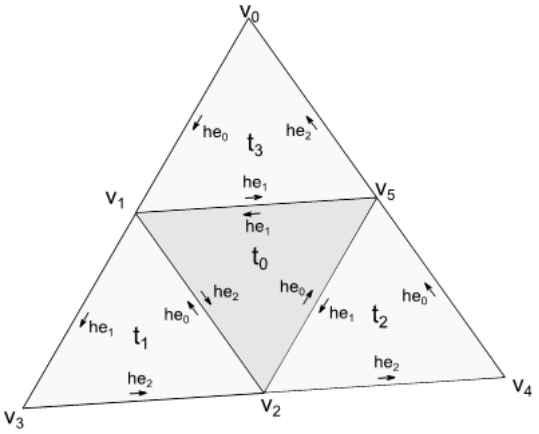


Fig. 11.5 A trimesh consisting of 4 triangles and 6 vertices. The half edge structure is shown.

This data is stored as arrays of vertices, triangle half edges and half edge neighbors in the Trimesh class and can be used directly.

Some helper classes make traversal of triangles easier, though. The neighborhood structure can be traversed by using the helper class **agxCollide::Mesh::Triangle**.

```
// Create the trimesh shape from a WaveFront OBJ file:
agxCollide::TrimeshRef trimesh =
      agxUtil::TrimeshReaderWriter::createTrimesh( "pin.obj" );
// Get first triangle in trimesh
agxCollide::Mesh::Triangle triangle = trimesh->getTriangle(0);
if (triangle.isValid()) // Important to check validity before use
      triangle = triangle.getHalfEdgePartner(0); // Get neighbor to first edge
```

Circling around a vertex can be done more comfortably using the helper class `agxCollide::Mesh::VertexEdgeCirculator`.

```
// Create the trimesh shape from a WaveFront OBJ file:
agxCollide::TrimeshRef trimesh =
      agxUtil::TrimeshReaderWriter::createTrimesh( "pin.obj" );
// Get a circulator around the first vertex in the first triangle.
agxCollide::Mesh::VertexEdgeCirculator circulator =
      trimesh->createVertexEdgeCirculator( 0, 0 );
// The circulator might be invalid if created from invalid parameters.
if (circulator.isValid()) {
  // Has the circulator done a whole turn around the vertex?
  while (!circulator.atEnd()) {
    // Set it to next position. The VertexEdgeIterator will even manage
    // holes in the neighborhood structure by jumping over all holes.
    circulator++;
    if (circulator.hasJustJumpedOverHole())//Have holes been passed?
      std::cout << "Vertex circulator jumped over hole\n";
```

```
  }
}
```

More elaborate examples can be found in tutorial_trimesh.cpp.

11.8.10. Convex

Convex is a geometry that describes a convex shape built up from a point cloud. The properties of a convex is among the following: none of the surface normals of a convex will ever cross each other, any point on a 2D projection of a convex will only be covered twice, once from the surface facing away the projection plane and once from the surface facing the projection plane.

A convex mesh can be created from a number of different sources.

1. From a triangle mesh that is known to define a convex mesh.
2. From a set of 3d points.
3. From 3d mesh files.
4. Using convex decomposition of triangle meshes.

If you have mesh data (vertices and indices) that you _know_ are convex, you can use the constructor to create a convex mesh:

```
agx::Vec3Vector vertices;
agx::UIntVector indices;
// We know this data to represent a convex mesh:
ConvexRef convex = new Convex(&vertices, &indices);
```

If you have a set of 3d points you can use the utility function in the `agxUtil::ConvexReaderWriter` namespace:

```
#include <agxUtil/ConvexReaderWriter/ConvexReaderWriter>
agx::Vec3Vector vertices;

for(size_t i=0; i < 30; i++)
{
  auto v = agx::Vec3::random( agx::Vec3(0, 0, 0), agx::Vec3(1, 1, 1) );
  vertices.push_back(v);
}

// Create the collision shape
agxCollide::ConvexRef convexMesh = agxUtil::ConvexReaderWriter::createConvex(vertices);
```

If you have a 3d mesh defined in a file:

```
#include <agxUtil/ConvexReaderWriter/ConvexReaderWriter>

// Create the collision shape
agxCollide::ConvexRef convexMesh = agxUtil::ConvexReaderWriter::createConvex("models/bunny.obj");
```

11.8.10.1. Convex decomposition



Fig. 11.6 A triangle mesh object decomposed into convex meshes with varying resolution.

The limitation of convex in that it can only describe a small subset of geometries (convex) can be overcome by taking a general (concave) triangle mesh and decompose it into convex meshes. The result will not be an exact representation of the original triangle mesh, but in many cases it will be good enough for use in simulations. AGX supports *convex decomposition* through the class `agxCollide::ConvexFactory` where you have full control over the parameterization of the algorithm.

```
#include <agxCollide/ConvexFactory.h>

// Create a ConvexFactory
agxCollide::ConvexFactoryRef factory = new agxCollide::ConvexFactory;

// Create a mesh reader for various mesh file formats.
agxIO::MeshReaderRef reader = new agxIO::MeshReader;
if (!reader->readFile( "mesh.obj"))
  error();

// Create a parameter object so we can fine tune the decomposition algorithm:
agxCollide::ConvexFactory::VHACDParametersRef parameters = new agxCollide::ConvexFactory::VHACDParameters();
parameters->resolution = 20;
parameters->alpha = 0.001;
parameters->beta = 0.01;
parameters->concavity = 0.01;
parameters->maxNumVerticesPerCH = 64

// Try to parse the data read from the reader and create convex shapes
size_t numConvexShapes = factory->build(reader, parameters);
if (numConvexShapes == 0)
  error();

// Get the vector of the created shapes
agxCollide::ConvexShapeVector& shapes = factory->getConvexShapes();

// Now insert shapes into one or more geometries.
```

An alternative is to use simpler utility function `agxUtil::ConvexReaderWriter::createConvexDecomposition`:

```
#include <agxUtil/ConvexReaderWriter/ConvexReaderWriter.h>

size_t resolution = 20;
agxCollide::ConvexShapeVector shapes
agxUtil::ConvexReaderWriter::createConvexDecomposition("mesh.obj", result, resolution);

// Now insert shapes into one or more geometries.
```

Below are the available parameters to the convex decomposition algorithm.

Table 11.2 Specifications of parameters to the convex decomposition algorithm

Parameter	Description	Range	Default value
resolution	maximum number of voxels generated during the voxelization stage. This will be recalculated as resolution ^3 before it reaches VHACD	[2, 400]	50
concavity	maximum concavity.	[0.0, 1.0]	0.0025
planeDownsampling	Controls the granularity of the search for the best clipping plane.	[1, 16]	4
convexhullDownsampling	Controls the precision of the convex-hull generation process during the clipping plane selection stage.	[1, 16]	4
alpha	Controls the bias toward clipping along symmetry planes.	[0.0, 1.0]	0.05
beta	Controls the bias toward clipping along revolution axes.	[0.0, 1.0]	0.05
pca	Enable/disable normalizing the mesh before applying the convex decomposition.	[false, true]	0
mode	0: voxel-based approximate convex decomposition, 1: tetrahedron-based approximate convex decomposition.	0/1	0
maxNumVerticesPerConvex	Controls the maximum number of triangles per convex hull.	[0, 1024]	64

Parameter	Description	Range	Default value
minVolumePerConvex	Controls the adaptive sampling of the generated convex hulls.	[0.0, 0.01]	0.0001
maxConvexHulls	Maximum number of convex hulls.	[1, 1024]	1024

11.8.11. Oriented bounding volumes

Oriented Bounding Volumes (OBV) (box, cylinder, capsule) can be computed based on a set of vertices. An optimal OBV should tightly enclose the vertices to best represent the more complex shape.

Below is a code snippet that will create an oriented Box shape that will fit the given vertices. The size and the transformation are stored in the supplied arguments to `agxUtil::computeOrientedBox`. The transformation will contain the translation and rotation of the box relative to the coordinate system of the vertices (mesh).

```
// Get the vertices from an existing mesh shape
const agx::Vec3Vector& vertices = mesh->getMeshData().getVertices();

// Here is where we will store the local transformation of the oriented box relative to the Mesh
agx::AffineMatrix4x4 transform;

// Here is where we will store the size of the computed oriented box
auto halfExtents = agx::Vec3();

// Compute the size and rotation of the oriented box given the vertices
agxVerify(agxUtil::computeOrientedBox(vertices, halfExtents, transform));

// Create a box primitive based on the computed size
auto box = new agxCollide::Box(halfExtents);

// Create a Geometry including the transform that will rotate/translate the oriented bounding box relative to the mesh
auto boxGeometry = new agxCollide::Geometry(box, transform);
```

Alternatively an oriented Cylinder or Capsule can be created using `agxUtil::computeOrientedCylinder` and `agxUtil::computeOrientedCapsule` There is an option on how the shape should be oriented using the enum `agxUtil::ShapeOrientation::Orientation`. Default is `agxUtil::ShapeOrientation::MINIMIZE_VOLUME` which will automatically choose the principal axis of the computed shape that results in the smallest volume.

Below is code that will create an oriented cylinder based on a set of vertices.

```
// Get the vertices from an existing mesh shape
const agx::Vec3Vector& vertices = mesh->getMeshData()->getVertices();

// Here is where we will store the local transformation of the oriented box relative to the Mesh
agx::AffineMatrix4x4 transform;

// Here is where we will store the radius and height of the computed cylinder
auto radiusHeight = agx::Vec2();

auto orientation = agxUtil::ShapeOrientation::MINIMIZE_VOLUME; // Default

// Compute the size and rotation of the oriented box given the vertices
agxVerify(agxUtil::computeOrientedCylinder(vertices, radiusHeight, transform, orientation));

// Create a cylinder primitive based on the computed size
auto cylinder = new agxCollide::Cylinder(radiusHeight[0], radiusHeight[1]);

// Create a Geometry including the transform that will rotate/translate the oriented bounding cylinder relative to the
mesh
auto boxGeometry = new agxCollide::Geometry(cylinder, transform);
```

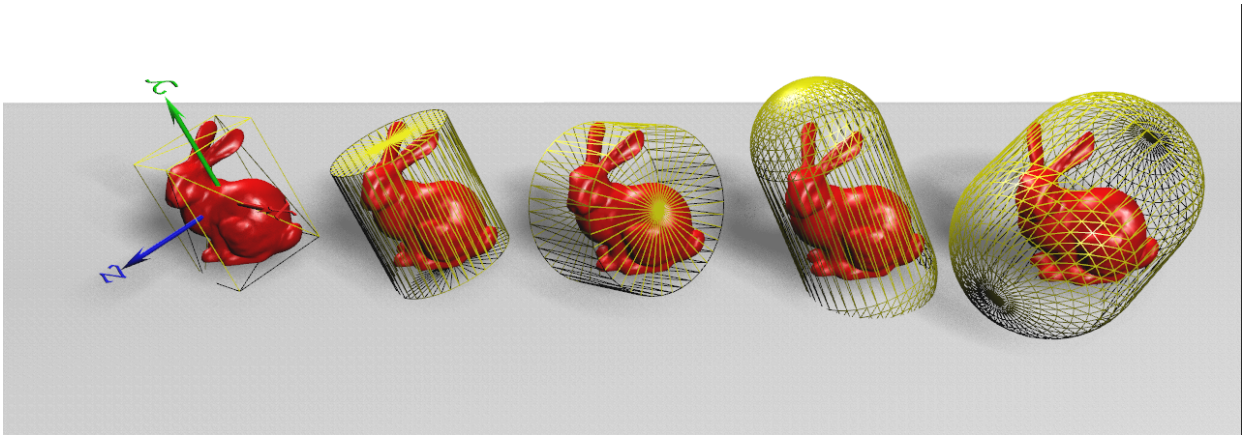


Fig. 11.7 This figure show a number of alternative bounding volumes and selection of principal axis for the computed shape.

11.8.12. WireShape

The wire shape is used internally for the collision handling of wire segments. It is based on the capsule, but might in the future have support for continuous collision detection. It is not advised to use this shape for other purposes, since implementation details might change.

11.8.13. HeightField

A height field is a geometric representation that can be used for terrain, sea beds etc. Height-fields are currently implemented as a scaled uniform grid centered at the local origin in x and y, and with z up. Furthermore, height-fields are treated like triangle meshes, where each rectangular grid cell is divided into a lower left and an upper right triangle. The corresponding class `agxCollide::HeightField` shares a common base class called `agxCollide::Mesh` with another mesh-based collision shape called `agxCollide::Trimesh`.

Compared to a trimesh, a height-field for collision detection as a terrain has the advantage of being able to modify the terrain under run-time. Trimeshes as terrains, on the other hand, can be modeled with fewer triangles by adapting the level of detail to the local change in terrain height, so that large plain parts can be covered by few large triangles.

Since trimeshes and height-fields share many routines for mesh traversal, please look into the trimesh chapter (11.8.9) for more information about that.

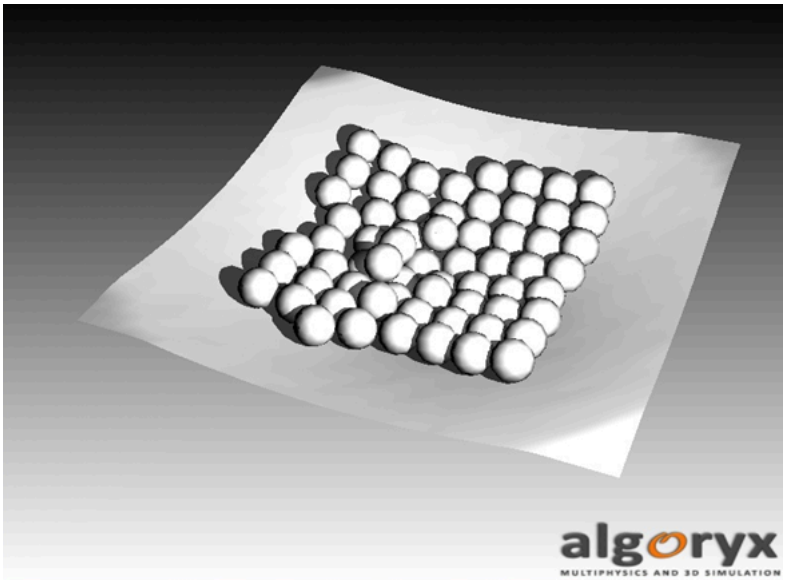


Fig. 11.8 HeightField representing uneven surface.

Like trimeshes used as terrains, height-fields make use of a bottom margin in order to have a safeguard in case objects tunnel downwards in local z-direction through the triangle surface.

11.8.13.1. HeightField creation

The height values in a height field can either be specified through a call to `agxCollide::HeightField::setHeights()` which accepts a vector in row matrix order or via a `agxUtil::HeightFieldGenerator()`, which can read data from an image (.png) or a raw binary file (no endian handling supported).

```
HeightField(size_t resolutionX, size_t resolutionY, agx::Real sizeX, agx::Real sizeY, agx::Real
bottomMargin=agx::Real(1));
```

11.8.13.2. Modifying parts of a HeightField

The height values in a height field can also be changed in only a part of the whole shape. One possibility is by calling a version of **agxCollide::HeightField::setHeights()** which specifies which rectangular part of the grid to modify.

```
bool setHeights(const agx::RealVector& heights,
      size_t minVertexX, size_t maxVertexX, size_t minVertexY, size_t maxVertexY);
```

Another possibility is to set heights at specific points in the grid using the function **agxCollide::HeightField::setHeight()**.

```
void setHeight(size_t xIndex, size_t yIndex, agx::Real height);
```

Note

All triangles around a changed height have to be updated. This update includes a recalculation of the triangle normal, as well as an update of the bounding volume hierarchy. Changing a single height will involve 6 updated triangles. Changing several heights within a rectangle can be done faster by using `agxCollide::HeightField::setHeights()` rather than `agxCollide::HeightField::setHeight()`, since this makes it unnecessary to update triangles several times.

11.8.13.3. HeightFields for dynamic simulation

There can be cases where it is of interest to combine the ease of modification of a height field with dynamical simulation, by making the height field part of the collision detection representation of an **agx::RigidBody** which is simulated dynamically. Since height-fields are generally assumed to be part of the static scenery, their mass properties are not calculated in the general case, since this comes with some computational cost. In order to get the mass-properties updated for dynamic simulation, call the method `agxCollide::HeightField::setDynamic()`.

In order to compute these, a sort of box-shape is assumed for the height-field, where the top has a relief defined by the heights, the sides go downwards at the borders of the grid, and the bottom is a rectangle parallel to the z plane at a position given by a specified minimum height. To specify this height, use **agxCollide::HeightField::setMinAllowedHeight()**.

Note

When changing parts of the height-field, the mass properties will not be updated. Instead, once all the updates are done and the new mass properties are needed for dynamic simulation, one should call `agxCollide::HeightField::calculateMassProperties()` which will recalculate the mass properties for the whole height-field and can potentially be costly. This might be optimized in a future release.

11.8.14. Mesh reader

The class `agxIO::MeshReader` is capable of reading a number of 3D Mesh formats into mesh data that can be used to create a Trimesh or Convex shapes for collision detection. The enum `agxIO::MeshReader::FileExtension` lists all available formats. Most of the readers (with an exception to .shl) are utilizing the (Open Asset Importer Library) library for parsing the mesh files.

Note

Currently the MeshReader will concatenate all meshes into one mesh.

Table 11.3 Supported 3D Mesh model formats

File extension	Description
obj	Wavefront Obj
shl	Shell data format
fbx	Filmbox AutoDesk
assimp	AssImp reader/writer
dae	Collada
blend	Blender 3D
3ds	3ds Max 3DS
ase	3ds Max ASE
gltfb	Binary GL Transmission format
gltf	Ascii GL Transmission format
xgl	XGL
zgl	ZGL
ply	Stanford Polygon Library

File extension	Description
dxf	AutoCAD DXF
lwo	LightWave
lws	LightWave Scene
lxo	Modo
stl	Stereolithography
x	DirectX X
ac	AC3D
ms3d	Milkshape 3D
scn	TrueSpace
xml	Ogre XML
irrmesh	Irrlicht Mesh
irr	Irrlicht Scene
mdl	Quake I
md2	Quake II
md3	Quake III Mesh
pk3	Quake III Map/BSP
md5	Doom 3
smd	Valve Model
m3	Starcraft II M3
unreal3d	Unreal
q3d	Quick3D
off	Object File Format
ter	Terragen Terrain
ifc	Industry Foundation Classes (IFC/Step)
ogex	Open Game Engine Exchange(.ogex)

11.8.15. Composite Shapes

Composite geometries are geometries containing more than one Shape. This can be useful when building more complex geometries from primitives.

There are basically two ways of creating a more complex Geometry structure around a rigid body.

1. Create a geometry *g1*, add shapes with relative transformations that builds up the final geometry. This also means that *g1* can have only one material. (Material is bound per Geometry). Then, create a body b1 and add the created composite geometry *g1*.
2. Create a body *b1*, create the geometries needed *g1*, *g2*, … and for each geometry add shapes (Box, Sphere, …). Then, add each geometry to the body. In this case each geometry *g1*, *g2*… can each have different materials.

Note

There are two limitations when using composite shapes:

- If the single shapes forming the composite shape do not overlap, interacting objects might get in between them exactly where they touch. This might happen since the single shapes which the composite shape consists of do not know about each other. Therefore it is recommendable to let shapes overlap.
- However, when having overlapping shapes, the automatic mass property computation will count the overlapping volume twice. So here, manual mass property computation has to be used.

11.9. Shape colliders

A shape collider is a class that is responsible for calculating contact data between two shapes. The table below shows the available shapes and for which pair of shapes the AGX collision engine currently can calculate contact data.

11.9.1. Collider Matrix

In the following table, an overview over the functionality of colliders for different shape combinations is given. Reduced functionality means that basic functionality is given, but might have to be improved in the future in order to work perfectly in all purposes. Plane and HeightField make only sense as static geometries and can therefore not collide with themselves/each other.

Table 11.4 Available primitive tests

	Box	Capsule	Convex	Cylinder	Line	Plane	Sphere	Trimesh	Composite	HeightField	Cone	Ho
Box	X	X	X	X	X	X	X	X	X	X	(×)	(×)
Capsule	X	X	X	X	X	X	X	X	X	X	(×)	(×)
Convex	X	X	X	X	X	X	X	X	X	X	(×)	(×)
Cylinder	X	X	X	X	X	X	X	X	X	X	X	X
Line	X	X	X	X	X	X	X	X	X	X	X	(×)
Plane	X	X	X	X	X	—	X	X	X	—		
Sphere	X	X	X	X	X	X	X	X	X	X	(×)	(×)
Trimesh	X	X	X	X	X	X	X	X	X	X		
Composite	X	X	X	X	X	X	X	X	X	X	(×)	(×)
HeightField	X	X	X	X	X	—	X	X	X	—		
Cone	(×)	(×)	(×)	X	X		(×)		(×)		X	X
HollowCone	(×)	(×)	(×)	X	(×)		(×)		(×)		X	X
HollowCylinder	[×]	[×]	(×)	X	(×)	[×]	[×]	[×]	(×)	[×]	X	X

The symbols in table 11.4 indicate: × Full functionality, (×) Reduced functionality, [x] using cylinder collider — Not needed (since both are supposed to be static).

11.9.2. Point, normal and depth calculation

In case of an intersection between two shapes, a shape collider will report one or more contact points. Each contact point consists of

- A point *p*,
- a normal *n*,
- a depth *d*.

The normal *n* points in the direction which lets shape1 move out of shape2.

The depth *d* measures how much shape 1 would have to be moved out of shape2 along normal *n* in order to resolve the overlap and thus have the two shapes just touching.

The point p is placed halfway between the two points corresponding to the surfaces of the two shapes, so that $p_{shape_1} = p - n * d * 0.5$ and $p_{shape_2} = p + n * d * 0.5$.

11.9.3. Results for contacts with Line Shapes

The Line Shape is special in that its use is not in rigid body simulation, but in other fields as e.g. mouse picking or depth computation for unknown geometries. Therefore, the computation of contact point, normal and depth differs from that of the other Shapes.

The Line Shape represents mathematically a line segment, defined by two points P_0 and P_1, which define the non-normalized Line direction $d = P_1 - P_0$.

The Line can thus be presented as: $P_0 + t\ d, 0 \leq t \leq 1$.

If a Line intersects a Shape, the first point in contact P_c along the infinite version of the line is reported as contact point. The Shape surface normal is reported as contact normal. The parameter t_c is the value which realizes $P_c = P_0 + t_c\, d$.

The depth is $(1 - t_c) * | P_0 - P_1 |$; this corresponds to how long the Line has to be pushed back in its direction in order to just touch the overlap point P_c.

Exceptions:

- For mesh-based shapes (Trimesh, HeightField and at the moment also Convex), lines which are completely inside the shape will not create any contact.

- The line-line collider will create one contact point with a normal which is orthogonal to both lines and a depth of 0.

11.10. Storing renderable data in shapes: agxCollide::RenderData

`agxCollide::RenderData` is a class which can be used for storing renderable data together with a shape.

Note

The vertex/normal data should be stored in the local coordinate system of the Shape.

```
// Create a RenderData object
agxCollide::RenderDataRef renderData = new agxCollide::RenderData;

// Specify the vertices
agx::Vec3Vector vertices;
renderData->setVertexArray( vertices );

// Specify the normals
agx::Vec3Vector normals;
renderData->setNormalArray( normals );

// Specify the indices, which constructs Triangles
agx::UInt32Vector indices;
renderData->setIndexArray( indices, agxCollide::RenderData::TRIANGLES);

// Store a diffuse color
renderData->setDiffuseColor( agx::Vec4(1,0,1,1) );
```

This data can then be associated with an agxCollide::Shape:

```
gxCollide::ShapeRef shape = new agxCollide::Sphere(1.0);
shape->setRenderData( renderData );
```

During serialization to disk, this data will also be stored together with a shape. Later it can be retrieved with the call:

```
agxCollide::RenderData *renderData = shape->getRenderData();
```

In the sample coupling to OpenSceneGraph, agxOSG namespace, the function agxOSG::createVisual(geometry, agxOSG::Group *) will parse this render data and create a visual representation based on the RenderData (if available), otherwise it will use the specified shape when the renderable data is created.

11.11. Mass Properties

`agx::MassProperty` is a class which stores and handles the mass properties of a rigid body namely the heavy and effective mass of a rigid body. Whenever Geometries are added/removed to/from a rigid body with the methods

```
RigidBody::add(Geometry*)
RigidBody::remove(Geometry*),
```

the inertia tensor is recalculated by default, unless a previous call to `RigidBody::getMassProperties()::setAutoGenerate(false)` has been done. In this case the inertia tensor and masses will NOT be updated when geometries are added and removed.

Attention

Currently, automatic mass property calculation (incremental or not) assumes all geometries and shapes to be disjoint (non-overlapping). If they overlap, the effect of this assumption is that the overlapping volume will be counted twice into mass and inertia computation. It is recommended to set the mass properties manually in this case.

An explicit inertia tensor can also be specified manually with the call:

```
body->getMassProperties()->setInertiaTensor(I);
```

where I can either be a SPDMatrix3x3 (semi positive definite) matrix or a Vec3 specifying the diagonal elements of the inertia tensor.

The method `MassProperties::setInertiaTensor` has a second parameter specifying whether the inertia tensor should be automatically recalculated, when set to false, the Inertia tensor for a rigid body will not be updated to reflect changes in associated Geometries.

During the calculation of mass properties, the density from the specified materials (described later) will be used to calculate the final mass. If no mass is specified explicitly for the body, the calculated mass will be used. If on the other hand the mass have been set by the user with the method `RigidBody::getMassProperties()::setMass()`, this mass will be used for the final result.

Unless the setAutoGenerate method has been called for a rigid body, the center of mass and inertia tensor will be updated for most operations such as adding/removing geometries, removing shapes from geometries. However there are a number of operations that will NOT be detected, and hence the user has to tell the body to update its mass properties:

- Transforming geometries relative the body after they are added to the body.
- Changing material properties (density) for any geometry added to a body.
- Transforming shapes that is part of a geometry associated to a body.

In the above cases the mass properties can be updated with a call to:

```
agx::RigidBody::updateMassProperties(agx::MassProperties::AUTO_GENERATE_ALL);
```

If you only want to update the inertia tensor you can change the argument to updateMassProperties. For more information see the SDK documentation.

11.12. Adding user forces

The class `agx::RigidBody` has the following methods for adding forces and torques:

addForce(), addForceAtPosition(), addForceAtLocalPosition(), addTorque().

These added forces are cleared after a time step has been taken in the simulation. The force added to a rigid body during the last timestep (including the forces introduced by the user via addForce/addForceAtPosition/addTorque , or via force fields etc. – however, NOT the ones from contacts and constraints!) can be accessed with the methods: `RigidBody::getLastForce()` and `RigidBody::getLastTorque()`

11.13. ForceFields

The class `agx::ForceField` can be used to induce user forces on all or a selected part of the bodies in a DynamicsSystem. It is derived from `agx::Interaction`.

```
#include <agx/ForceField.h>
class MyForceField : public agx::ForceField
{
  public:
  MyForceField() {}
  ~MyForceField() {}

  // Implement this method to add forces/torques of all or selected bodies
  void updateForce( DynamicsSystem *system );
};

// Create the force field and add to the DynamicsSystem.
agx::InteractionRef forceField = new MyForceField;
simulation->getDynamicsSystem()->add( forceField );
```

For a more thorough example, see tutorial_bodies.cpp in <agx>/tutorials/agxOSG/tutorial_bodies.cpp

11.14. GravityField

Gravitational forces should be applied through an instance of an `agx::GravityField`. The reason for this is that certain mechanics inside for example the wire implementation (`agxWire::Wire`) needs to know the individual gravity for each body.

By default, an instance of a class `agx::UniformGravityField` is created in DynamicsSystem, with the default acceleration of G = 9.80665 m/s^2. Another gravitational model is `agx::PointGravityField`, which will add a force towards a specified origin point. This model can be used for simulating the gravity on the surface of the earth.

To register a gravity field with the simulation the following call is done:

```
agxSDK::SimulationRef sim = new agxSDK::Simulation;
sim->setGravityField( new agx::PointGravityField );
```

`Simulation::setUniformGravity` and `Simulation::getUniformGravity` are just wrapper methods to simplify the use of the standard uniform gravity. A call to `Simulation::setUniformGravity()` will return false if the current gravity field model is not a `agx::UniformGravityField`.

11.14.1. CustomGravityField

The standard gravity fields (UniformGravityField and PointGravityField) are implemented as computational kernels to achieve best possible performance. For certain situations, for example when simulating micro-gravity on a asteroid, a custom gravity field might be required.

To implement a custom gravity field, a class that derived from `agx::CustomGravityField` is created. The actual code for calculating the gravity is implemented in the virtual method `calculateGravity`. The overhead compared to the standard gravity fields is the call to virtual method which should be small.

It is however important to remember that the method `calculateGravity()` will be called simultaneously from several threads if AGX is using more than one thread. This means that the method must be made thread safe.

```
class MyGravityField : public agx::CustomGravityField
{
public:
  MyGravityField() {}
  /// overidden method to calculate the gravity for a point in 3D Space
  virtual agx::Vec3 calculateGravity(const agx::Vec3& position) const
  {
    return position*-1;
  }
protected:
  ~MyGravityField() {}

};

// Register the gravity field with the simulation
simulation->setGravityField( new MyGravityField() );
```

11.15. Material

The material of a geometry will define how the geometry will get affected and affect the rest of the system. A material contains definitions of friction, restitution, density and other parameters that describe the physical surface, bulk and line attributes of a material. A material can be associated to a geometry with the method:

```
agxCollide::Geometry::setMaterial(agx::Material*)
```

An important thing to remember is that the name of a material has to be unique. Adding two materials with the same name will result in that only the first will be used in the system.

New materials must be registered with the `agxSDK::MaterialManager` before they can be used in the simulation (which is done when the material is added to the simulation).

When two geometries collide, the resulting contact parameters from the two involved materials are derived into an `agx::ContactMaterial`. The default contact material function can be overridden by adding explicit contact materials to the material manager.

Materials consist of three sub-classes which can be accessed through the calls:

```
material->getSurfaceMaterial();
material->getBulkMaterial();
material->geWireMaterial();
```

To complement the Material API, AGX also contains a Material Library with some predefined material properties that can be loaded and used as a starting point when designing simulations. See further in the Material Library section.

11.15.1. SurfaceMaterial

This class defines the behavior/surface attributes for contacts between geometries using two materials. The attributes currently available are:

- Roughness – Corresponds to friction coefficient, the higher value, the higher friction.
- Adhesion – determines a force used for keeping colliding objects together.
- Viscosity – Defines the compliance for friction constraints. It defines how “wet” a surface is.
- ContactReduction - Specifies whether contact reduction should be enabled for this material.

11.15.1.1. Roughness

The roughness describe what *potential* friction this material would have when it interacts with another material. The exact friction coefficient is calculated for the implicit contact material using the following equation: $friction = \sqrt{(roughness_1 \times roughness_2)}$

11.15.1.2. Adhesion

Adhesion defines an attractive force that can simulate stickiness. It operates only on bodies that have colliding geometries, i.e. it is not a general force field.

Adhesion is set using two parameters:

```
SurfaceMaterial::setAdhesion( adhesionForce, adhesiveOverlap );
```

adhesionForce specifies a force >=0, that is an upper bound of the attractive force acting between two colliding rigid bodies in the normal direction.

adhesiveOverlap is an absolute distance that defines the contact overlap where the adhesive force is active. Because that the adhesive force only acts upon overlapping rigid bodies, it is important to specify an overlap > 0. Otherwise a separating force would immediately (within a timestep) cause two bodies to separate, hence no adhesive force would then act on the bodies. The adhesiveOverlap is a safe zone to ensure that there will be time for the adhesiveForce to act.

Because the adhesive force operates only in the normal direction of the contact, it is also important to increase the roughness/friction to get a sticky behavior also in the friction direction.

$max_adhesive_force \approx adhesiveOverlap \,/\, contact_compliance$

The adhesion distance will change the depth of the resting contact as seen in Fig. 11.9:.

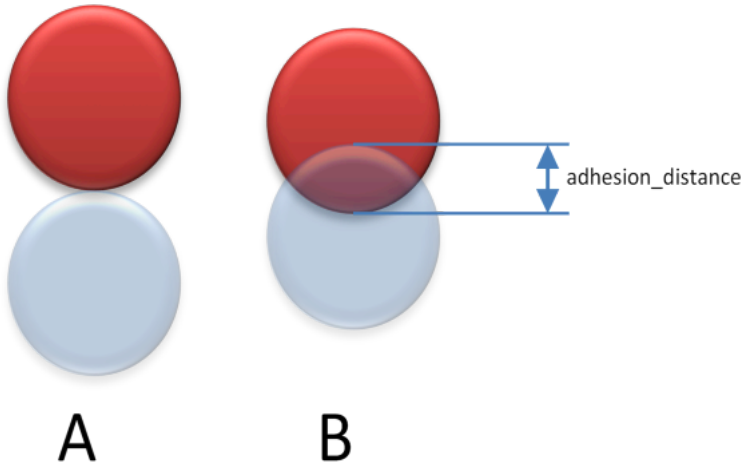


Fig. 11.9 A: no adhesion. B: adhesion_distance=r/2

The resulting adhesion parameters of an implicit contact material is calculated as $(adhesion_1 + adhesion_2, min(adhesiveOverlap_1, adhesiveOverlap_2)$

11.15.1.3. Surface viscosity

The viscosity of a surface material is the same thing as compliance but for friction in contacts. The resulting viscosity of an implicit contact material is calculated as $viscosity = viscosity_1 + viscosity_2$

11.15.1.4. Contact Reduction

This parameter specifies whether contact reduction should be enabled for this material. If two geometries overlap, the resulting contact material will get contact reduction enabled only if both materials have contact reduction enabled (logical and).

11.15.2. BulkMaterial

This class defines the density and other internal attributes of a simulated object. Lines use for example YoungsModulus and PoissonsRatio to calculate bending stiffness etc.

Attributes are:

- *Density*
- *YoungsModulus* – Stiffness of the material. Used when calculating contacts and the bending behavior of lines.
- *Viscosity* – Used for calculating restitution.
- *Damping* – Used for controlling how much time it should take to restore the collision penetration. A larger value will result in “softer” contacts.

11.15.2.1. Density

The density is used for calculating the mass of the rigid body from the volume of the geometry.

11.15.2.2. YoungsModulus

This specifies the *stiffness* of the contact between two interacting geometries/bodies. The resulting stiffness/young's modulus for an implicit contact material is calculated as $(Y_1 \times Y_2) \div (Y_1 + Y_2)$

11.15.2.3. Bulk viscosity

This defines the "bounciness" of an object or *restitution* in the contact material. The resulting restitution of an implicit contact material is calculated as $restitution = \sqrt{((1 - viscosity_1) \times (1 - viscosity_2))}$

11.15.2.4. Damping Time

This defines the time it should take for the solver to restore an overlap. A higher value will lead to smaller restoration forces since the overlap should be restored slower. The resulting damping time of an implicit contact material is calculated as $damping = max(damping_1, damping_2)$

11.15.3. WireMaterial

Defines the behavior when a geometry with this material collides with a wire. Determines the stick and slide friction. When a wire is colliding with a geometry (where the wire is overlapping an edge of a box/trimesh, or onto a cylinder), a ShapeContactNode is created. This node is allowed to slide along the edge and also in the direction along the wire. To solve for the normal force, a 1D solver will analyze the system and solve how the contact node is allowed to move, depending on masses, gravity, physical properties in the wire and the tension in the wire.

Damping and *Young's Modulus* can be specified for two attributes in the wire material:

- *Bend* – Constraint which tries to straighten a wire.
- *Stretch* - Constraint which works against stretching the wire.

For more information, see the SDK documentation on Material.

11.15.4. ContactMaterial

The class `agx::ContactMaterial` is used by the system to derive the actual material attributes when two materials collide. It can also be used to explicitly define a material used at contact calculations. All attributes found in the surface, bulk and line material is represented in the ContactMaterial, which is later used by the solver.

AGX use by default a contact point based method for calculating the corresponding response between two overlapping geometries.

There is also a different method available which is named *area contact approach*. This method will try to calculate the spanning area between the overlapping contact point. This can result in a better approximation of the actual overlapping volume and the stiffness in the response (contact constraint).

In general, this will lead to slightly less stiff, more realistic contacts and therefore the Young's modulus value usually has to be increased to get a similar simulation results as running the simulation without the contact area approach.

```
// Create a contact material based on two agx::Material's m1 and m2
agx::ContactMaterial *cm = simulation->getOrCreateContactMaterial(m1, m2);

// When two geometries collide, where m1 and m2 is involved, the contact area approach
// will be used.
cm->setUseContactAreaApproach( true );
```

```
agx::Real minElasticRestLength=0.0005, agx::Real maxElasticRestLength=0.05;
cm->setMinMaxElasticRestLength(minElasticRestLength, maxElasticRestLength);
```

The above code line determines the contact depth-span in which the material is elastic. AGX calculates this distance, but the material settings will put limits on this value. It can not be less than the minimum material rest length or greater than the maximum material rest length. For this reason, the minimum material rest length can not be greater than the maximum material rest length. For the theory background, see Contact mechanics (Elastic Foundation Model).

11.15.5. Overriding ContactMaterial friction and restitution

A GeometryContact contains collision information between two geometries. One ContactMaterial is assigned to each GeometryContact. Hence all contact points in the GeometryContact would use the same friction and restitution.

To support different values per contact point, the ContactMaterial values for friction and restitution are used to initialize the corresponding values for each ContactPoint in the GeometryContact.

These values can then be modified as needed on a per-point basis in a user callback, for example via a *ContactEventListener* or via the *preCallback* in a *StepEventListener*. These events are after Space have performed collision detection, but before the solver used these values.

11.16. Calculating contact friction

At present, there is no difference between static and kinetic friction coefficients. According to published data and common models, these two coefficients differ by 10-20%. This is relevant in simulation of gear dynamics but not in most other cases. Friction coefficients do depend on the two materials in contacts.

So given material1 with roughness 0.5 and material2 with roughness 0.2, what friction coefficient should be used when these two materials get in contact? There is no clear picture of how to calculate this resulting friction coefficient in the general case. Therefore, by default, the resulting friction when material1 and material2 collides will be the *geometric averaged* value:

```
sqrt(material1.roughness*material2.roughness) = 0.31623
```

This is of course a crude approximation which does not have any relation to the real world physics. To overcome this, AGX supports *explicit ContactMaterial*, see Section 11.16.2.

Contacts are handled in two separate stages, namely, *impact*, and *resting contact*. Impacts correspond to a rigid body which is found in geometric interference at the beginning of a step with sufficiently large incident velocity, i.e., when the relative velocity projected on the separating normal is sufficiently negative. During impact resolution, the impacting energy is *approximately* restituted to the level of the restitution coefficient. This is done by computing an impulsive stage which preserves all constraints but applies a Newton restitution model on impacting contacts, with post impact velocity a fraction of the pre-impact incident velocity. That fraction is the restitution coefficient and it is usually a number between 0 and 1. The restitution coefficient applies to all pairs of rigid bodies and is part of the pair-wise contact properties. Because we do not locate the exact time of impact, there can be a *small* decrease in energy even if restitution is exactly 1.

After impact, constraint *regularization* and *stabilization* come into play. From the modeling perspective, this is no more and no less than a spring/damper system. Numerically, the spring and damper system is unconditionally for nonzero damping. The two parameters are accessible as *contact compliance* (1/YoungsModulus) and *damping coefficient* in the material properties. Compliance is just the inverse of the desired spring constant, with the usual units. The AGX damping parameter (also called “Spook damping parameter”) has units of time for technical reasons. Use the functions in `agxUtil::Convert` in order to convert from viscous damping coefficient to AGX’ damping coefficient.

There are basically two ways of setting up the material table for a complete system: defining *implicit* or *explicit* material properties.

11.16.1. Defining implicit Material properties

By creating an `agx::Material`, setting the properties, assigning it to a geometry and adding it to the Simulation, we have told the system that this geometry is created from some kind of material. When this geometry collides with another geometry, the pair of materials are used to calculate the resulting `agx::ContactMaterial`. This is a fairly easy way of creating the materials in the system, as every pair of materials does not really have to be known in beforehand. The calculation of material properties for the implicit contact material cm between material m1 and m2 is done as follows:

- cm.surfaceFrictionCoefficient = sqrt(m1.roughness*m2.roughness)
- cm.restitution = sqrt((1-m1.viscosity) * (1-m2.viscosity))
- cm.adhesion = m1.adhesion+m2.adhesion
- cm.youngsModulus = m1.youngsModulus * m2.youngsModulus / (m1.youngsModulus + m2.youngsModulus)
- cm.damping = max(m1.damping, m2.damping)
- cm.surfaceViscosity = m1.surfaceViscosity+m2.surfaceViscosity

The frictional parameters for wire friction are using the same geometric medium as surface friction.

Note

We recommend that *explicit* contact material are used. This will give a better control over the simulation result instead of relying on the automatically calculated values in an implicit contact material.

11.16.2. Defining explicit ContactMaterial properties

If the actual values used in the friction solver has to be known exactly, the above method is not appropriate. Instead an explicit *material table* should be created.

This can be done by associating two `agx::Material`, M1 and M2 to *one user created* `agx::ContactMaterial` *CM*. So whenever the two materials *M1* and *M2* are involved in a contact, the attributes of *CM* are used. However, the following attributes of M1 and M2 still have to be set, to correctly calculate the mass properties of a RigidBody and to have correct behavior for LineConstraints.

```
BulkMaterial::setDensity()
BulkMaterial::setYoungsModulus()
```

At contact between two instances of `agx::Material` the following list of attributes are used and should therefore be set to the required value:

```
ContactMaterial::setRestitution()
ContactMaterial::setFrictionCoefficient()
ContactMaterial::setDamping()
ContactMaterial::setEnableContactReduction()
```

FRICTION	PLASTIC	METAL	WOOD
Plastic	??	0.7	0.3
Metal	—	??	0.2
Wood	—	—	??

Table showing friction coefficients between different materials, defined via explicit or implicit contact materials.

To build a table of explicit materials each ContactMaterial are associated to two instances of `agx::Material`. Code for setting up this table could look like:

```
agx::MaterialRef wood = new agx::Material("Wood");
agx::MaterialRef plastic = new agx::Material("Plastic");
agx::MaterialRef metal = new agx::Material("Metal");

// Define the material for plastic/metal
agx::ContactMaterialRef plastic_metal = new agx::ContactMaterial(plastic, metal);
plastic_metal->setRestitution( 0.6 );
plastic_metal->setFrictionCoefficient( 0.7 );
plastic_metal->setYoungsModulus( 0.1E9 ); // GPa
// Define the material for wood/metal
agx::ContactMaterialRef wood_metal = new agx::ContactMaterial(wood, metal);
wood_metal->setRestitution( 0.2 );
wood_metal->setFrictionCoefficient( 0.2 );
```

```
// Define the material for wood/plastic
agx::ContactMaterialRef wood_plastic = new agx::ContactMaterial(wood, plastic);
wood_plastic->setRestitution( 0.4 );
wood_plastic->setFrictionCoefficient( 0.3 );
wood_plastic->setYoungsModulus( 3.654E9 ); // GPa

// Add ALL materials to the simulation
simulation->add(wood)
simulation->add(metal)
simulation->add(plastic)
simulation->add(plastic_metal);
simulation->add(wood_metal);
simulation->add(wood_plastic);
```

The above code example still leaves to the system to automatically derive the ContactMaterial for wood/wood, plastic/plastic and metal/metal.

11.16.2.1. ContactReduction

This parameter controls whether contact reduction should operate on the contacts where this contact material is used. For details, see Section 38.7.

11.16.2.2. Contact Area approach

If set to “true”, an approximation to the contact area will be geometrically computed for each contact involving this contact material. For each contact, its area will then be evenly distributed between its contact points. The contact compliance will be scaled with the inverse of the area for each contact point.

Note

This is an experimental feature, which has been tested on contacts involving meshes and boxes, but with a cruder approximation for contacts involving spheres or capsules. Recommended for contact mechanics with a higher level of fidelity, such as grasping (involving meshes/boxes). If set to “false”, the unity area will be assumed (default).

```
// Define the material for wood/plastic
agx::ContactMaterialRef wood_plastic = new agx::ContactMaterial(wood, plastic);
```

```
wood_plastic->setUseContactAreaApproach(true);
```

11.16.3. Friction model

There are several friction model implementations in AGX Dynamics, each with different characteristics. They are commonly defined around the *primary* ($\hat{u}$) and *secondary* ($\hat{v}$) friction directions. $\hat{u}$ is orthogonal to $\hat{v}$ and both are orthogonal to the contact normal $\hat{n}$. The plane spanned by $\hat{u}$ and $\hat{v}$ is called the *friction plane* and the friction forces are applied in this plane at the contact point $\bar{p}$.

Finding an accurate friction force is nonlinear, i.e., the friction force depends on the normal force and the normal force depends on the friction force. This nonlinearity affects how quick a solver will find a solution, and how accurate the results are. The different friction models and choice of solve type enables the configuration possibility to have simulations with (relatively) computationally expensive, precise, to computationally inexpensive, acceptable, results.

In AGX Dynamics, a nonlinear frictional contact is solved for the three force directions along $\hat{u}$, $\hat{v}$ and $\hat{n}$ such that,

$$\begin{aligned} |F_n| &\geq 0 \\ |F_u| &\leq \mu_u^{'} \mid F_n| \\ |F_v| &\leq \mu_v^{'} \mid F_n| \\ F_f &= F_u + F_v \end{aligned}$$

where F_n is the normal force, F_u and F_v the primary and secondary friction forces and F_f the resulting friction force. This formulation is consistent with the ideal **Coulomb friction law**;$F_f \leq \mu \mid F_n|$, when $\hat{u}$, $\hat{v}$, $\mu_u^{'}$ and $\mu_v^{'}$ are appropriately selected. AGX Dynamics different friction models and solvers supports this computation and various approximations that are more easily solved.

Even though the values of $\mu_u^{'}$ and $\mu_v^{'}$ depends on the implementation of the friction model, consider them as friction coefficients constrained by the given primary and secondary friction coefficients μ_u and μ_v such that,

$$\begin{aligned} 0 \leq \mu_u^{'} \leq \mu_u \\ 0 \leq \mu_v^{'} \leq \mu_v \end{aligned}$$

where μ_u and μ_v can be assigned to explicit contact materials:

```
agx::ContactMaterialRef cm1 = new agx::ContactMaterial(m1, m2);
// Both primary and secondary friction coefficients are assigned
// the given value - isotropic friction.
cm1->setFrictionCoefficient(0.4);

// Anisotropic friction, where primary != secondary friction coefficient.
agx::ContactMaterialRef cm2 = new agx::ContactMaterial(m1, m3)
cm2->setFrictionCoefficient(0.4, agx::ContactMaterial::PRIMARY_DIRECTION)
cm2->setFrictionCoefficient(0.2, agx::ContactMaterial::SECONDARY_DIRECTION)
```

Note

The primary and secondary friction directions, $\hat{u}$ and $\hat{v}$, are by default aligned with worlds' principal axes. See Oriented friction models or Custom friction models for more information and examples how to configure the friction directions.

The ideal **Coulomb friction law** states that, given a friction coefficient μ, the resulting friction force is$|F_f \mid \leq \mu \mid F_n|$. This means that, given a known (solved for) normal force F_n, F_f will be constrained within a circle in the friction plane (segmented circle below). However, it's common in numerical physics (including some friction model implementations in AGX Dynamics) to approximate $\mu_u^{'} = \mu_u$ and $\mu_v^{'} = \mu_v$, which won't result in F_f to be within the circle, but rather within a "box"/rectangle in the friction plane (solid line below). That's why friction model names may include the name "box", such as BoxFrictionModel and ScaleBoxFrictionModel, while IterativeProjectedConeFriction can find an accurate F_f within the circle.

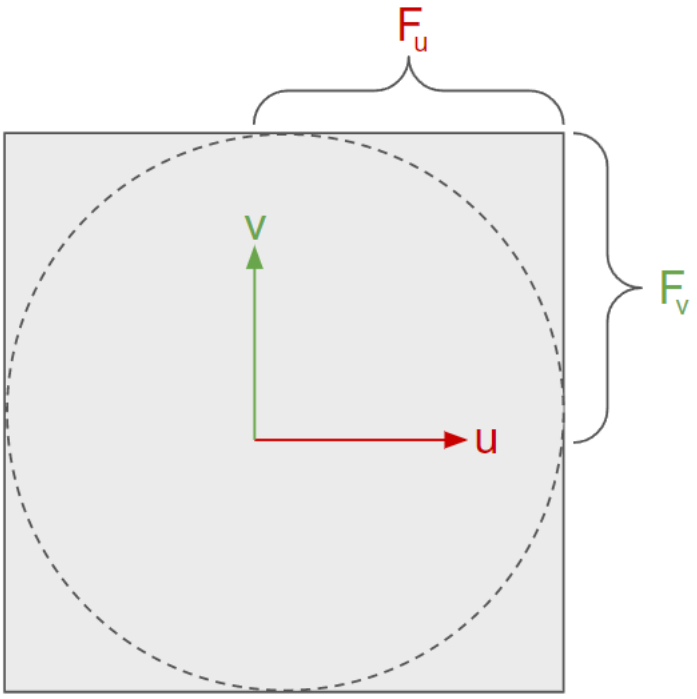


Fig. 11.10 Friction forces F_u, F_v and resulting friction force F_f is desired (for accuracy) to be within the segmented circle in the friction plane. "Box" friction models are an approximation resulting in the desired (ideal) circle to be a rectangle (solid) in the friction plane.

Worse case scenario for box friction models is when the resulting friction force F_f is at 45° (and $45^\circ + 90^\circ = 135^\circ$ etc.) in the friction plane. I.e., when $F_u = \mu_u F_n$ and $F_v = \mu_v F_n$, then

$$\begin{array}{ccc} |F_f| & = & |F_u + F_v| = \\ \sqrt{F_u^2 + F_v^2} & \leq & \sqrt{\mu_u^2 + \mu_v^2}\,|F_n| = \\ [\mu_u = \mu_v = \mu] & = & \sqrt{2}\mu\,|F_n| \end{array}$$

so F_f is overestimated by a factor of $\sqrt{2} \approx 1.41$ in the worse case directions. The symptoms of the built in anisotropy of the box algorithm can be mitigated in some scenarios by using Oriented friction models or eliminated using IterativeProjectedConeFriction. There are also relevant example implementations in Custom friction models.

More general than box or circle (defined for a *given* normal force F_n), an ideal friction model is required to have a resulting friction force F_f within the so called *friction cone* (**A** in the picture below), for any given normal force. Box friction approximation models will not recover the desired *friction cone* where the resulting friction forces F_f are instead within an inverted *friction pyramid* (**B** in the picture below). **Note** that **box** friction models are optimal for convergence and performance of a solver, especially when the solver is using direct solving methods, e.g., `agx::FrictionModel::DIRECT` Solve type in AGX Dynamics.

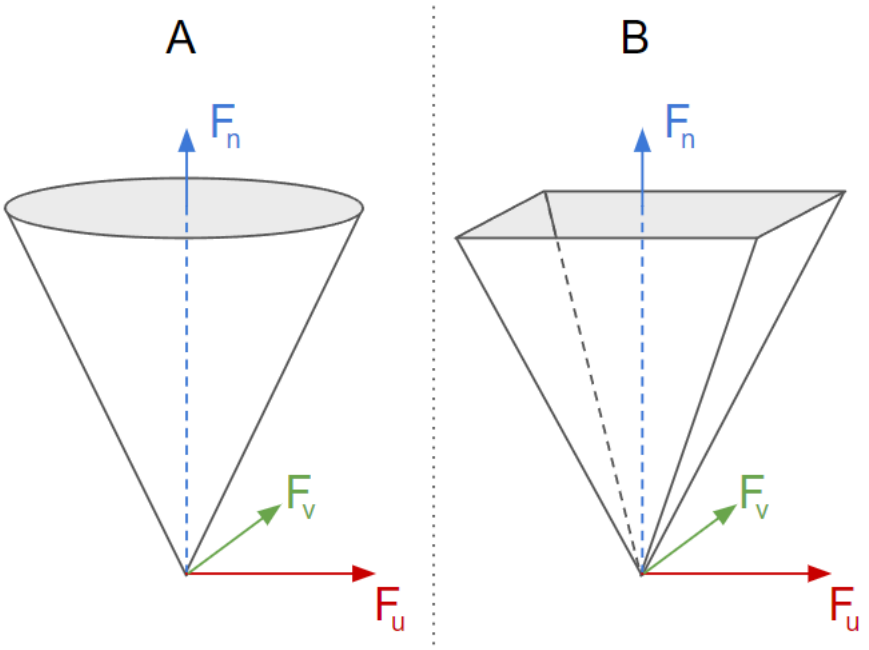


Fig. 11.11 The resulting friction forces should, ideally, be on the surface or inside the *friction cone* (**A**). It's convenient and (computationally) efficient to approximate $\mu_u^{'} = \mu_u$ and $\mu_v^{'} = \mu_v$ which will result in the final friction force to be on the surface of or inside of the (inverted) *friction pyramid* (**B**).

In the different friction model implementations below, assume that the following code has been executed:

```
using namespace agx;

// Create two materials
MaterialRef wood = new Material("wood");
MaterialRef steel = new Material("steel");

// Create two geometries with material wood and steel.
agxCollide::GeometryRef geom1 = new agxCollide::Geometry();
geom1->setMaterial(wood);
agxCollide::GeometryRef geom2 = new agxCollide::Geometry();
geom2->setMaterial(steel);

// Create a contact material that will be used when steel collides with wood
ContactMaterialRef steelWood = simulation->getMaterialManager()->getOrCreateContactMaterial(wood,
                                                                                            steel);
```

11.16.3.1. BoxFrictionModel

This friction model has fixed friction bounds throughout the solve stage. It's (CPU) performance vice "optimal", but is in general not a suitable friction model in simulations where friction and/or accuracy is important.

Normally one would expect the friction force to be:$|F_f| \leq \mu |F_n|$, but since the normal force magnitude $|F_n|$ is unknown before the solver starts, the size of the normal force has to be guessed/approximated. When guessed, the masses of dynamic rigid bodies involved are used so that the order of magnitude matches the scale of the interaction: $|F_n| = (m_1 + m_2)g$, where m_1 and m_2 are masses and g the default gravity acceleration (9.80665…). In ConstantNormalForceOrientedBoxFrictionModel, the normal force magnitude is customizable.

If Contact Warmstarting is enabled, the normal force from the previous time step is used as approximation. This approximation is of course better than using the masses, but is in general also not accurate due to the slow propagation of the nonlinearity of a frictional contact.

```
BoxFrictionModelRef boxFriction = new BoxFrictionModel();

// Use box friction as the friction model for this contact material
steelWood->setFrictionModel(boxFriction);
```

11.16.3.2. ScaleBoxFrictionModel

This friction model has dynamic friction bounds throughout the solve stage, hence “scale” in the name. I.e., the friction bounds will be updated given the current normal force, such that,

$$|F_u| \leq \mu_u |F_n|$$
$$|F_v| \leq \mu_v |F_n|$$

and note the “box” approximation $\mu_u' = \mu_u$ and $\mu_v' = \mu_v$ (**B** in Fig. 11.11 above).

The nonlinear response of the friction force given the normal force is computationally more expensive but results in accurate friction, even though the forces may exceed the Coulomb friction law in some directions (see calculation in Friction model).

```
ScaleBoxFrictionModelRef scaleBoxFriction = new ScaleBoxFrictionModel();

// Use scaled box friction as the friction model for this contact material
steelWood->setFrictionModel(scaleBoxFriction);
```

11.16.3.3. IterativeProjectedConeFriction

This friction model, with solve type SPLIT, is the default in AGX Dynamics. I.e., ContactMaterial friction model instance is by default `nullptr` and this is implicitly equivalent to `new IterativeProjectedConeFriction(FrictionModel::SPLIT)`.

The name “Iterative Projected Cone Friction” originates from an early algorithm where the friction equations were solved with an iterative solver, i.e., solve type SPLIT. For solve type DIRECT, this friction model was identical to ScaleBoxFrictionModel in AGX Dynamics **2.34.0.3 and earlier**. Versions **2.35.0.0 and later** includes improved direct projection algorithms finding μ_u' and μ_v' fulfilling ideal Coulomb friction (**A** in Fig. 11.11 above) for solve types DIRECT, SPLIT and ITERATIVE, i.e.,

$$\begin{aligned} |F_u| &\leq \mu_u' |F_n| \\ |F_v| &\leq \mu_v' |F_n| \\ F_f &= F_u + F_v \\ |F_f| &\leq \sqrt{(\mu_u')^2 + (\mu_v')^2}\, |F_n| \end{aligned}$$

where

$$\begin{aligned} 0 \leq \mu_u' &\leq \mu_u \\ 0 \leq \mu_v' &\leq \mu_v . \end{aligned}$$

See Fig. 11.12 below how the resulting $\mu_{u|v}'$ relates to the given (input) friction coefficients $\mu_{u|v}$.

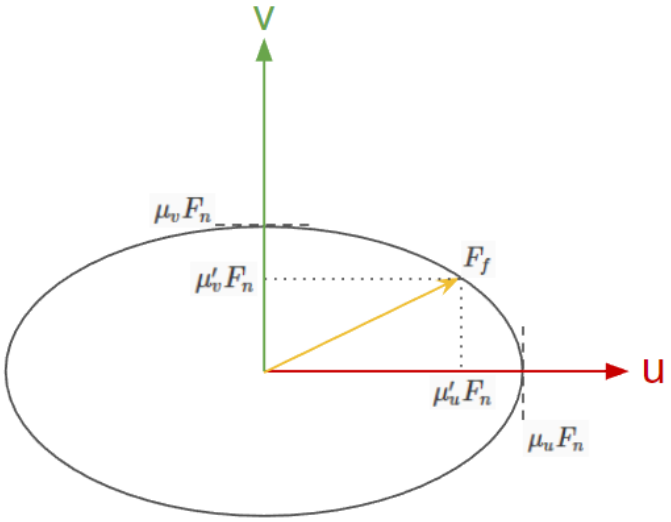


Fig. 11.12 With anisotropic friction ($\mu_u \neq \mu_v$), the circle in Fig. 11.10 is an ellipse, and it’s desired the resulting friction force F_f to be on the surface or within this ellipse. The solver (friction model) is responsible of finding $\mu_{u|v}'$ so that the Coulomb friction law, $|F_f| \leq \sqrt{(\mu_u')^2 + (\mu_v')^2}\, |F_n|$, is fulfilled.

For solve type DIRECT, the **default behavior** is to make some qualified assumptions about the direction of the resulting friction force. This may result in slight overestimated friction forces in some cases, but is much more suitable for nonlinear direct methods with better stability and convergence (close to

optimal ScaleBoxFrictionModel). To enable exact projections, guaranteeing the resulting friction force within the friction cone (Fig. 11.11, Fig. 11.12), use:

```
/**
Enable/disable exact cone projection when solve type is DIRECT. This feature
is disabled by default when it may affect the convergence of the solver. When
enabled, the friction forces will be within the friction cone at any given
resulting direction.
\param enable - enable/disable exact cone projection when solve type is DIRECT
*/
void setEnableDirectExactConeProjection( bool enable );
```

```
IterativeProjectedConeFrictionRef ipcFriction = new IterativeProjectedConeFriction();
ipcFriction->setSolveType(FrictionModel::DIRECT);
ipcFriction->setEnableDirectExactConeProjection(true);
steelWood->setFrictionModel(ipcFriction)
```

```
IterativeProjectedConeFrictionRef ipcFriction = new IterativeProjectedConeFriction();

// Use iterative projected cone friction as the friction model for this contact material
steelWood->setFrictionModel(ipcFriction);
```

11.16.3.4. Oriented friction models

Oriented friction models is of the same type as the ones described above but with the friction box/ellipse oriented with a given frame of reference and a primary friction direction (in the given frame).

```
const auto localPrimaryDirection = agx::Vec3::X_AXIS();
auto obfm = new agx::OrientedBoxFrictionModel(rb1->getFrame(),
                                              localPrimaryDirection);
auto osbfm = new agx::OrientedScaleBoxFrictionModel(rb2->getFrame(),
                                                    localPrimaryDirection);
auto oipcfm = new agx::OrientedIterativeProjectedConeFrictionModel(rb3->getFrame(),
                                                                   localPrimaryDirection);
cm1->setFrictionModel(obfm);
cm2->setFrictionModel(osbfm);
cm3->setFrictionModel(oipcfm);
```

Given a contact with an oriented friction model, the friction plane is calculated given the contact normal and the given primary friction direction. The secondary friction direction is orthogonal to the primary direction.

11.16.3.5. ConstantNormalForceOrientedBoxFrictionModel

This friction model is similar to `OrientedBoxFrictionModel` but it has some additional features:

1. The normal force F_n is given, resulting in the primary friction force F_u to be bound by:$-\mu_u F_n \leq F_u \leq \mu_u F_n$ and the secondary friction force F_v to be bound by:$-\mu_v F_n \leq F_v \leq \mu_v F_n$ – where μ_u is the primary friction coefficient and μ_v the secondary friction coefficient.
2. Contact depth dependent maximum friction forces (disabled by default), by scaling the given normal force with the depth of each contact. Enabling this feature may impact the performance but the resulting behavior of the contacting objects may be better.

11.16.3.6. Custom friction models

Custom friction models are user defined implementations of friction in AGX Dynamics. While it's possible to recreate any native friction model implementation, the value of having a custom implementation is often due to something simulation/system/model dependent that is identified and possible to utilize as a friction model.

There are two methods possible to build the custom friction model around:

- `FrictionModel::initialize`, assigning the initial friction bounds.

```
/**
Initializes bounds given contact point (pointIndex).
\param geometryContact - geometry contact
\param pointIndex - point in geometry contact
\param dt - time step
\param ret_boundU - returned, primary direction bounds
```

```
\param ret_boundV - returned, secondary direction bounds
*/
virtual void initialize( const Physics::GeometryContactPtr geometryContact,
                         size_t pointIndex,
                         agx::Real dt,
                         agx::RangeReal& ret_boundU,
                         agx::RangeReal& ret_boundV ) const override;
```

- `FrictionModel::calculateTangentPlane`, assigning the primary ($\hat{u}$) and secondary ($\hat{v}$) friction directions.

```
/**
Calculates friction plane given normal. Only valid if \p normal,
\p ret_u and \p ret_v are orthogonal.
\param geometry1 - first geometry in the geometry contact
\param geometry2 - the other geometry in the geometry contact
\param point - contact point in world coordinate system
\param normal - contact normal in world coordinate system
\param depth - penetration depth
\param ret_u - returned, primary direction in friction plane
\param ret_v - returned, secondary direction in the friction plane
*/
virtual void calculateTangentPlane( const agxCollide::Geometry* geometry1,
                                    const agxCollide::Geometry* geometry2,
                                    const agx::Vec3& point,
                                    const agx::Vec3& normal,
                                    const agx::Real depth,
                                    agx::Vec3& ret_u,
                                    agx::Vec3& ret_v ) const override;
```

The examples below are implemented in Python because there are some additional remarks compared to a C++ implementation. Converting the examples from Python to C++ is straightforward since the Python API is generated given the C++ API.

▶ [click to expand] Implementation skeleton with important remarks and how to access data.

▶ [click to expand] Randomized tangential directions.

▶ [click to expand] Constant/given normal force box friction model.

11.16.4. Solve type

The solve type of a friction model defines in which type of solver the normal and/or tangents of a frictional contact are solved. In which solver a frictional contact is solved affects performance, accuracy and stability. The default solve type is SPLIT. For more information about the different solvers in AGX Dynamics, see Direct solver, Split solver and Iterative solver, and examples in Using correct solve type.

```
FrictionModel* frictionModel = steelWood->getFrictionModel();

frictionModel->setSolveType(FrictionModel::DIRECT);
frictionModel->setSolveType(FrictionModel::SPLIT);
frictionModel->setSolveType(FrictionModel::ITERATIVE);
frictionModel->setSolveType(FrictionModel::DIRECT_AND_ITERATIVE);
```

11.16.4.1. DIRECT

The contact normals and tangents are solved with the Direct solver. Nonlinear friction models, such as ScaleBoxFrictionModel and IterativeProjectedConeFriction, will result in additional solves when the friction bounds are adjusted given the solved for normal force. The nonlinear friction models with this solve type are the most accurate but requires more calculations to find a solution.

Linear friction models, such as BoxFrictionModel, requires only a single solve when the friction bounds are fixed with an estimated normal force.

11.16.4.2. ITERATIVE

The contact normals and tangents are solved with the Iterative solver. The nonlinearity of some friction models, such as ScaleBoxFrictionModel and IterativeProjectedConeFriction, doesn't require more calculations than the linear (approximated), which makes this solve type the most (CPU) efficient and is therefore the suitable solve type for large contact systems.

11.16.4.3. SPLIT

The contact normals are solved in both the Direct solver and the Iterative solver, while the tangents (friction) are solved with the Iterative solver with the normal forces from the direct solve as input. See Split solver for more information.

As described in the iterative solve type, solving friction with an iterative solver is more efficient than with a direct solver. This solve type will always provide the important responses of the contact normal forces and in many cases provide frictional response forces that are suitable enough in many scenarios.

The friction forces calculated in the iterative solver are transferred back to the direct solver as a weak interaction (external force). If other interactions (e.g., constraints and/or other contacts) exists with direct solve types and are interacting with a split contact, the direct interactions may remove some of the friction force, resulting in viscous (constant speed) slide of the contact in the friction plane. The undesired slide can be mitigated by setting solve type `DIRECT_AND_ITERATIVE` on the constraints interacting with the sliding object or, if possible, change the solve type of the friction model to DIRECT.

11.16.4.4. DIRECT_AND_ITERATIVE

Note

This solve type is experimental and may, in its default behavior, result in too large resulting friction forces.

The contact normals and tangents are solved in both the Direct solver and the Iterative solver. The contact normals and friction bounds are estimated in the Iterative solver by:

1. Direct solve with infinite friction bounds.
2. Iterative solve where the friction bounds are updated as per the Friction model this solve type is set for.
3. Direct solve using the resulting friction bounds from the iterative solve.

This solve type makes all friction models “linear” in the direct solver, i.e., similar to BoxFrictionModel but with the normals and friction bounds estimated with the previous direct and iterative solve in this time step.

The initial direct solve with infinite friction is a probable source to overestimated resulting friction forces. In a generic implementation the bounds may either be zero or infinite, and with an initial direct solve without friction, the resulting friction forces are instead underestimated. If this is a preferred solve type in your application, it may be suitable with a custom implementation where `initialize` is overridden, setting the initial bounds given something application dependent, e.g.,

```
class MyDirectAndIterativeFrictionModel(agx.ScaleBoxFrictionModel):
    @property
    def initial_max_friction_force(self) -> float:
        return self._initial_max_friction_force

    @initial_max_friction_force.setter
    def initial_max_friction_force(self, max_friction_force: float) -> None:
        assert max_friction_force >= 0.0, 'Invalid initial friction force value < 0.0.'
        self._initial_max_friction_force = max_friction_force

    def __init__(self,
                 max_friction_force: float,
                 solve_type: int = agx.FrictionModel.DIRECT_AND_ITERATIVE):
        super().__init__(solve_type)
        self.__disown__()
        self.initial_max_friction_force = max_friction_force

    def initialize(self,
                   gc_ptr: agxCollide.GeometryContactPtr,
                   point_index: int,
                   dt: float,
                   ret_boundU: agx.RangeReal,
                   ret_boundV: agx.RangeReal) -> None:
        # Note that it's not required to use the friction coefficient(s) here,
        # the friction coefficient will be used during the following iterative
        # and last direct solve.
        ret_boundU.setLower(-dt * self.initial_max_friction_force)
        ret_boundU.setUpper(dt * self.initial_max_friction_force)

        ret_boundV.setLower(-dt * self.initial_max_friction_force)
        ret_boundV.setUpper(dt * self.initial_max_friction_force)
```

11.17. Material Library

AGX contains some predefined material properties to help provide a starting point for the Material definitions in a simualtion. The definitions in the Material Library are generic, e.g. *wood* is defined, but not *birch*, *oak* or *balsa tree*. Those different types of wood have rather different properties, e.g. density. So for

specific needs, the Materials used in a Simualtion should be defined according to your needs.

The definitions in the Material Library are limited to contain material properties and does not contain AGX implementation details. If for example a certain friction model is desired, then this has to be configured by the user.

When two materials from the material library interact, the material library is automatically searched for a predefined ContactMaterial for said Material pair. If none such ContactMaterial exists, then an implicit ContactMaterial is created and used.

The Material Library contains the following parts:

- Material definitions
- ContactMaterial definitiions
- TerrainMaterials
- BeamModelProperties
- CableProperties

Note

Some classes use more than just a Material for its definitions. Examples of this is e.g.

- `agxCable::Cable` which use both a Material and CableProperties,
- `agxTerrain::Terrain` which use Material and TerrainMaterial
- `agxModel::BeamModel` which use Material and BeamModelProperties

Example usage of the Material Library:

```
# Load into existing Material
my_material1 = agx.Material()
my_material1.loadLibraryMaterial("wood")

# Get a Material, None on failure
steel = agx.MaterialLibrary.loadMaterial("steel")

# Custom Materials can be based on items defined in the library
my_soft_steel = agx.Material(steel)
# customize my_soft_steel

# Modify the TerrainMaterial for a agxTerrain.Terrain
my_terrain.loadLibraryMaterial("dirt_1")

# Set both Material and CableProperties on a Cable:
my_cable.setMaterial(steel)
my_cable.loadLibraryProperties("cable_1")

# Some class based upon agxModel.BeamModel
my_i_beam.loadLibraryProperties("beam_1")
```

Classes that use properties have the method *getAvailableLibraryProperties()* whereas Terrain and Material have *getAvailableLibraryMaterials()* that can be used to inspect available items. This can also be done via *agx::MaterialLibrary::getAvailableItems(item_type)*.

Which items that are available can also be seen by looking at the contents of *<agx>/data/MaterialLibrary/<item type>/*.

The different items are referenced by their short name. So for example if the Material *steel* is loaded, AGX searches for *MaterialLibrary/Materials/steel.json*. In the same way, if a CableProperty called *cable_1* is requested, then the file *MaterialLibrary/CableProperties/cable_1.json* is search for. When a definition for a ContactMaterial is searched for, the name is the material1-material2 where the materials are in alphabetical order.

The lookup for all items in the material library is done using the *RESOURCE_PATH* and if you want to make AGX search at more locations, see Locating files and resources.

Note

GranularBodySystem settings in ContactMaterial

The ContactMaterial definitions all list default values used by the GranularBodySystem for GraularBody specific parameters such as rolling resistance and twisting resistance. The reason for this is to expose the values in case someone wants to define their own materials. These values are highly shape dependent and what the granulars look like have a much larger effect on the parameter values than the actual combination of materials that interact.

E.g. if the granulars that interact are spheres or boxes influence the values more than if it is wood-wood or wood-steel.

11.17.1. Adding new items to the Material Library

The material library can be extended by placing a json file in the corresponding *MaterialLibrary/<item type>/* directory. Existing files can be used to see which parameters that should be defined.

When adding a Material there is one special value in the file that is important for how contact materials are found. That is the value for *name*. In the files provided with AGX Dynamics, the name value matches the filename, e.g. *MaterialLibrary/Materials/wood.json* have *"name": "wood"*. It is the pair of two such names that are used when searching for a contact material when two library materials interact.

The reason for *"name"* is that one can have e.g. *birch.json* and *pine.json* and both use *wood* for name. Then interactions for the specific birch material would use the predefined contact materials that already exist for wood.

11.18. Closest point computation

Getting the closest distance between two shapes or geometries can be computed using `agxCollide::computeShapeDistance` and `agxCollide::computeGeometryDistance`. This feature is currently only available for convex shapes for which the method `hasSupportFunction()` returns true including convex, sphere, box, convex, cylinder, capsule, cone, hollow cylinder and hollow cone.

The closest distance can be computed between two geometries `computeGeometryDistance` (containing one or more shapes) or between two shapes `computeShapeDistance`. The transformation must always be explicitly specified, even for geometries (see below).

```
// Assume we have two geometries, each with a transformation.
agxCollide::GeometryRef geometry1;
agxCollide::GeometryRef geometry2;

// Closest point on each geometry
agx::Vec3 point1, point2;

// Compute the closest distance
agxCollide::ShapeDistanceResult result = agxCollide::computeGeometryDistance(
  geometry1,
  geometry1->getTransform(),
  geometry2,
  geometry2->getTransform(),
  point1, point2);

  // If result is success we can use the output!
  if (result == agxCollide::ShapeDistanceResult::SUCCESS)
    std::cerr << "Distance: " << (point2 - point1).length() << std::endl;
  else if(result == agxCollide::ShapeDistanceResult::OVERLAPPING_SHAPES)
    std::cerr << "Shapes are overlapping, no information available" << std::endl;
```

See also `tutorials/tutorial_shape_distance.agxPy` for more examples of how to use this functionality.

11.19. Testing user geometry against Space

In some cases it is of interest to intersect the current configuration in the collision space with some geometries, without actually inserting a new geometry into the collision space.

As soon as a geometry is inserted into space, it is a part of the simulation. Also testing for contacts in between two simulation frames can sometime be used for example for path finding etc.

There are two methods for achieving this in `agxCollide::Space`:

```
bool Space::testBoundingVolumeOverlap(
      agxCollide::GeometryPtrVector& testGeometries,
      agxCollide::GeometryPairVector& result,
     bool skip = true );

bool Space::testGeometryOverlap(
     agxCollide::GeometryPtrVector& testGeometries,
     agxCollide::GeometryContactVector& result,
     bool skip = true );
```

testBoundingVolumeOverlap will take a vector of Geometries and return pairs of Geometries with overlapping bounding volumes.

testGeometryOverlap will take a vector of Geometries and return a vector of geometry contacts, including contact points, penetration, normals etc. …

11.20. Reading forces from the simulation

Often when you build a simulation you also want to read forces/torques from the system. Assume you have built your scene with geometries, bodies, constraints and wires. Now how do you access the forces from your simulation?

Note

As in many parts of this documentation, the term “force” is here used interchangeably for a 3d force vector in contrast to a 3d torque vector and, where applicable, as a 6d vector consisting of both force and torque.

Forces can be divided into a few separate categories:

- External/ForceField/Gravity forces
- Constraint forces
- Contact forces

11.20.1. External/ForceField/Gravity forces

Forces and torques can be applied to a body by the methods:

`RigidBody::addForce, RigidBody::addTorque`, etc. see 11.12.

External forces also include forces added by ForceFields, GravityFields. These external forces can at any time be read from a given rigid body with the method:

```
const agx::Vec3& RigidBody::getLastForce() const;
const agx::Vec3& RigidBody::getLastTorque() const;
```

11.20.2. Constraint forces

It is possible to read constraint forces applied to rigid bodies during the last time step *for each constraint*. However, it is currently not possible to ask a rigid body for its constraint forces directly - you have to keep track of the constraints attached to a specific body yourself and ask those constraints for the forces.

11.20.2.1. Constraint::getLastForce per body

For computational reasons, the calculation of constraint forces is disabled by default. To enable the computation of constraint forces call the method:

```
void Constraint::setEnableComputeForces(bool);
```

once for each constraint that you want to read forces from in the future.

Then, *after the solver has been executed*, you can access (e.g. with a StepEventListener in POST) the forces/torques applied to the constrained bodies with a call to:

```
// If you know the index of your body in the Constraint (0 or 1)
bool Constraint::getLastForce( UInt bodyIndex, Vec3& retForce, Vec3& retTorque, bool giveTorqueAtCm=false ) const;

// If you have pointer to the rigid body
bool Constraint::getLastForce( const RigidBody* rb, Vec3& retForce, Vec3& retTorque, bool giveTorqueAtCm=false ) const;
```

Note

Note that getLastForce is only valid when called for a rigid body with motion control DYNAMICS or KINEMATICS (not STATIC). Check the return value to detect invalid calls to getLastForce.

The returned force and torque by getLastForce are defined in the world coordinate system.

The returned torque depends on the value of the flag *giveTorqueAtCm*:

- A value of *true* gives the torque in the body’s center of mass (corresponding to `RigidBody::addTorque(Vec3)`),
- whereas the default value of *false* gives the value in the constraint’s attachment point *r* for the body, so that *tau_attachment* = *tau_cm* + *r cross F*

11.20.2.2. Constraint::getCurrentForce per constraint-DOF

If you are interested in obtaining the force applied by the constraint at a certain degree of freedom (in the constraint’s local coordinate system), you can use

```
// \return the current force magnitude (i.e., the force magnitude last time step) in given DOF >= 0 && <= NUM_DOFS
```

```
bool Constraint::getCurrentForce( UInt degreeOfFreedom) const;
```

Note that the degrees of freedom have different meanings for each constraint type; the `enum DOF` which is defined per constraint should be used.

The method *getCurrentForce* exists also for secondary constraints (see Section 12.13).

Attention

The naming difference in *getCurrentForce* and *getLastForce* is misleading since both methods read the forces from the same source and time. These methods will likely be renamed in the future to reflect their behavior better.

11.20.3. Contact forces

Contact forces are included in the contact data, together with contact points, normals depths, … . Contact forces are valid *after the solver has been executed*. You can access them (e.g. with a StepEventListener in POST) like in the following example.

Assume you want to collect the sum of the absolute values of all the contact forces (normal and friction) being applied on a specific rigid body. One way is to explicitly iterate through all geometry contacts. This will give you the best control and will allow you to read out all the available data from contacts. Below is an example demonstrating how to access the contact points via the `agxCollide::Space object`. More examples can be found in the tutorial file *tutorials/agxOSG/tutorial_contactForces.cpp*.

```
#include <agxCollide/Contacts.h>
#include <agxCollide/Space.h>

// Return the sum of the magnitude for all contact forces being applied to a specific body.
agx::Real getContactForce(agx::RigidBody* body, agxCollide::Space* space)
{
  // Get the vector with all the contacts
  const agxCollide::GeometryContactPtrVector& contacts = space->getGeometryContacts();

  agx::Real contactForce = 0;
  // For all contacts in the system
  for (size_t i = 0; i < contacts.size(); ++i)
  {
    agxCollide::GeometryContact *gc = contacts[i];
    // Is the contact enabled?
    if (!gc->isEnabled())
      continue;
    // Is our body involved in this contact?
    if (gc->rigidBody(0) != body && gc->rigidBody(1) != body)
      continue;

    // Get all the points and sum the magnitude for the contact force (normal force)
    const agxCollide::ContactPointVector& points = gc->points();
    for (size_t n = 0; n < points.size(); ++n)
    {
      // Is the contact point enabled?
      if (!points[n].enabled())
        continue;
      contactForce += points[n].getNormalForceMagnitude();
    }
  }
  return contactForce;
}

// Build your simulation
// ...

// Create a body for which you want to read contact forces
agx::RigidBody *myRigidBody = new agx::RigidBody(new agxCollide::Geometry(new agxCollide::Sphere(0.3)));
simulation->add(myRigidBody);
simulation->stepForward();

agx::Real contactForce = getContactForce(myRigidBody, simulation->getSpace());
```

11.20.3.1. ContactForceReader

An alternative is to use the `agx::ContactForceReader` class. It has a simple interface and will perform the filtering and looping over contacts for you. It can collect contact forces based on some interacting rigid bodies or geometries.

```
// Create a ContactForceReader object with a specific simulation.
agx::ContactForceReaderRef contactForceReader = new agx::ContactForceReader(simulation);

// Assume we have a simulation containing two rigid bodies (with geometries), ground and box:
agx::RigidBody* box = createARigidBodyBox();
agx::RigidBody* ground = createAGroundBody();

// Use a ContactForceReader to extract the sum of normal and frictional forces between two rigid bodies:
auto displayContactForces = [application](agxSDK::Simulation* , agx::ContactForceReaderRef contactForceReader,
agx::RigidBody *ground, agx::RigidBody *box)
{
  auto sceneDecorator = application->getSceneDecorator();
  sceneDecorator->clearText();
  sceneDecorator->setText(0, "Tutorial contact forces: ContactForceReader Friction force.");

  // Get the total normal force between the two rigid bodies:
  agx::Vec3 normalForce = contactForceReader->getNormalContactForces(ground, box);

  // Get the total frictional force between the two rigid bodies
  agx::Vec3 frictionForce = contactForceReader->getTangentialContactForces(ground, box);

  sceneDecorator->setText(1, agx::String::format("  - Current normal force:          %.10f N", normalForce.length()));
  sceneDecorator->setText(2, agx::String::format("  - Current friction force:          %.10f N",
frictionForce.length()));

};
agxUtil::StepEventCallback::post(displayContactForces, simulation, contactForceReader, ground, box);
displayContactForces(simulation, contactForceReader, ground, box);
```

11.21. ObserverFrame

`agx::ObserverFrame` is a class that implements a coordinate transform that can be attached relative to a rigid body that will follow the movements of the body. Given an observer frame the following data can be queried:

- Position
- Rotation
- Linear/angular velocity
- Linear/angular acceleration
- Relative position, rotation, acceleration

This means that you can use an observer for measuring data in a system: What is the linear velocity on this point of a rigid body? The linear velocity will then be computed based on the rigid body's linear and angular velocity. A typical scenario is to attach an observer to the chassie of a car to measure various data for the vehicle such as its position, heading, velocity etc.

Below is a code snippet illustrating the use of an ObserverFrame. There is also a python tutorial available: `tutorial_observerFrame.agxPy`.

```
// Create an observer that is attached to a specific RigidBody:
// It will get a name and a transformation relative to the attached rigidBody.
agx::ObserverFrameRef observer = new agx::ObserverFrame("MyObserver", rigidBody, agx::AffineMatrix4x4::translate(1, 0,
0));

// Add the observer to the simulation, only needed if you do not have a reference pointer to it, or
// if you want to find it later using: simulation->getObserver("MyObserver");
simulation->add(observer);

// Later in the simulation, after stepping, in an event listener etc. you can query the observer for data:
auto speed = observer->getVelocity().length();

// Simulate for 10 seconds
simulation->stepTo(10);

// Check if acceleration exceeds 9g -> damage to the person riding the car?
auto accelerationMagnitude = observer->getAcceleration().length();
```

```
if (accelerationMagnitude > 9 * agx::GRAVITY_ACCELERATION)
  std::cerr << "Driver probably unconscious" << std::endl;
```

12. Constraints

A Constraint will define a geometrical relationship between one rigid body relative to the world, or two or more rigid bodies relative to each other, that should be met under certain conditions. Each constraint is built upon *elementary constraints*, which define the geometrical relationship. Acting upon these elementary constraints can be *secondary order* constraints, motors, limits or locks.

Example of how to use the constraint API is showcased in the **tutorials/tutorial_tutorial5.cpp**

One important thing to keep in mind when constructing systems with constraints is that:

Attention

Always position/rotate the rigid bodies involved in a constrained system before attaching them to a constraint.

Available constraints in the current AGX version are:

Table 12.1 Available constraints

CONSTRAINT NAME	DESCRIPTION
BallJoint	Ball and socket joint
DistanceJoint	Restore distance between two specified points on bodies.
Hinge	Allow rotation around one specified axis (revolve)
UniversalJoint	Transmission joint.
LockJoint	Removes all degrees of freedom between two bodies, keep the locked together.
Prismatic	Allows translation along one axis (slide)
PlaneJoint	Reduce translation movement to a plane
CylindricalJoint	Hinge and Prismatic combined
AngularLockJoint	Reduces all rotational DOF between two bodies.
SplineJoint	Allows movement along a spline. Related to CylindricalJoint.
WheelJoint	Allows for rotation around two axes (wheel, steering) and movement along one (steering for suspension).

Some of the constraints have one additional version which includes support for having slack in the joint to model a non-ideal joint. These are:

Table 12.2 Available constraints with slack

CONSTRAINT NAME
SlackCylindricalJoint
SlackHingeJoint
SlackLockJoint
SlackPrismaticJoint

Slack joints can be used to model a system with a specified gap. Let's say you are modeling a robot with a tool. The tool is attached to the robot arm, but the tool has a few millimeters slack in translation. Modeling this using collision detection can be complicated and resource demanding. Small features usually mean you have to use shorter time steps. With a slack joint, this gap can be modelled using a constraint. This allows for better performance and much more reliable result.

Each slack joint has *slack range parameters* which can be controlled. Depending on which constraint is being used, there are different number of degrees of freedom available for the slack modeling. For example, the SlackLockJoint has 6 DOF for slack modelling, whereas the SlackPrismaticJoint only has 5 (the fifth is the free DOF controlled using motors/lock/range).

The slack can be both positional (meter) and rotational (radians). If the slack feature is not needed, then it is better to use the ordinary version of the constraint. The reason for this is that the ordinary version uses half the number of equations compared to the slack version and will therefore be faster for the solver to handle.

Note

Currently the slack joints only has one argument for the rotational slack. This means that the rotational slack is applied to all rotational degrees of freedoms.

12.1. Elementary constraints

Elementary constraints are constraints that ordinary, more complex constraints are based upon. These elementary constraints are:

Table 12.3 Elementary constraints

NAME	DESCRIPTION
Dot-1	Given two vectors v_1 and v_2, this constraint will retain $v_1 \cdot v_2 = 0$.
Dot-2	Given a separation vector d^{ij} between body *i* and *j* and and body fixed vector a^i, this constraint will retain $d^{ij} \cdot a^i = 0$.
Spherical	This is a 3D constraint which computes $x^i + p^i - x^j - p^j = 0$, where **x** is a body fixed vector and **p** the position of that vector.
Distance	Given two positions p^i and p^j, this constraint will retain $\| p^i - p^j \| = l_o$, where l_0 is some constant.
QuatLock	Locks all rotational degrees of freedom between two bodies.

For example, a hinge is a composite of two Dot-1 and one Spherical elementary constraint.

12.1.1. Converting spring constant and damping

In the agxUtil namespace there are functions for converting from the spring constant and damping to AGX regularization parameters:

Converting from a spring constant to compliance: compliance = 1/springConstant:

```
// Convert from a spring constant to compliance
agx::Real compliance = agxUtil::convert::convertSpringConstantToCompliance( springConstant );
// Use the compliance
constraint->setCompliance( compliance );
```

Convert (viscous) damping coefficient to spook damping, where:

dampingCoefficient – Damping. For linear dimensions, in force*time/distance; for rotational dimensions, in torque*time/radians.

springConstant – Spring constant. For linear dimensions, in force/distance; for rotational dimensions, in torque/radians.

```
agx::Real spookDamping = agxUtil::convert::convertDampingCoefficientToSpookDamping(
   dampingCoefficient, springConstant);

constraint->setDamping( spookDamping );
```

12.1.2. Regularization parameters

Ordinary constraints can be modeled using regularization parameters. These parameters specify the *compliance* i.e. stiffness of the constraint and damping, i.e. how fast the constraint should be restored to a non-violated state.

The regularization parameters are accessed with the call:

```
// Set compliance for all DOF:s
hinge->setCompliance( 1E-14 );
```

To access the regularization parameters for each degree of freedom (DOF):

```
// Set compliance for all DOF:s
for (size_t dof=0; dof < hinge->getNumDOF(); dof++)
  hinge->getRegularizationParameters( dof )->setCompliance( 1E-4 );
```

The ordering of DOF is different per constraint; see the `enum DOF` which is defined per constraint in order to set meaningful values per DOF.

12.1.3. Linearization

Compliance control how much a constraint will be deformed, or violated, when put under load from a force or torque. The deformation follows Hook's law, meaning that with a compliance *c* and a load *l* the deformation *d* at rest will be

$$d = l \cdot c$$

The unit of *d* is typically unit length for translational elementary constraints and radians for rotational elementary constraints. However, for elementary constraints based on quaternions, such as QuatLock, the deformation will be linear in quaternion space instead. To help account for this `agx::Constraint` provides a linearization feature, enabled with the member function `setEnableLinearization`. When enabled for a constraint any elementary constraint that doesn't produce linear deformations natively will adjust the parameters it sends to the solver to make it appear as-if the

elementary constraint was linear according to Hook's law. The adjustment depend on the angle of the constraint and the linearization process uses the angle at the start of the time step. If the relative angular velocity between the constrained bodies is high then the non-linear response of the QuatLock will dominate and the linearization will produce inaccurate results. The linearization is accurate when the relative angular velocity of the constrained bodies is low.

12.1.4. Attachment frames

All constraints uses agx::Frame to define the geometrical relationship between the constrained body and the world, *or* between two constrained rigid bodies.

A Frame for a constraint defines a coordinate system attached to a rigid body. The constraint is satisfied/relaxed when these two coordinate systems/frames are coincident in the world. For example, assume a LockJoint is created between the two rigid bodies A and B below. When creating the constraint, two agx::Frame are used. The frames are translated relative to the origin of the bodies (which are assumed to be the green dot in the center).

```
agx::RigidBodyRef A, B; // Two bodies
// Create an attachment frame relative body A:s model origin
agx::FrameRef frameA = new agx::Frame();
frameA->setLocalTranslate(-0.5,0,0.5); // Translate the frame left and up relative to body A.
A->addAttachment(frameA, "frameA");

// Create an attachment frame relative body B:s model origin
agx::FrameRef frameB = new agx::Frame();
frameB->setLocalTranslate(0.5,0,-0.5); // Translate the frame right and down relative to body B.
B->addAttachment(frameB, "frameB");

// Create a LockJoint using the attachments
agx::LockJointRef lock = new agx::LockJoint(a, a->getAttachment("frameA"),
                                            b, b->getAttachment("frameB"));
```

Fig. 12.1 below show the initial configuration (left) after executing the above code. To the right, the relaxed configuration is shown. *When the two attachment frames are coincident in space, the constraint is relaxed.*

Notice also that a frame can be associated to a rigid body using a name: **RigidBody::addAttachment(Frame *, const agx::String&)** This only means that the rigid body will keep a reference to the frame for later use. Think of attachments as "snap on" targets.



Fig. 12.1 Left, initial configuration. Right, relaxed configuration.

12.2. Ordinary constraints

Ordinary constraints (derived from the class *agx::Constraint*) are composites of elementary and secondary constraints. They can be used in a simulation to build up machines, such as cars, elevators etc. Ordinary constraints also consist of *secondary constraints*, which can define motors and limits. A body without constraints, is free to *translate* (t_1, t_2, t_3) and *rotate* (r_1, r_2, r_3), a total of 6 degrees of freedom (DOF). A constraint will restrict the motion of an attached body to one or more DOF.

For example, a *Prismatic* will allow a body to translate along one axis only, the other five DOF are constrained, while a *LockJoint* will restrict the movement of a body to all six DOF. Table 12.4 below show the number of degrees of freedom removed for each constraint.

Table 12.4 Degrees of freedoms removed for constraints

CONSTRAINT	#DEGRESS OF FREEDOMS REMOVED	#TRANSLATIONAL	#ROTATIONAL
Hinge	5	3	2
LockJoint	6	3	3
AngularJoint	3	0	3
Prismatic	5	2	3
CylindricalJoint	4	2	2
DistanceJoint	1	1	0

CONSTRAINT	#DEGRESS OF FREEDOMS REMOVED	#TRANSLATIONAL	#ROTATIONAL
UniversalJoint	4	3	1
BallJoint	3	3	0
SplineJoint	4	2	2
WheelJoint	3	2	1

Table 12.5 lists each ordinary constraint with their respective elementary and secondary constraints.

Table 12.5 Elementary- and SecondaryConstraints usage for Constraints

Constraint	ElementaryConstraints	SecondaryConstraints
Hinge	1 x Spherical, 2 x Dot1	Range1D, Motor1D, Lock1D
LockJoint	1 x Spherical, 1 x QuatLock	None
AngularJoint	1 x QuatLock	None
Prismatic	1 x QuatLock, 2 x Dot2	Range1D, Motor1D, Lock1D
CylindricalJoint	2 x Dot1, 2 x Dot2	2xRange1D, 2xMotor1D, 2xLock1D
DistanceJoint	1 x Distance	Range1D, Motor1D, Lock1D
UniversalJoint	1 x Spherical, 1 x Dot1	Motor
BallJoint	1 x Spherical	None
SplineJoint	2 x Dot1, 2 x Dot2	2xRange1D, 2xMotor1D, 2xLock1D, 1xKinematic Body
WheelJoint	2x Dot2, 1x Dot1	4 x Range1D, 3 x Motor1D, 3 x Lock1D

Each constraint declares an enum, specifying an index for each DOF that it removes:

```
enum DOF
{
  ALL_DOF=-1,        /**< Select all degrees of freedom */
  TRANSLATIONAL_1=0, /**< Select DOF for the first translational axis */
  TRANSLATIONAL_2=1, /**< Select DOF for the second translational axis */
  TRANSLATIONAL_3=2, /**< Select DOF for the third translational axis */
  ROTATIONAL_1=3,    /**< Select DOF corresponding to the first rotational axis */
  ROTATIONAL_2=4,    /**< Select DOF corresponding to the second rotational axis */
  ROTATIONAL_3=5,    /**< Select DOF corresponding to the third rotational axis */
  NUM_DOF=6          /**< Number of DOF available for this constraint */
};
```

Which can be used as follows:

```
// Set the compliance for translational movements for the hinge:
hinge->setCompliance( 1E-8, Hinge::TRANSLATIONAL_1 );
hinge->setCompliance( 1E-8, Hinge::TRANSLATIONAL_2 );
hinge->setCompliance( 1E-8, Hinge::TRANSLATIONAL_3 );
```

Most constraints will also return the number of DOF it applies onto a body with the method:

```
// Get the number of DOF for this constraint. -1 if numDof is not defined for the constraint.
int numDof = hinge->getNumDOF();
```

12.3. Hinge

A Hinge removes 5 degrees of freedom from the constrained body allowing for rotation around an axis. A hinge has an angular range, lock and motor. The geometrical configuration of a hinge in AGX is defined by using Attachment frames (12.1.4). By default, a hinge revolves around the positive Z-axis.

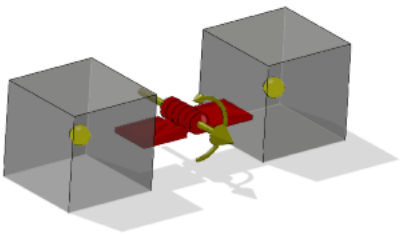


Fig. 12.2 Illustration of a hinge where (1) is the hinge center and (2) the axis. The bodies are free to rotate around this axis.

Hinge example 1: Create a single body hinge (connected to world)

```
// body1 is a rigid body
agx::FrameRef frame = new agx::Frame();
frame->setLocalTranslate( -1.0, 0.0, 0.0 ); // Hinge axis translated -1 along X

// Align hinge axis along negative y-axis instead of Z.
frame->setLocalRotate( agx::Quat(agx::Vec3::Z_AXIS(), agx::Vec3::Y_AXIS()*(-1)) );
agx::ref_ptr< agx::Hinge > h = new agx::Hinge( frame, body1 );
```

Hinge example 2: Connect two bodies with a hinge which reproduces the example from Fig. 12.2 assuming body1 is the one to the left:

```
// body1 and body2 are rigid bodies
agx::FrameRef frame1 = new agx::Frame;
agx::FrameRef frame2 = new agx::Frame;
frame1->setLocalTranslate( 1,0,0);
frame1->setLocalRotate( agx::EulerAngles(agx::degreesToRadians(90.0),0,0) );
frame2->setLocalTranslate( -1,0,0);
frame2->setLocalRotate( agx::EulerAngles(agx::degreesToRadians(-90.0),0,0) );

agx::HingeRef h = new agx::Hinge( body1, frame1, body2, frame2 );
```

Finally, to be realized, the constraint has to be added to the simulation.

```
simulation->add( h.get() );
```

12.4. Ball Joint

A ball joint removes the three translational degrees of freedom for a constrained body.

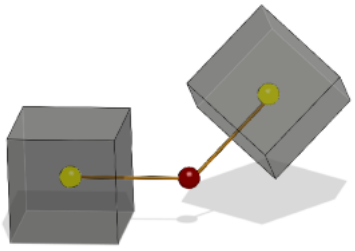


Fig. 12.3 Illustration of a Ball joint where the red sphere is the ball joint center.

Ball joint example 1: Create a single body Ball joint (connected to world) at the origo of the rigid body:

```
// body1 is a rigid body
agx::FrameRef frame = new agx::Frame();
agx::BallJointRef b = new agx::BallJoint(body1  frame);
```

Ball joint example 2: Connect two bodies with a Ball joint

```
// body1 and body2 are rigid bodies
agx::FrameRef frame1 = new agx::Frame();
agx::FrameRef frame2 = new agx::Frame();

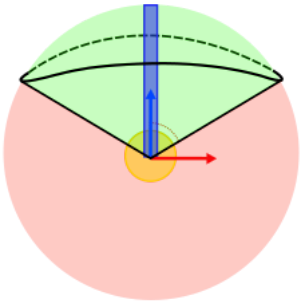
frame1->setLocalTranslate(-1,0,0);
frame2->setLocalTranslate(1,0,0);

// Create A BallJoint using the two frames
agx::BallJointRef b = new agx::BallJoint( body1, frame1, body2,       frame2 );
```

```
simulation->add( b );
```

12.4.1. ConeLimit

A ball joint does not have a range limit, instead it has a cone limit. You can imagine it as if there is a cone in the frame of the base attachment, the z-axis of the other attachment is then limited to move inside that cone.



In the ball joint, the second attachment will always be the base attachment of the cone limit.

The cone limit has two limiting angles. Setting them to the same value will create a simple circular cone limit and setting them to different values will result in an elliptic cone limit. Note that the limit angles have to be smaller than $\frac{\pi}{2}$.

```
// Enable cone limit, by default, limit angles are both PI/4
ballJoint->getConeLimit()->setEnable(true);

// Set limit angles
ballJoint->getConeLimit()->setLimitAngles(PI/4, PI/3);
```

12.4.2. TwistRangeController

The TwistRangeController limits how much an object can rotate around its own z-axis, relative to another object's z-axis. Just as for the cone limit, the second attachment of the ball joint will always be seen as the base attachment, and the first one is what we limit relative to that one.

Note

The TwistRangeController fails when the z-axis of the frame is exactly opposite the z-axis of the reference frame. This controller is intended to be used together with the ConeLimit, where this cannot occur.

The range that can be set on the twist range controller is limited to be smaller than π and larger than $-\pi$. Setting the angles to the same value will create a lock at that angle.

```
// Enable twist range, by default the range is -3/4 pi to 3/4 pi
ballJoint->getTwistRangeController()->setEnable(true);
```

```
// Set the range limit
ballJoint->getTwistRangeController()->setRange(-PI/4, PI/4);
```

12.4.3. FrictionControllers

The FrictionControllers in the ball joint works much in the same way as FrictionController - joint internal friction.The difference is that the ball joint contains multiple friction controllers. There is one for each of the three rotational degrees of freedom. The friction coefficient can be set individually on each of them, or it is possible to use the help functions on the ball joint to set it on all of them at once.

```
// Get the friction controllers of the different rotational directions
FrictionControllerRef frictionControllerU = ballJoint->getRotationalFrictionControllerU();
FrictionControllerRef frictionControllerV = ballJoint->getRotationalFrictionControllerV();
FrictionControllerRef frictionControllerN = ballJoint->getRotationalFrictionControllerN();

// Get all rotational friction controllers
FrictionControllerRefSetVector frictionControllers = ballJoint->getRotationalFrictionControllers();

// Enable and set friction coefficient to the same value on all rotational friction controllers
ballJoint->setEnableRotationalFriction(true);
ballJoint->setRotationalFrictionCoefficient(0.1);
```

There are two additional FrictionControllers that can be used to set friction on the cone limit. One controls the friction for sliding along the limit, and the other for rotating on the limit. These FrictionControllers can be accessed separately or all at once, just as with the rotational friction controllers.

```
// Get the specific friction controllers
FrictionControllerRef limitFrictionController = ballJoint->getConeLimitFrictionControllerLimit();
FrictionControllerRef rotationalFrictionController = ballJoint->getConeLimitFrictionControllerRotational();

// Get all the cone limit friction controllers
FrictionControllerRefSetVector frictionControllers = ballJoint->getConeLimitFrictionControllers();

// Enable and set friction coefficient to the same value on all cone limit friction controllers
ballJoint->setEnableConeLimitFriction(true);
ballJoint->setConeLimitFrictionCoefficients(0.1);
```

Note that if the cone limit is not active, the cone limit friction controllers will not apply any forces at all, since they are only active on the limit.

12.5. Universal joint

A universal joint transfer rotation via a central point through two rotation axes. The two rotation axes can be “bent” against each other. A universal joint is usually modeled with two hinges, however this requires an additional rigid body at the central point. The universal joint does not require this extra body. In addition the universal joint has motor, lock and range for the two remaining rotation dof’s.

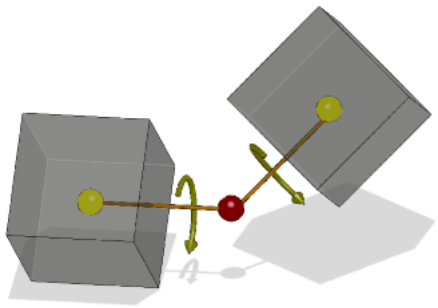


Fig. 12.4 Illustration of a Universal joint where the red sphere is the universal joint center and the orange sticks are the rotational axes for each body, (rotations about this axis is equal for both bodies). The bodies are free to move around the universal center and rotation about the universal axis is constrained (equal rotation).

Universal joint example: Connect two bodies with a Universal joint

```
// Creating a universal joint, axis along world z.
agx::FrameRef rbFrame = new agx::Frame();
rbFrame->setTranslate(0,0,1.5);
```

```
agx::FrameRef rb2Frame = new agx::Frame();
rb2Frame->setTranslate(0,0,-1.5);
agx::UniversalJointRef universal = new agx::UniversalJoint( body1, rbFrame, body2, rb2Frame );
```

Then to realize the constraint it has to add the constraint into the simulation:

```
simulation->add( universal );
```

12.5.1. Motor

A universal joint has two motors, one for each rotational axis which can be individually controlled:

```
// Enable the two motors
universal->getMotor1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_1 )->setEnable( true );
universal->getMotor1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_2 )->setEnable( true );

// Set a speed to the two motors
universal->getMotor1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_1 )->setSpeed( 0.5 );
universal->getMotor1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_2 )->setSpeed( 0.2 );
```

12.5.2. Range

A universal joint also has two ranges, one for each rotational axis:

```
// Enable the ranges
universal->getRange1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_1 )->setEnable( true );
universal->getRange1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_2 )->setEnable( true );

// Set the ranges
universal->getRange1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_1 )->setRange( -agx::PI*0.3, agx::PI*0.3 );
universal->getRange1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_2 )->setRange( -agx::PI*0.3, agx::PI*0.3 );
```

12.5.3. Lock

The universal joint also has a lock for each rotational axis:

```
// Enable the locks
universal->getLock1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_1 )->setEnable( true );
universal->getLock1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_2 )->setEnable( true );

// Set the lock positions
universal->getLock1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_1 )->setPosition(agx::PI);
universal->getLock1D( agx::UniversalJoint::ROTATIONAL_CONTROLLER_2 )->setPosition(agx::PI);
```

12.6. Prismatic universal joint

A prismatic universal joint is the same type of constraint as a universal joint but with an addition of a prismatic joint with its own motor, lock and range. Constructing a prismatic universal joint is analogue to all other joints.

```
// Creating a universal prismatic joint, axis along world z.
agx::FrameRef rbFrame = new agx::Frame();
rbFrame->setTranslate(0,0,1.5);
agx::FrameRef rb2Frame = new agx::Frame();
rb2Frame->setTranslate(0,0,-1.5);
agx::PrismaticUniversalJointRef universal = new agx::PrismaticUniversalJoint(
                                                                                body1,
rbFrame, body2, rb2Frame );
```

Accessing the prismatic motor, range and lock is done through the same interface as for a universal joint:

```
// Enable the prismatic motor
universal->getMotor1D( agx::UniversalJoint::TRANSLATIONAL_CONTROLLER_1 )->setEnable( true );
```

```
// Set the force range and speed
universal->getMotor1D( agx::UniversalJoint::TRANSLATIONAL_CONTROLLER_1 )->setSpeed( 0.1 );
universal->getMotor1D( agx::UniversalJoint::TRANSLATIONAL_CONTROLLER_1 )->setForceRange(-100,100);
```

12.7. Distance joint

Distance joints remove one translational degree of freedom for a constraint rigid body. A Distance joint can also be seen as a spring. It will try to restore the distance between two attached rigid bodies. (Or an attachment point in the world and one rigid body). A Distance joint also has a motor which can push constrained bodies away or pull them together. This can be used to simulate a hydraulic actuator.

Distance joint example1: Connect one body to world with a Distance joint

```
// body1 is a rigid body
// Given in body frame
// Position in the world where the attachment of the distance joint is
agx::Vec3 worldConnectPoint( 0.0, 0.0, 10.0 );
agx::FrameRef frame = new agx::Frame(); // Other end of joint is at origo of rigid body
agx::DistanceJointRef d = new agx::DistanceJoint(body1, frame, worldConnectPoint);
```

Distance joint example 2: Connect two bodies to each other with a Distance joint and enable the motor:

```
// body1 and body2 are rigid bodies
agx::FrameRef frame1 = new agx::Frame()
frame1->setLocalTranslate(
agx::FrameRef frame2 = new agx::Frame()

agx::DistanceJointRef d = new agx::DistanceJoint(body1,frame1, body2, frame2);

// Disable the lock so that it will not interfere with the motor
d->getLock1D()->setEnable(false);

// Enable the motor and increase the distance between the two bodies with 0.1 m/s
d->getMotor1D()->setEnable(true);
d->getMotor1D()->setSpeed(0.1);

// set locked at zero speed, to avoid slipping when the distance is supposed to be constant
d->getMotor1D()->setLockedAtZeroSpeed(true);
```

12.8. LockJoint

A LockJoint will remove all six DOF from an attached body.

To lock a single body to the world or two bodies together it is possible to use the AGX Lock joint. Lock joint example 1: Connect one body to the world with a Lock joint

```
// body1 is a rigid body
agx::LockJointRef l = new agx::LockJoint( body1 );

simulation->add( l.get() );
```

Lock joint example 2: Connect two bodies to each other with a Lock joint

```
// body1 and body2 are rigid bodies
body1->setPosition( agx::Vec3( 10.0, 0.0, 0.0 ) );
body2->setPosition( agx::Vec3( 5.0, 0.0, 0.0 ) );
agx::Frame frame1 = new agx::Frame;
frame1->setLocalTranslate(-2.5,0,0);
frame2->setLocalTranslate(2.5,0,0);

agx::LockJointRef l = new agx::LockJoint(body1, frame1, body2, frame2);

simulation->add( l );
```

12.9. Prismatic

A prismatic constraint will remove 5 DOF from any attached rigid body. The body is free to translate along a specified axis. Compare to an elevator.

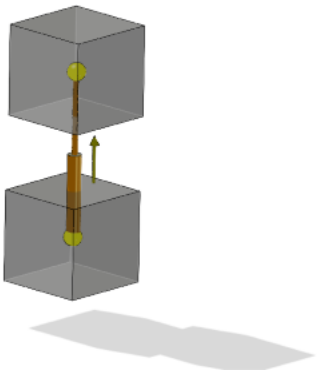


Fig. 12.5 Illustration of a Prismatic joint. Bodies are free to translate along one axis. No rotation is allowed around this axis.

12.10. CylindricalJoint

A CylindricalJoint is a 2D constraint. It consists of a hinge and a prismatic. It means that two degrees of freedom are left for a constrained body, one rotational and one translational. Basically, the constrained body can translate along the specified axis as well as rotating around it. Both has a motor, range and lock.

Special for the cylindrical constraint is that it also has a Screw1D secondary constraint (more about secondary order constraints in 12.13).

Each of the motors, locks and ranges can be accessed through the call:

```
agx::Motor1D* linearMotor = cylindrical->getMotor1D( agx::Constraint2DOF::FIRST );
agx::Motor1D* angularMotor = cylindrical->getMotor1D( agx::Constraint2DOF::SECOND );
```

12.10.1. Screw1D

The Screw1D secondary order constraint connects the prismatic (linear) and the hinge (rotational) part of a CylindricalJoint.

If you try to translate the constrained body, it will also start to rotate. The ratio between the translational movement and the rotational is specified through the call to **setLead(agx::Real)**:

```
// Create a Cylindrical joint (pseudo code)
agx::CylindricalJointRef cylindrical = new agx::CylindricalJoint(...);
// Enable the Screw1D
cylindrical->getScrew1D()->setEnable( true );

// For each revolution, the constrained body will translate 0.1 units.
cylindrical->setLead(0.1);
```

12.11. SplineJoint

A SplineJoint is a 2D constraint. It is related to the Cylindrical joint in that it consists of a hinge and a prismatic. In addition it also contains a hidden kinematic rigid body. This joint will limit translation along a specified spline. It means that two degrees of freedom are left for a constrained body, one rotational and one translational. Basically, the constrained body can translate along the specified spline as well as rotating around it. Both has a motor, range and lock which can be accessed in the same manner as the CylindricalJoint.

```
agxUtil::Spline* spline = new agxUtil::CubicSpline();
spline->add( agxUtil::Spline::Point(agx::Vec3(0,0,0)));
spline->add( agxUtil::Spline::Point(agx::Vec3(0,2,1)));
spline->add( agxUtil::Spline::Point(agx::Vec3(0,4,5)));
spline->updateTangents();

// Specifies a relative transformation between the spline point/tangent and the attached rigidbody
agx::FrameRef frame = new agx::Frame();

// Create the joint
agx::SplineJointRef joint = new agx::SplineJoint(body, frame, spline);
```

12.12. Constraint Forces

It is possible to read forces applied by a constraint. For more details, see 11.20.2.

12.13. Secondary order constraints

Secondary constraints are constraints on constraints such as limits and motors. The Secondary constraint will operate on the DOF which is not constrained. For example, a Hinge allows a body to rotate around one axis. The Secondary constraints of a Hinge will operate on that DOF.

Different constraints have different secondary order constraints. For example, a Hinge has *Lock1D*, *Motor1D* and *Range1D* secondary constraint while LockJoint does not have any secondary constraint at all. The secondary constraints can be requested through a call to the constraint API:

```
agx::Motor1D* motor = hinge->getMotor1D();
agx::Lock1D* lock = hinge->getLock1D();
agx::Range1D* range = hinge->getRange1D();
```

The CylindricalJoint (12.10) has one additional secondary constraint:

```
agx::Screw1D* screw1D = cylindrical->getScrew1D();
```

12.13.1. Motor1D

Motor1D is a controller which operates on the free degree of freedom for a constraint such as the rotational axis of a Hinge, the sliding axis of a Prismatic and or the rotational/sliding axis of a CylindricalJoint. A motor can be speed controlled either by specifying a speed or a force/torque, or both:

```
hinge->getMotor1D()->setSpeed( 1.2 ); // Set the desired speed in radians/sec
// Specify the available amount of torque in negative and positive direction
hinge->getMotor1D()->setForceRange( -1000, 1000 );  // Nm
```

A motor for a rotational constraint such as a hinge has the units of rotational speed and torque, whereas a linear constraint such as a prismatic uses linear distance and force.

A motor is a velocity constraint; it will apply a force/torque to reach the desired speed. This means that a Motor1D will not be able to keep a constrained system stationary over time. It will be subject to a drift in position. To keep a constraint at a specified position, the Lock1D secondary constraint should be used.

The Motor1D however has a utility method called **setLockedAtZeroSpeed()** which will automatically transform the Motor1D to a Lock1D when the desired speed is zero:

```
// Whenever the desired speed is set to 0, the motor1d will operate as a Lock1D
// The same force range will be used for locking the constrained bodies.
hinge->getMotor1D()-setLockedAtZeroSpeed( true );
```

12.13.2. Lock1D

Lock1D can be used to lock a constrained body to a specific position. A specified force/torque range can be given which will limit the max force/torque that the lock can apply to maintain the desired position.

A Lock1D can for example be used to specify a wounded spring:

```
// Specify a position of -100 radians, ie. a wounded up spring.
hinge->getLock1D()->setPosition( -100 );
hinge->getLock1D()->setCompliance( 1E-5 );
```

12.13.3. Range1D

Range1D defines a valid range for a constrained rigid body to move around/along the specified axis. At the limits of a range you effectively get a contact behavior. By default the limits are –inifinity, +infinity.

```
// Enable the range
hinge->getRange1D()->setEnable( true );
// Specify the range limits
hinge->getRange1D()->setRange( agx::RangeReal(-agx::PI, 10*agx::PI) );
```

The above code would limit the rotation of a body between PI and 10 PI radians.

12.13.4. FrictionController - joint internal friction

The FrictionController is valid and available for **Hinge** (rotational), **Prismatic** (translational) and **CylindricalJoint** (rotational and translational), and adds an internal friction force such that:

$$F_{friction} \leq \mu F_{normal}$$

Note that for **Hinge** and rotational **CylindricalJoint** we've a resulting torque. For the friction condition to be dimensionally correct, the axle radius should be included in the friction coefficient, giving:

$$\tau_{friction} \leq \mu r F_{normal}$$

We have stick friction mode when $F_{friction} < \mu F_{normal}$ and the objects are sliding when $F_{friction} = \mu F_{normal}$.

The friction controller is disabled by default.

```
// Enable the friction controller in the hinge.
hinge->getFrictionController()->setEnable( true );
// Assign friction coefficient: Default: 0.4167
hinge->getFrictionController()->setFrictionCoefficient( 0.25 );

// Enable friction controller and set friction coefficient in the cylindrical translational part.
cylindrical->getFrictionController( agx::Constraint2DOF::FIRST )->setEnable( true );
cylindrical->getFrictionController( agx::Constraint2DOF::FIRST )->setFrictionCoefficient( 0.7 );

// Enable friction controller and set friction coefficient in the cylindrical rotational part,
// including axle radius.
agx::Real axleRadius = 0.025;
cylindrical->getFrictionController( agx::Constraint2DOF::SECOND )->setEnable( true );
cylindrical->getFrictionController( agx::Constraint2DOF::SECOND )->setFrictionCoefficient( 0.15 * axleRadius );
```

12.13.4.1. Minimum static friction

The friction controller includes a setting for minimal static friction force, meaning there is a friction restistance even with the friction coefficient set to zero.

```
// Enable the friction controller in the hinge.
hinge->getFrictionController()->setEnable( true );
// Assign minimum static friction torque to 10Nm for both rotational directions: Default: agx.RangeReal(0,0).
hinge->getFrictionController()->setMinimumStaticFrictionForceRange( agx.RangeReal(-10,10 );
```

If you also specify a friction coefficient the friction force may exceed the static friction limit, $F_{friction} = max(\mu F_{normal}, F_{minimumstaticfriction})$.

12.13.4.2. Normal force

The normal force is calculated differently depending on which type of constraint the friction controller is active in. Given the constraint axis is defined to be **along/about z**, the normal force is defined to be translational constraint force along **x** (F^x_{normal}) and **y** (F^y_{normal}), which gives us the normal force:

$$F_{normal} = \sqrt{(F^x_{normal})^2 + (F^y_{normal})^2}$$

12.13.4.3. Linear and non-linear modes

The friction force depends on the normal force and in some non-trivial scenarios the normal force depends on the friction force. A simple and performance wise fast approach is to use the normal force from the previous time step to calculate the maximum friction force in the current time step. This is the default mode and will not significantly affect the performance but may affect the accuracy of the simulation. E.g., the first time step the friction forces will be zero since there's no information regarding previous normal force.

For constraint solve types `agx::Constraint::DIRECT` and `agx::Constraint::DIRECT_AND_ITERATIVE` the friction controller supports non-linear update of the friction forces given current normal force. This mode is similar to direct friction for contacts and will affect the performance but will also increase the accuracy of the simulation. To enable non-linear mode:

```
// By default: constraint->getFrictionController()->getEnableNonLinearDirectSolveUpdate() == false
constraint->getFrictionController()->setEnableNonLinearDirectSolveUpdate( true );
```

12.13.5. Current Position

The secondary constraints has a current angle/position which can be queried with the call to **getAngle()**. For a hinge the call is:

```
// Get the current angle in radians
agx::Real currentAngle = constraint->getAngle();
```

And for a prismatic constraint:

```
// Create a prismatic
agx::Constraint1DOFRef constraint = new agx::Prismatic(...);

// Same call but here we get the current position as it is a Prismatic constraint
agx::Real currentPosition = prismatic->getAngle();
```

Notice that for a hinge, the angle is aware of winding. This means that the whole API for accessing/setting angles can accept values larger than 2*PI.

12.13.6. Current force

The current force applied by a secondary constraint can be accessed by a call to getCurrentForce, notice that the method is the same for both linear and angular constraints.

```
// Get the current torque applied by the hinge
agx::Real currentTorque = hinge->getMotor1D()->getCurrentForce();
```

For more details, see 11.20.2.

12.13.7. Regularization parameters for secondar constraints

The regularization parameters can be used to model the effect of a Secondary constraint. For example, when a hinge constraint reaches a limit, what parameters are then used? Should it be a stiff limit? (Compliance reaches zero) or a softer limit (compliance> 0). Should the limit be fulfilled immediately, (damping=0), or will we allow violation for some time (damping > 0).

```
// Specify how stiff the limit should be:
hinge->getRange1D()->getRegularizationParameters()->setCompliance(1E-5);

// Specify the damping of the limit
hinge->getRange1D()->getRegularizationParameters()->setDamping(3.0/60);
```

12.14. Rebind

The geometry configuration of a constraint can be changed during runtime, the attachment frames can be modified, bodies can be moved relative to each other, however this requires a call to **Constraint::rebind()** which will use the new configuration as the initial configuration. The result is a relaxed system where the bodies are at their current/modified transformation.

Observe that explicitly transforming dynamic objects during a simulation can have unwanted results: instability, missing collision detection, large constraint violations etc.

12.15. Solve type

The solver allows a great flexibility when it comes to configuring constraints. A Constraint can be set to be solved in the iterative/direct/both solvers with the method:

```
Constraint::setSolveType( Constraint::SolveType solveType );
```

Solving a constraint in the iterative solver, means that it will have a lower maximum compliance, i.e. it will be softer/more springy. The direct solver is able to handle very stiff constraints. The benefit of solving constraints in the iterative solver, is commonly it is faster. So depending on your specific configuration of your system, some constraints can be iterative and some can be direct.

It is however important that constraints that *connects* two subsystem where one is solved with the iterative and the other with the direct solver is set to DIRECT_AND_ITERATIVE. This is necessary to transfer the correct forces between the two subsystems. If not, the system can become unstable.

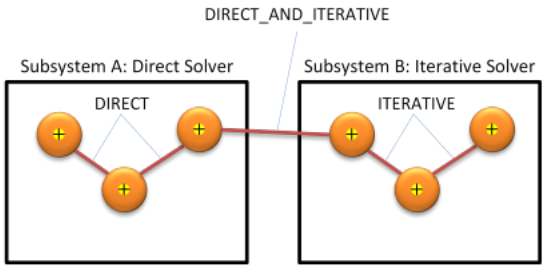


Fig. 12.6 Coupling between subsystems. (Red lines indicate constraints between bodies)

12.16. Constraint utility methods

The class Constraint also has a number of utility methods for creating constraint configurations:

Given two rigid bodies, a point in the world and an axis, the method below will compute the two attachment frames to create a relaxed system:

```
agx::Vec3 worldPoint(0,0,0); // Anchor point for the constraint
agx::Vec3 worldAxis(agx::Vec3::Y_AXIS()); // Axis for the constraint

agx::Constraint::calculateFramesFromWorld(worldPoint, worldAxis, body1, frame1, body2, frame2 );

// Create a hinge with the rotation axis along Y
agx::HingeRef hinge = new agx::Hinge(body1, frame1, body2, frame2);
```

12.17. Virtual Inertia

12.17.1. Basic usage

Virtual inertia and mass can be added to constraints along or about the central constraint axis (N-axis) to add resistance to acceleration on the bodies in the constraints. The dynamics in the constraint axis will then behave as though it has to move an extra mass and/or inertia. This can be used to emulate mechanisms, that are modeled using a single constraint, that have internal inertia.

Both translational mass and rotational inertia can be added separately, affecting the translation and rotational acceleration in the axis accordingly. However, the extra inertia/mass must act through one or more of the RigidBodies active in the constraint. This means that the user must decide how it should be distributed, which can be done in the constructor.

The virtual inertia is applied through an external class. The constructor takes the following arguments:

Argument	Description
constraint	The constraint that the virtual inertia should be active on.
rigidBody1TranslationInertia	The virtual mass in kg that will act on RigidBody 1 along the constraint translational axis.
rigidBody1RotationalInertia	The virtual inertia in kg*m2 that will act on RigidBody 1 about the rotational direction in the constraint axis.
rigidBody2TranslationInertia	The virtual mass in kg that will act on RigidBody 2 along the constraint translational axis.
rigidBody2RotationalInertia	The virtual inertia in kg*m2 that will act on RigidBody 2 about the rotational direction in the constraint axis.

12.17.2. Virtual Inertia in C++

The following code snippet demonstrates how to add virtual inertia to a hinge constraint with two active bodies. This will cause angular acceleration about the central hinge axis to be affected by an extra total inertia of 1 kg*m2. The inertia is added on the second body in the constraint, which means that only that body will feel the extra inertia. The virtual inertia can also be distributed evenly on the bodies, where both will be affected by half of the total inertia.

```
// Create a hinge constraint between two rigid bodies
agx::Frame* frame1 = new agx::Frame(); // empty frame
agx::Frame* frame2 = new agx::Frame(); // empty frame
agx::RigidBody* body1 = new agx::RigidBody(new agxCollide::Geometry(new agxCollide::Box(agx::Vec3(1.0,1.0,1.0))));
agx::RigidBody* body2 = new agx::RigidBody(new agxCollide::Geometry(new agxCollide::Box(agx::Vec3(1.0,1.0,1.0)))) ;
agx::Hinge* hinge = new agx::Hinge(body1, frame1, body2, frame2 );

/**
Add virtual inertia to the constraint to create artificial resistance to acceleration in the constraint axis.
Virtual inertia can be added in the translational and rotational general constraint axis direction.
```

```
The translational virtual inertia will resist acceleration along the constraint N-axis direction (extra mass) and the
rotational virtual inertia will resist rotational acceleration around the constraint N-axis direction.

This example adds rotational inertia to the hinge constraint adding a "virtual resistance" to rotational acceleration
in the hinge axis.
**/
agx::VirtualConstraintInertia* virtualInertia = new agx::VirtualConstraintInertia(hinge, 0, 0, 0, 1.0);

simulation->add( virtualInertia );
```

The following code snippet demonstrates how to add virtual inertia to a prismatic constraint with two active bodies. This will cause a resistance to acceleration along the prismatic axis. The extra mass is added evenly to both bodies.

```
// Create a prismatic constraint between two rigid bodies
agx::Frame* frame1 = new agx::Frame(); //empty frame
agx::Frame* frame2 = new agx::Frame(); //empty frame
agx::RigidBody* body1 = new agx::RigidBody(new agxCollide::Geometry(new agxCollide::Box(agx::Vec3(1.0,1.0,1.0))));
agx::RigidBody* body2 = new agx::RigidBody(new agxCollide::Geometry(new agxCollide::Box(agx::Vec3(1.0,1.0,1.0))));
agx::Prismatic* prismatic = new agx::Prismatic(body1, frame1, body2, frame2 );

/**
Add virtual inertia on both bodies in the translational direction (N-axis) of the constraint.
**/
agx::VirtualConstraintInertia* virtualInertia = new agx::VirtualConstraintInertia(prismatic, 0.5, 0, 0.5, 0);

simulation->add( virtualInertia );
```

13. Particle systems

See also GranularBodySystem

13.1. Basic usage

There are two types of particle systems available in AGX: RigidParticleSystem and ConstraintFluid. The RigidParticleSystem simulates a collection of hard spheres that are held apart by contact constraint solved by a Gauss-Seidel solver. The ConstraintFluid simulates a fluid where each particle samples a density field and the particles are held apart by density constraints solved by a conjugate-gradient solver. There are two-way interactions between the rigid particles and other parts of the simulation, but the constraint fluid currently only supports static geometries in the scene.

A particle system can be created either from C++ code or from a Python scene script, but the setup procedure is very similar. Examples in code are given below for everything that is described in the following in both C++ and Python code.

The first step is to create an instance of the particle system class in question, the RigidParticleSystem class for the hard sphere particles and ConstraintFluid for the density constrained fluid. The second step is to create a set of particles to simulate. They can either be allocated and initialized using the particle API, or created in a regular grid using the spawnParticlesInBound method available in the ParticleSystem base class.

Regardless of how the particles were created, individual particles can be accessed and manipulated using ParticlePtrs. A ParticlePtr is an accessor object that gives access to the individual attributes of a particle, stored as elements in the buffers that make up the EntityStorage that the particle in question is part of. The ParticlePtr exposes one accessor method for each attribute of the Particle entity.

- position
- velocity
- force
- life

13.1.1. Particle systems in C++

The following code snippet demonstrates how to create a simple rigid particle system in C++.

```
// Create and configure a rigid particle system.
ParticleSystemRef particleSystem = new RigidParticleSystem();
simulation->add( particleSystem );
particleSystem->setParticleMass( Real(10.0) );
```

```
agx::Physics::ParticlePtr particle = particleSystem->createParticle();
particle.position() = Vec3(0, 0, 4.5);
```

```
// Create a bunch of particles in a bound.
agx::Bound3 spawnBound = agx::Bound3( Vec3(0.0, 0.0, 0), agx::Vec3(4.0, 1.0, 4.0) );
particleSystem->spawnParticlesInBound( spawnBound, agx::Vec3f(0.1) );
```

13.1.2. Particle systems in Python

The following code snippet demonstrates how to create a simple rigid particle system in Python.

```
# Create and configure a particle system.
particle_system = agx.RigidParticleSystem()
sim.add( particle_system )
particle_system.setParticleMass( 10 )
```

```
# Create a single particle.
particle = particle_system.createParticle()
particle.position().set( agx.Vec3(0, 0, 4.5) )
```

```
# Create a bunch of particles in a bound.
spawn_bound = agx.Bound3( agx.Vec3(0, 0, 0), agx.Vec3(4,1,4) )
particle_system.spawnParticlesInBound( spawn_bound, agx.Vec3(0.1) )
```

13.1.3. Particle emitter

The particle system comes with a collection of tasks designed to be part of a particle system update. One such task is the particle emitter, which creates new particles at a certain rate within a specified spawn volume. This task has an explicit Python binding and can therefore be created directly from Python code. The volume in which particles may be created is defined by a geometry and a method is used to set the volume.

```
emitter = agx.ParticleEmitter( )
emitter.setGeometry( agxCollide.Geometry( agxCollide.Cylinder(1.6, 10) ) )
simulation.add( emitter );
```

The emitter contains a parameter to control the rate of the particle creation, in particles per second. This parameter can be set from Python using the following code snippet:

```
emitter.setRate( 15 )
```

To set the initial “life” of an emitted particle:

```
emitter.setLife( 10 );
```

14. AGX Granular

14.1. Introduction

`agx::Physics::GranularBodySystem` is a extension of the `agx::ParticleSystem` class that contains spherical particles, called granular bodies, with 6 degrees of freedom (6-DOF) with Hertzian contact elasticity laws, Coulomb friction and rolling resistance. AGX Granular is based on a non-smooth dynamics formulation of the discrete element method (DEM) called non-smooth discrete element method (NDEM) [2] [1]. The granular system is solved together with the rest of AGX Dynamics which allows for a strong-coupling between articulated rigid body systems and granular bodies.

Note

In this chapter, the term `particle` and `granular body` (or just `body`) will be used interchangeably. They will all refer to the 6-DOF particle instance mentioned in the introduction above. A `GranularBodyPtr` is an extension of a `ParticlePtr` with additional buffers for inertia, rotation and angular velocity.

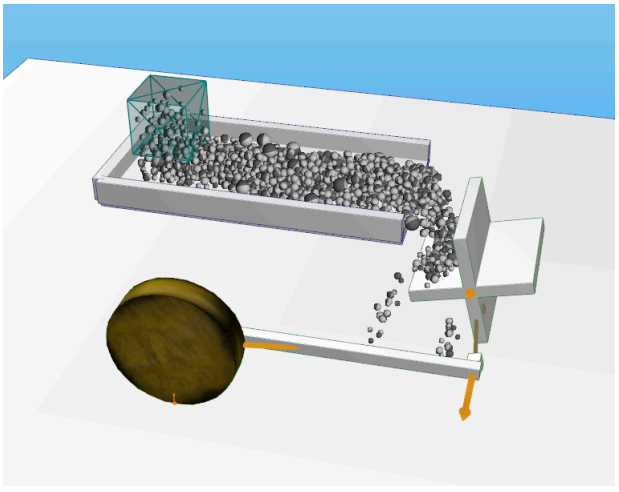


Figure 1: Example showcasing a granular body flow interacting with and driving a mechanical system of constrained rigid bodies.

A contact situation is illustrated in Fig 1. Contact forces and velocities can be decomposed in the direction of the contact normal $\vec{n}$ and tangents $\vec{t}$. The gap function $g(\boldsymbol{x})$ measures the magnitude of the overlap between the contacting bodies.

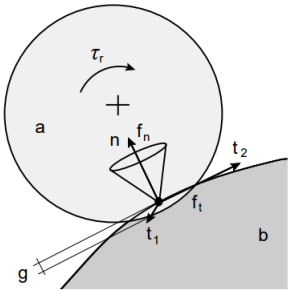


Figure 2: Illustration of a contact between two bodies a and b giving rise to a normal force f_n, tangential force f_t and rolling resistance τ_r. The contact overlap is denoted by g, the unit normal n and tangents t_1 and t_2.

`agx::Physics::GranularBodySystem` is used to simulate the movement of dynamic soil in agxTerrain.

14.1.1. Feature Overview

The `GranularBodySystem` contains memory-aligned attribute storages (position, velocity, inertia, etc…) for increased efficiency and performance. Granular bodies have two-way interactions with all the other physical systems in AGX Dynamics, such as articulated multibody systems. GranularBody particles can be created from the system object or by using particle emitter. Particle distribution tables containing particle models with different radii and material properties can be used in the creation of particles. The large contact system that often results from granular body interaction can be solved using a parallel solver (PPGS) for increased performance where a warm starting option exist for increased solution fidelity. Particle states can also be written and read from files via the GranularReaderWriter class. The granular API is accessible via Python.

14.2. Basic Usage

14.2.1. Setup and Creation

The following code snippet demonstrates how to create a `agx::Physics::GranularBodySystem` system in C++:

```
// Create and configure a granular body system
agx::Physics::GranularBodySystemRef granularBodySystem = new agx::Physics::GranularBodySystem();
simulation->add( granularBodySystem );
```

Default values for created bodies can be set in the `GranularBodySystem` object:

```
// Properties set on the granularBodySystem are the default values of
// all created particles. These can of course be changed on an individual
// particle basis after creation.

// Most of the methods have optional arguments for updating existing particles
bool updateExistingParticles = false;

// Configure default values for created particles.
// When the radius is updated on particles, the mass is automatically calculated from
// the volume change.
granularBodySystem->setParticleRadius( 0.1, updateExistingParticles );

// Set the default agx::Material that governs body density and contact material.
// When the material is updated on particles, the mass is automatically calculated from
// the density in material->getBulkMaterial()->getDensity()
granularBodySystem->setMaterial( particleMaterial, updateExistingParticles );
```

Individial bodies can be created via the `GranularBodySystem` object. The body initially has default values but they can be changed after creation:

```
// Creation returns a agx::Physics::GranularBodyPtr representing the created particle
agx::Physics::GranularBodyPtr body = granularBodySystem->createParticle();

// Set position of the particle
body->setPosition( Vec3(0, 0, 4.5) );

// Set radius of particle ( updates mass )
body->setRadius( 0.1 );

// Set lifetime of particle in seconds
body->setLife( 2.0 );

// Set agx::Material of particle ( updates mass )
body->setMaterial( particleMaterial->getEntity() );
```

We can also create batches of bodies using bounds specified by the user:

```
// Create a bunch of particles in a bound.
agx::Bound3 spawnBound = agx::Bound3( Vec3(0.0, 0.0, 0), agx::Vec3(4.0, 1.0, 4.0) );

// The radius of all the created particles
agx::Real particleRadius = 0.1;

// Spacing between the created particles in the bound.
// Note that a spacing of 2 x radius will give a tight packing as
// particles will be placed next to eachother
agx::Vec3 spacing = agx::Vec3( radius * 2, radius * 2, radius * 2 );

// Add a random jitter on the placement of the particles. The jitterFactor is defined as
// the fraction of the specified particle spacing which will be used as the bound for
// the random perturbation of the particle placement.
agx::Real jitterFactor = 0.1;

// Place particles in an evenly spaced lattice 3D square lattice ( with jiter )
granularBodySystem->spawnParticlesInBound( bound, radius, spacing, jitterFactor );

// Place particles in a Hexagonal-Close-Packing (HCP) lattice
granularBodySystem->spawnParticlesInBoundHCP( bound, radius, spacing, jitterFactor );
```

Unless specified in the function call, the particles spawned carry default properties set in the `agx:::Physics::GranularBodySystem` object.

Deleting a granular body can be done in multiple ways, mainly via utiliy functions on the `GranularBodySystem` object:

```
// This will tag the particle for removal at the end of the timestep
// ensuring that it will be destroyed in a safe way.
// This can be called saftely any time during the timestep and is the
// recommended way of removing particles.
granularBodySystem->removeParticle( particle )

// You can check if a particle has been tagged for removed by checking it's state variable
// Option 1: via the particle instance
particle.state().removed()
// Option 2: via the granular system ( recommended ). Also checks if the particle instance
//           has been destroyed ( See below )
granularBodySystem->isValid( particle )

// This function immmediately destroys the particle instance.
// NOT recommended to do this in the middle of a simulation step since it
// will change the internal structure that can invalidate the simulation.
// ONLY do this after the simulation step in post() or last()
particle->destroy();
```

14.2.2. Motion Control

GranularBodies have a `MotionControl` state similar to RigidBodies. This can be set on bodies via a utility function on the `GranularBodySystem` object.

```
// Granular bodies support two MotionControls:
// 1. DYNAMICS - "Affected by forces, position/velocity is integrated by the *DynamicsSystem*."
// 2. KINEMATICS - "Not affected by forces, considered to have infinite mass. Velocities can be set by user."
// Make particle kinematic.
granularBodySystem->setMotionControl( particle, agx::Physics::GranularBodySystem::MotionControl::KINEMATICS )

// Particle will now effectivley have infinite mass. We now assign velocities to it.
particle->setVelocity( agx::Vec3( 1, 0, 0 ) ) // Set fixed translational speed in x-direction
particle->setAngularVelocity( agx::Vec3( 0, 0, agx::PI_2 ) ) // Rotational speed of 90 deg / s around z-axis

// Returns the motion control of the particle.
agx::Physics::GranularBodySystem::MotionControl particleMotionControl = granularBodySystem->getMotionControl( particle )
```

Note

In granular contacts where both granular bodies have the KINEMATICS motion control, the contact forces will not be calculated. This can be used to “freeze” areas of the simulation where particles are in rest in order to increase performance.

14.2.3. Access Granular Data

Regardless of how the granular bodies were created, individual particles can be accessed and manipulated using `agx::Physics::GranularBodyPtrs`; the same convention as with `agx::ParticleSystem`. A `agx::Physics::GranularBodyPtr` is an accessor object that gives access to the individual attributes of a granular body, stored as elements in the buffers that make up the `EntityStorage` that the particle in question is part of. The `GranularBodyPtr` exposes one method of access for each attribute of the GranularBody entities.

Table 14.1 Accessing Granular data

Position	`getPosition/setPosition`	Gets/Sets the position (`agx::Vec3`) of the body.
Rotation	`getRotation/setRotation`	Gets/Sets the rotation (`agx::Quat`) of the body.
Velocity	`getVelocity/setVelocity`	Gets/Sets the velocity (`agx::Vec3`) of the body.
Angular Velocity	`getAngularVelocity/setAngularVelocity`	Gets/Sets the angular velocity (`agx::Vec3`) of the body.
Force	`getForce/setForce`	Gets/Sets the force (`agx::Vec3`) acting on the body. Useful for using explicit force models on the bodies.
Mass	`getMass/setMass`	Gets/Sets the mass of the body. See note about automatic calculation below!
Inertia	`getInertia/setInertia`	Gets/Sets the inertia of the body. See note about automatic calculation below!

Life	`getLife/setLife`	Gets/Sets the lifetime of the particle in seconds.
Color	`getColor/setColor`	Gets/Sets the `agxRender::Color (agx::Vec4f)` of particle which is used in rendering.
Material	`getMaterial/setMaterial`	Gets/Sets the `agx::Material` of the particles which is used for determining contact material and in calculation of mass and density.

Note

Note that *inertia* and *mass* is automatically calculated when a `agx::Material` is assigned via the `setMaterial` function to the particle. If you want to manually edit these, be sure to do it after the `agx::Material` assignment.

```
agx::Real averageMass = 0.0;
agx::Real averageRadius = 0.0;
agx::MaterialRef newMaterial = new agx::Material( "newParticleMaterial" );
newMaterial->getBulkMaterial()->setDensiy( 1700 ); // kg/m3
granularBodies = granularBodySystem->getParticles();
for ( agx::Physics::GranularBodyPtr gPtr : granularBodies )
{
  gPtr->setVelocity( agx::Vec3( 0, 0, 0 ) );
  gPtr->setMaterial( newMaterial ); // This function updates the body mass from BulkMaterial density
  gPtr->setRadius( 0.05 ); // This function also updates the body mass
  averageMass += gPtr->getMass(); // extract the calculated mass
  averageRadius += gPtr->getRadius();
}

averageMass /= granularBodySystem->getNumParticles();
averageRadius /= granularBodySystem->getNumParticles();
```

14.3. Material Configuration and Parametrization

Physical properties and dynamic bulk behavior of the granular bodies are governed by the material settings and the contact material in the granular contacts. This section will cover the most important aspects of material parametrization to affect granular bulk behavior.

Particle density/mass is primary derived from the set `agx::Material` of the particle but can also be overridden manually:

```
// Set material density
agx::MaterialRef particleMaterial = new agx::Material( "particleMaterial" );
particleMaterial->getBulkMaterial()->setDensiy( 1700 ); // kg/m3

// Set the material as the default of granular bodies created from this system
// ( unless specified otherwise, e.g from a distribution table in an emitter )
granularBodysystem->setMaterial( particleMaterial );

// Manually create a particle and apply material to it
granularBodyPtr = granularBodysystem->createParticle();
granularBodyPtr->setMaterial( particleMaterial );

// Can also override this and apply a fixed mass. Not really recommended since
// material and radius assginment overries manually set mass values.
granularBodyPtr->setMass( 10 ); // kg

// Radius of the particle determines volume and thus resulting mass
granularBodyPtr->setRadius( 0.05 ); // m
```

In AGX Dynamics the constitutive equations in granular contacts follows Hertz contact law with Coulomb friction and rolling resistance [2]. Contact material properties determines coefficients used in the contact laws where there are some unique parameters that are only active in granular contacts. The example below shows how to setup a contact material between granular bodies:

```
// Create new material that will be set to particle
agx::MaterialRef particleMaterial = new agx::Material( "particleMaterial" );
simulation->add( particleMaterial );
auto particleParticleCM = simulation->getMaterialManager()->getOrCreateContactMaterial( particleMaterial,
particleMaterial );

// Apply the particle material to the granular body system so that it becomes the
```

```
// default material for new granular bodies
granularBodySystem->setMaterial( particleMaterial );

// Set the Young's Modulus of the contact material. In granular contacts, this corresponds
// direcly to Young's Modulus in Hertz contact law.
particleParticleCM->setYoungsModulus( 1e8 );

// The coulomb friction coefficient in the granular contacts
particleParticleCM->setFrictionCoefficient( 0.3 );

// The rolling resistance coefficients in the tangential direction of a contact
particleParticleCM->setRollingResistanceCoefficient( 0.3 );

// The twisting resistance coefficient in the normal direction of a contact
particleParticleCM->setTwistingResistanceCoefficient(0.001);

// The adhesive effects of the granular contacts. Determined by a max force in the adhesive
// contact as well as the depth overlap where the adhesive force is active
// Arguments are: max adhesive force ( N ), adhesive overlap ( m )
particleParticleCM->setAdhesion( 10, 0.05 );

// Set the restitution of the granular contact
particleParticleCM->setRestitution( 0.0 );

// Set the numerical damping of the contact
// In high timestep, it is usually some coefficient multiplied by the time step
particleParticleCM->setDamping( timeStep * 2 );
```

Note

The rolling and twisting resistance coefficients of the agx::ContactMaterial is unique to the `granular <-> granular` and `granular <-> geometry` contacts. They are not used in regular geometry and rigid body contacts.

14.4. Collision Groups

Collision groups can be set to the granular body system in order to configure collision settings in between particles and other geometries. These collision groups will be applied to all particles upon creation with optional argument for applying all existing instances as well.

```
// Optional argument for updating existing particles with the group
bool updateExistingParticles = false;

// Add collision group to system
granularBodySystem->addCollisionGroup( "group1", updateExistingParticles );

// Remove collision group from system
granularBodySystem->removeCollisionGroup( "group1", updateExistingParticles );

// Check if system has collision group ( expecting false )
granularBodySystem->hasCollisionGroup( "group1" );
```

Groups can be set on individual particles using the `agx::Physics::GranularBodySystem` interface:

```
// Add collision group to a single particle
agx::Physics::GranularBodyPtr ptr = granularBodySystem->createParticle()
granularBodySystem->addCollisionGroupParticle( "group1", ptr );

// Remove collision group from GranularBody
granularBodySystem->removeCollisionGroupParticle( "group1", ptr );

// Check if particle has collision group ( expecting false )
granularBodySystem->hasCollisionGroupParticle( "group1", ptr );
```

You can also disable collision between particles and specific geometries:

```
// geometry argument is a agxCollide::Geometry
```

```
granularBodySystem->setEnableCollisions( geometry, false );
```

14.5. Particle Distribution Tables

Particle distribution tables are used to represent distributions of particle types with different properties such as radii and material. The `agx::EmitterDistributionTable` is a base class in the `agx::Emitter` class and can be used for particle distributions in Particle emitters or for rigid body distributions in RigidBody emitters.

Distribution tables contains a set of different particle distribution models. A distribution model consists of a `radius`, an `agx::Material` and a `weight`. The weight is the probability to create a particle of the corresponding model when the table is queried of a random model.

The user specifies a `agx::Emitter::Quantity` when a distributions table is created. This quantity is the basis of the probability of how particle models should be selected with respect to their weight. If a model has a higher weight relative to the other models in a table it will contribute more to the total amount of created quantity (number of particles, mass or volume) from that table.

Table 14.2 Different quantity settings of the distribution table

QUANTITY_COUNT	Counting number of individual particles as the quantity.
QUANTITY_VOLUME	Counting the total volume of emitted particles as the quantity.
QUANTITY_MASS	Counting the total mass of the emitted particles as the quantity.

Example:

The quantity is set to `QUANTITY_MASS` and the table contains two models, one that has weight 1 and the other weight 2. Then the probability of particle creation is set so that the second model will contribute two times more *mass* than the first model to the total mass that is created from the table. If `QUANTITY_COUNT` is set, then the second model will yield twice as many *discrete elements* as the first one of the total amount of particles created from the table.

```
// Create a new distribution table with specified quantity
agx::Emitter::DistributionTableRef table = new agx::Emitter::DistributionTable( agx::Emitter::QUANTITY_MASS )

// Arguments to a particle model is: radius, agx::Material, weight
table->addModel( new agx::ParticleEmitter::DistributionModel( 0.05, pelletmat, 0.3 ) )
table->addModel( new agx::ParticleEmitter::DistributionModel( 0.09, pelletmat, 0.6 ) )
table->addModel( new agx::ParticleEmitter::DistributionModel( 0.15, pelletmat, 0.1 ) )
```

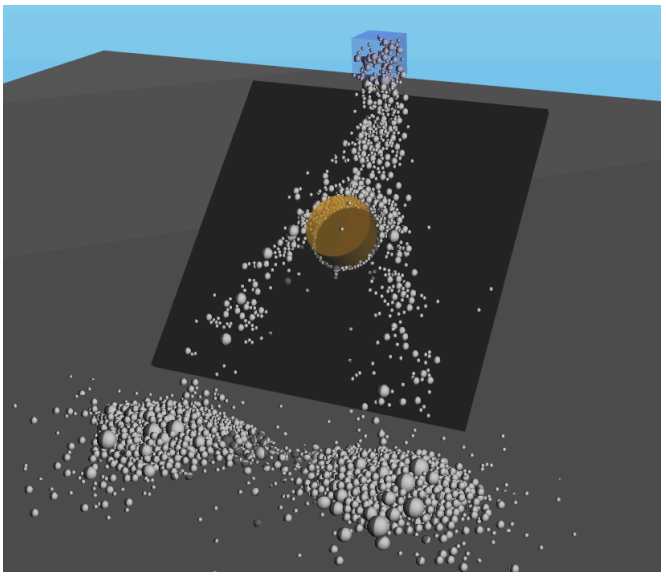
Random models can be queried from the table using the following call:

```
// Query the table for a random model
DistributionModel* model = distributionTable->getRandomModel();

// Since this will return an abstract model ( particle and rigid body models are subclasses ),
// we need to cast it to a particle model
agx::ParticleEmitter::DistributionModel* particleModel = model->as< agx::ParticleEmitter::DistributionModel >();
```

Distribution tables are used both in particle emitters and when creating batches of particles.

14.6. Particle Emitter



The `agx::ParticleEmitter` is used to continuously create a stream of particles in a predefined volume defined by a `agxCollide::Geometry`. The created particles carry the default properties of the coupled granularBodySystem, with the exception of `color` and `life` which can be set individually per emitter.

Note

In order to create emitters that emits bodies/particles with different materials, see Distribution tables

```
// When a particle emitter is created, it needs to be coupled to a existing ParticleSystem/granularBodySystem.
agx::ParticleEmitterRef emitter = agx::ParticleEmitter( granularBodySystem );

// Set the geometry that will server as a spawning volume for the particles
emitter->setGeometry( agxCollide::Geometry( agxCollide::Cylinder( 1.6, 10 ) ) );

// Add the emitter to the simulation to connect it's update task to the simulation
simulation->add( emitter );
```

Note

The emitter `agxCollide::Geometry` acts just like any other geometry. You can add it to a body in order to create a dynamically moving emitter.

A `Quantity` needs to be assigned to an emitter to determine the nature of the creation, similar in creation of particle distribution tables. This determines if the given creation rate is based on discrete number of particles, mass or volume.

```
emitter->setQuantity( agx::Emitter::Quantity::COUNT );
```

The emitter contains a parameter to control the rate of the particle creation which is dependent on current set quantity of the emitter.

```
emitter->setRate( 15 );
```

The maximum amount of quantity (num particles, mass or volume) that can be emitted can be adjusted:

```
/**
What the set amount means depends on the current Quantity setting of the emitter.
QUANTITY_COUNT  - max number of objects
QUANTITY_VOLUME - max total volume of objects
QUANTITY_MASS   - max total mass of objects
*/
emitter->setMaximumEmittedQuantity( 15 );
```

You can get information about emitted quantity and the number of particles emitted:

```
/**
* Returns the quantity amount of that this emitter has emitted.
*/
emitter->getEmittedQuantity() const;
```

```
/**
\return the number of bodies that this emitter has emitted
*/
emitter->getNumEmitted() const;
```

Distribution tables can be used in emitters by a simple assignment:

```
emitter->setDistributionTable( particle_table );
```

It will then query the table for a particle model when creating new elements instead of using the default settings in the coupled `GranularBodySystem`.

Collision Groups of emitted particles can also be adjusted on an emitter basis. These groups are added on top of the groups that has been added via the granular body system object:

```
emitter->addCollisionGroup( "emitter_group" );
emitter->removeCollisionGroup( "emitter_group" );
bool hasGroup = emitter ->hasCollisionGroup( "emitter_group" );
```

The `life` parameter of particles can also be set on an emitter basis.

```
emitter->setLife( 10 );
```

Initial velocity of created particles can also be set via the emitter:

```
emitter->setVelocity( agx::Vec3( 1, 1, 1 ) );
```

14.7. Granular Bodies in Python

The functionality around `GranularBodySystem` and particle emitters can be used via the Python API.

Note

Due to the SWIG mapping the subclass namespace structure in C++ is compressed in the resulting Python API. This primary affects the distribution table, particle model and the `GranularBodySystem` classes.

Table 14.3 Table showing the SWIG mapping of the Granular C++ API to Python API

C++ API	Python API
`agx::Physics::GranularBodySystem`	`agx.GranularBodySystem`
`agx::Emitter::DistributionTable`	`agx.EmitterDistributionTable`
`agx::ParticleEmitter::DistributionModel`	`agx.ParticleEmitterDistributionModel`

The following code snippet demonstrates how to create a simple granular body system in Python and some example applications for creating bodies.

```
# Create and configure a particle system.
granular_body_system = agx.GranularBodySystem()
sim.add( granular_body_system )
granular_body_system.setMaterial( granularMaterial )
```

```
# Create a single particle.
gBodyPtr = granular_body_system.createParticle()
gBodyPtr.setPosition( agx.Vec3( 0, 0, 4.5 ) )
```

```
# Create a bunch of particles in a bound.
particle_radius = 0.1
spawn_bound = agx.Bound3( agx.Vec3(0, 0, 0), agx.Vec3(4,1,4) )
granular_body_system.spawnParticlesInBound( spawn_bound, particle_radius, agx.Vec3( particle_radius * 2 ) )
```

Sample code for setting up a `DistributionTable`:

```
# Create a new distribution table with specified quantity
table = agx.EmitterDistributionTable( agx.Emitter.QUANTITY_MASS )

# Arguments to a particle model is: radius, agx::Material, weight
table.addModel(agx.ParticleEmitterDistributionModel( 0.05, pelletmat, 0.3 ) )
table.addModel(agx.ParticleEmitterDistributionModel( 0.09, pelletmat, 0.6 ) )
table.addModel(agx.ParticleEmitterDistributionModel( 0.15, pelletmat, 0.1 ) )
```

Emitters can also be created in Python:

```
# init emitter with mass quantity setting
emitter = agx.ParticleEmitter( granularBodySystem, agx.Emitter.QUANTITY_MASS )

# Set the previously declared distribution table to the emitter.
emitter.setDistributionTable( table )
emitter.setRate( 15 ) # 15 kg/s
emitter.setMaximumEmittedQuantity( 100 ) # 100kg

# Create spawn geometry for the emitter. The particles will be created inside the bound of the coupled geometry.
emitterSize = agx.Vec3( 0.5, 0.5, 0.5 )
emitterGeom = agxCollide.Geometry( agxCollide.Box( emitterSize ) )
emitterGeom.setPosition( g_emitterSpecification[ "position" ] )

# Disable collisions in emitter to remove contacts with the particles created in the volume.
emitterGeom.setEnableCollisions( False )
sim.add( emitterGeom )
emitter.setGeometry( emitterGeom )

# Set an initial velocity downward
emitter.setVelocity( agx.Vec3( 0 , 0, -1.0 ) )
sim.add( emitter )
```

14.8. Extract contact data in Python

Contact data involving granular bodies can be extracted via `agxCollide::Space`. The following snippet showcases how to extract and read:

Warning

Contact normals and tangents are in `agx.Vec3f` format. They need to be converted to `agx.Vec3` in order to be used in multiplication with other data in the `agx.Vec3` format that is returned from functions like `getPosition()` and `getVelocity()`.

```
# Extract particle <-> particle contacts
particle_particle_contacts = sim.getSpace().getParticleParticleContacts()
for c in particle_particle_contacts:
    # Get particle ids from the contact
    id_particle1 = c.getParticleId1()
    id_particle2 = c.getParticleId2()

    # Get ParticlePtr/GranularBodyPtr from ids by using the GranularBodySystem
    p1 = granularBodySystem.getParticle(id_particle1)
    p2 = granularBodySystem.getParticle(id_particle2)

    # Get normals and tangents. These are all in agx.Vec3f format
    # they need to be converted to agx.Vec3 in order to be multiplied with
    # agx.Vec3 from methods like 'getVelocity' or 'getPosition'
    normal = c.getNormal()
    tu = c.getTangentU()
    tv = c.getTangentV()

    # Extract contact force. Note that this will be zero if read in the
    # 'pre-step' phase of the simulation. Forces are available in the
    # 'post-step' phase of the simulation which is executed after the solver.
    f = c.getLocalForce()

    # Get contact point and depth
```

```
    point = c.getPoint()
    depth = c.getDepth()

# Extract particle <-> geometry contacts
particle_geometry_contacts = sim.getSpace().getParticleGeometryContacts()
for c in particle_geometry_contacts:
    # Get particle id in the contact
    id_particle = c.getParticleId()

    # Get ParticlePtr/GranularBodyPtr from id by using the GranularBodySystem
    p = granularBodySystem.getParticle(id_particle)

    # Get the geometry from the contact and then it's rigid body, if it exists
    id_geometry = c.getGeometryId()
    geom = sim.getSpace().getGeometr(id_geometry)
    # Will return None if geometry has no rigid body
    rigidBody = geom.getRigidBody()

    # Get normals and tangents. These are all in agx.Vec3f format
    # they need to be converted to agx.Vec3 in order to be multiplied with
    # agx.Vec3 from methods like 'getVelocity' or 'getPosition'
    normal = c.getNormal()
    tangentU = c.getTangentU()
    tangentV = c.getTangentV()

    # Extract contact force. Note that this will be zero if read in the
    # 'pre-step' phase of the simulation. Forces are available in the
    # 'post-step' phase of the simulation which is executed after the solver.
    f = c.getLocalForce()

    # Get contact point and depth
    point = c.getPoint()
    depth = c.getDepth()
```

Existsing Python tutorials goes further in showcasing `GranularBodySystem` functionality.

14.9. Solver Settings

Solver settings affect the performance and fidelity of `GranularBodySystem`. Every granular contact is solved using an iterative solver configuration; The typical large contact network that results from big granular systems quickly become unfeasible for a direct solver.

Note

`GranularBodySystem particle <-> particle` and `particle <-> geometry contacts` are **ALWAYS** solved using an iterative solver for all the 6 contact equations *(1 x normal, 2 x friction, 2 x tangential rolling resistance and 1 x twisting resistance)*. Contact material `Friction model` settings have no effect on this.

A basic setup of the solver can look like this:

```
// Set maximum amount of threads for usage in the parallel solver
// We prefferable want to use a number of threads equal to number of physical
// cores present in the machine. So account for potential HT/SMT.
agx::setNumThreads( 0 );
int n = agx::getNumThreads() / 2; // Account for hyper-threading
agx::setNumThreads( n );

// Apply simulation solver parameters.
// Use the parallel PGS solver for best performance using multiple threads.
sim->getDynamicsSystem()->getSolver()->setUseParallelPgs( true )

// Enable 32bit float solver for granular bodies to increase performance.
sim.getDynamicsSystem()->getSolver()->setUse32bitGranularBodySolver( true );

// Enable granular warm starting for all solver configurations to enable better solutions.
sim->getDynamicsSystem()->getSolver()->setUseGranularWarmStarting( true );

// Set number of iterations that should be used in the Parallel Gauss-Seidel Solver.
// The higher number the better solution, but will also impact performance.
```

```
sim->getDynamicsSystem()->getSolver()->setNumPPGSRestingIterations( 50 );

// Set the number of resting iterations in the regular serial iterative solver.
// Higher iterations result in better solutions with increaded computational cost.
sim->getDynamicsSystem()->getSolver()->setNumRestingIterations( 50 );
```

14.9.1. Parallel Projected Gauss Seidel (PPGS)

The contact system can be solved by a Parallel-Projected Gauss-Seidel Solver (PPGS). The solver first partitions the system into different subdomains and interaction zones that are solved in parallel via multithreading. The iteration count used in the solver can be adjusted according to fidelity needs.

Solver settings can be adjusted to increase or decrease simulation fidelity with respect to computational performance.

```
// Use the parallel PGS solver for best performance using multiple threads
sim->getDynamicsSystem()->getSolver()->setUseParallelPgs( true )

// Set number of iterations that should be used in the Parallel Gauss-Seidel Solver.
// The higher number the better solution, but will also impact performance.
sim->getDynamicsSystem()->getSolver()->setNumPPGSRestingIterations( 50 )
```

Note

PPGS can be used for rigid body contacts as well. When enabled every rigid body contact with a contact material that has the `agx::IterativeProjectedConeFriction( agx::FrictionModel::ITERATIVE)` as set `Friction Model` will be included in the PPGS solver.

14.9.2. Warm Starting Granular Contacts

The PPGS solver can be warm-started by using the constraint multiplier lambda (λ) from the solution of the contact constraint equations in the previous time-step. In general, this will enable contacts that persists over time to converge faster and result in a more precise solution of PPGS solver with fewer number of iterations. This option is recommended when simulating slowly evolving systems with contact networks. For rapidly evolving systems, there is little benefit from using warm-starting or even risk that it worsen the convergence.

```
// Enable granular warm starting for all solver configurations to enable better solutions
sim->getDynamicsSystem()->getSolver()->setUseGranularWarmStarting( true )
```

14.9.3. 32bit Granular Solver

The iterative solvers that are used when solving the granular contact networks always has some error. The order of magnitude of the error is in all cases higher than the precision of 64it floats. Due to this fact there exists an option of only using 32bit floats in the granular solver to increase compute performance by up to 30-40%.

```
// Enable granular warm starting for all solver configurations to enable better solutions
sim->getDynamicsSystem()->getSolver()->setUseGranularWarmStarting( true )
```

14.10. GranularReaderWriter

`GranularBodySystem` and particle data can be serialized and stored separately apart from the regular serialization functionality of AGX Dynamics using the `agx::GranularReaderWriter` class. This can be used to store granular data in `.agx`, `.aagx` and `.csv`.

```
// Extension of filename determines the file format that is stored ( .agx, .aagx or .csv )
std::string filename = "particles.agx";

// Vector of GranularBodyPtr:s, can be individually picked. Can also store every body by
// extracting them via the getParticles() function
agx::Physics::GranularBodyPtrVector bodies = granularBodySystem.getParticles();

// Optional argument transform is applied to the position and velocity
// of the bodies before storing to file.
// Create simple translation matrix in this case
agx::AffineMatrix4x4 transform = agx::AffineMatrix4x4::translate( 0, 1, 2 );

// Write all the existing bodies to an .agx file
agx::GranularReaderWriter::writeFile( filename,
                                      bodies,
```

```
                                    granularBodySystem,
                                    transform );
```

We can then load stored bodies from an `.agx` file:

```
// Specify the .agx file to load bodies from
std::string filename = "particles.agx";

// Vector of GranularBodyPtr:s where the loaded bodies will be inserted
agx::Physics::GranularBodyPtrVector granularBodies;

// Optional argument transform is applied to the position and velocity
// of the bodies before loading into the simulation.
// In this case, we want to invert the previous translation matrix to bring the bodies back to the origin frame.
agx::AffineMatrix4x4 transform = agx::AffineMatrix4x4::translate( 0, 1, 2 ).inverse();

// Load the bodies from the specified filename
bool success =  loadFileAGX( filename,
                             granularBodies, // loaded bodies will be inserted into this vector
                             granularBodySystem, // loaded bodies will be inserted into this granular body system
                             transform );
```

References

[1]

3. Lacoursière, Regularized, stabilized, variational methods for multibodies, in: D.F. Peter Bunus, C. Führer (Eds.), The 48th Scandinavian Conference on Simulation and Modeling (SIMS 2007), Linköping University Electronic Press 2007, pp. 40–48

[2] (1,2)

Wang, D. (2015). Accelerated granular matter simulation (PhD dissertation). Umeå. pdf.

14.11. RigidBody emitter

An **agx::RigidBodyEmitter** creates new rigid bodies at a certain rate within a specified volume and adds them to the simulation. This is useful when you do not want to create all the rigid bodies at the start of the simulation and/or when the rigid bodies in your simulation should follow some kind of size/shape distribution.

The following code snippet demonstrates how to create a rigid body emitter in c++:

```
agx::RigidBodyEmitterRef emitter = new agx::RigidBodyEmitter();
emitter->setRate(30.0);
emitter->setMaximumEmittedQuantity(200);
emitter->setQuantity(agx::Emitter::QUANTITY_COUNT);
```

The rate and the maximumEmittedQuantity decides how fast the emitter creates new bodies and what the maximum quantity is. There are three different ways of counting the quantity of the emitter.

Table 14.4 Different quantity settings of the emitter and distribution table

QUANTITY_COUNT	Counting number of individual rocks as the quantity.
QUANTITY_VOLUME	Counting the total volume of emitted rocks as the quantity.
QUANTITY_MASS	Counting the total mass of the emitted rocks as the quantity.

When the emitter creates a new rigid body it randomly chooses a distributionModel from the distributionTable. A distributionModel consists of a rigidBody template and a weight. The weight is the probability to create a clone of the corresponding rigid body template and adding it to the simulation. By adding several different distributionModels to the distributionTable of the rigid body emitter, the same emitter can emit rigid bodies with different geometries, sizes and materials. A short code example on how to create a distributionTable with one distributionModel:

```
// Create distribution table
agx::Emitter::DistributionTableRef distributionTable = new
agx::Emitter::DistributionTable(agx::Emitter::Quantity::QUANTITY_MASS);

// Create template
agx::RigidBodyRef bodyTemplate1 = new agx::RigidBody();
agxCollide::GeometryRef geometryTemplate1 = new agxCollide::Geometry(new agxCollide::Cylinder(0.4, 0.1));
```

```
geometryTemplate1->setMaterial(material);
bodyTemplate1->add(geometryTemplate1);

distributionTable->addModel(new agx::RigidBodyEmitter::DistributionModel(bodyTemplate1, 0.3));
emitter->setDistributionTable(distributionTable);
```

Finally the rigidBodyEmitter needs a volume in which rigid bodies will spawn. It is defined as an **agxCollide::Geometry**:

```
agxCollide::GeometryRef emitGeometry = new agxCollide::Geometry(new agxCollide::Sphere(3.0));
emitGeometry->setSensor(true);
emitter->setGeometry(emitGeometry);
```

15. AGX Wire

The agxWire module contains numerous classes for simulating wires, ropes and chains. These classes can be used to simulate cranes, winches, pulleys etc. The wire is an implementation of a lumped element structure with dynamic resolution. The main difference between classical lumped elements, such as chains of bodies and ball socket joints (or hinges), is that the wire will adapt the resolution so that no unwanted vibrations will occur. Whenever a wire slides over a geometry (such as a box, cylinder or trimesh), it will create shape contact nodes, which will stabilize contacts even under very high tension. The Wire is stable even under very high mass ratios/tension. This is necessary to be able to model real life scenarios with wires, chains, ropes etc.

Wire-wire interaction can also be enabled (not enabled by default). The overlap test between wire-wire is performed with sweep tests (continuous collision detection) to reduce the risk for accidental missed overlaps.

15.1. Known limitations

The current limitations of the wire are:

- A wire cannot handle torsion. However, rotation for Link's between Wire's can be controlled. See Section 15.21.6

15.2. agxWire::Wire

`agxWire::Wire` (wire) has one material for the whole wire. Available operations on a wire are: cut, reverse, merge. It can be winched using the class `agxWire::WireWinchController`.

Below is a table of classes in the agxWire namespace.

15.3. Modeling primitives

Table 15.1 Classes related to wire simulation

CLASS	Description
agxWire::Wire	Class for simulating a wire of a single segment type.
agxWire::WireWinchController	A winch that can spool in/out wire
agxWire::WireSimpleDrumController	A drum implementation for use with the agxWire::Wire class.

15.4. Wire nodes

Table 15.2 Node classes used for routing wires

CLASS	DESCRIPTION
agxWire::Node	Base class for a node used when routing an agxWire::Wire. A wire is forced to pass through this specified point.
agxWire::FreeNode	Nodes used for routing a wire at points in the world coordinate system.
agxWire::BodyFixedNode	Nodes used at end of wires to attach rigid bodies.
agxWire::EyeNode	A node which defines a "hole" through which the wire can slide. A Link can not slide through an EyeNode.
agxWire::ContactNode	OBSOLETE! A node which is either created by the user, or by the collision detection system to any edge where the line is overlapping geometries. Can only exist on geometries with single shapes. The shape can only be box, cylinder or some type of mesh.
agxWire::ShapeContactNode	A node which is either created by the user, or by the collision detection system to any edge where the line is overlapping geometries. No limitation on on the number of shapes for a geometry. Can exist on all shapes, expect lines, rays and WireShapes.
agxWire::ConnectingNode	A node commonly used to attach a rigid body between two agxWire::Wire. This node has additional stabilization constraints.

15.5. Rendering a wire

Table 15.3 Classes used for rendering/Storage

CLASS	DESCRIPTION
agxOSG::WireRenderer	Class for rendering an agxWire::Wire in OpenSceneGraph
agxWire::NodeConstIterator	Iterator for accessing nodes in a Wire.

15.6. Overall description of a Wire

15.6.1. Dynamic Resolution

A wire's route is defined by its *nodes*. There are various nodes for different purposes. Nodes are explained later in the document.

A very important concept of a Wire is the ability to change resolution of nodes per length unit. A Wire with a certain material (radius/density) will have a certain mass. If this mass is distributed along the wire in many nodes (lumped elements), we will get small masses in each node. If a Wire with many small masses is affected by a large tension, we will get oscillations due to the large ration between tension and mass of each mass element. Therefore, the Wire will automatically remove nodes during high tension.

The maximum resolution for a Wire is specified in its constructor

```
agx::Real resolution = 2; // Two lumped elements per meter is the max resolution.
agxWire::WireRef wire = new agxWire::Wire(
                      0.01,         // Radius
                      resolution ); // Number of elements per length unit
```

or through the call:

```
void agxWire::setResolutionPerUnitLength( agx::Real resolutionPerUnitLength );
```

15.6.2. Constraint model of a Wire

A wire consists of a set of bodies and constraints. From the attributes in the agxWire::Wire class, a structure is created which contain the physical model of a wire.

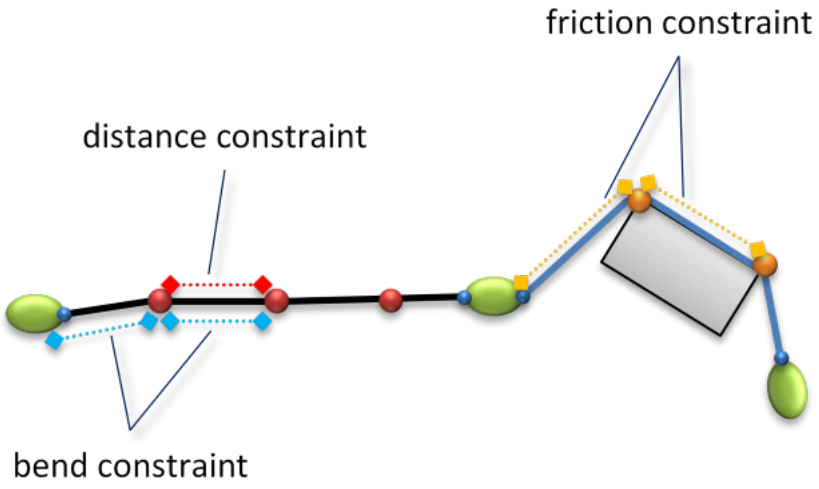


Fig. 15.1 The various constraints involved in the modeling of a Wire.

Fig. 15.1 illustrates the schematic structure of a simulated Wire. The constraints involved are:

Table 15.4 Constraints involved in a wire simulation

Constraint model	Description
Distance constraint	A distance constraint will try to keep the two bodies locked at a specified distance. There are distance constraints between all nodes (bodies) in a wire.
Bend constraint	A bend constraint is a three body constraint that will try to restore the wire into a straight line. A Wire with a large radius is of course much more resistant against bending than a thin rope. A bend constraint also increases stability for a Wire, especially for Links.
Friction constraints	Whenever a Wire collides with the edge of another geometry, a shape contact node is created. Depending on the material attributes (friction coefficient) a friction constraint will try to work against moving the wire over the edge, both along the edge and orthogonal to it.

15.6.3. Geometry model of a Wire

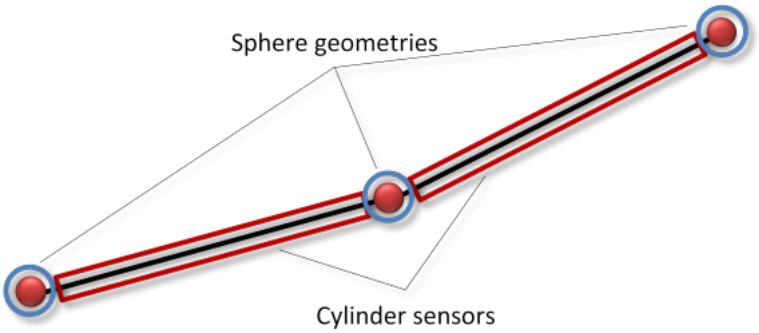


Fig. 15.2 Geometries associated to a Wire.

To make the wire interact with the rest of the simulation, it is also consisting of geometries. The geometries cause the wire to be able to collide with other geometries. A wire consists of two types of geometries:

1. **Sphere geometries**: Each lumped element (node with a rigid body) will be represented as a sphere (blue in above figure), with a radius derived from the radius of the wire/segment. This geometry collides with other geometry and creates contacts which the solver will see.
2. **Cylinder geometries**: Between each lumped element (node), a cylinder (red in the above figure) will be kinematically transformed between time steps. This cylinder is used for detecting contacts between the Wire and edges on other geometries. There are no contacts generated between this cylinder and other geometries that the solver will see.

Similar to an ordinary agxCollide::Geometry, the API for the wire has a number of methods to control which other geometries can collide with the geometries in the wire:

```
void setEnableCollisions(const agx::RigidBody* body, bool flag);
void setEnableCollisions(const agxCollide::Geometry* geometry, bool flag);
void addGroup(unsigned id);
void addGroup(const agx::Name& name);
```

15.6.4. Operations on a Wire

A wire is created in a *routing* process described below. When a wire is created it can be merged/cut/split/reversed for more information see Section 15.11.

15.6.5. Devices used with wires

A wire can be routed together with various devices, for example a *Winch* and *WireSimpleDrumController*. A winch is a device which resembles a real life winch. It is capable of pulling in and feeding out wire just as a winch would. The mass of the wire that is pulled into a winch will not be simulated.

15.7. agxWire::Node

A Node is used for routing a wire and to attach a wire to a certain RigidBody. A Node defines a point through which a wire is forced to pass. This means that a Node attached to a RigidBody will give a 1-1 interaction between the RigidBody and the wire.

The nodes are stored in an ordered list. If a wire hasn't been cut or detached, the first and last nodes are the ones the user used when creating the wire.

BodyFixed nodes (or lumped element nodes) are nodes that will appear at a certain fix distance from each other or a multiple of that distance. Each body fixed node is placed in the center of mass of a rigid body, which carry some of the mass of the wire. The dynamic resolution makes body fixed nodes (and their bodies) disappear and appear as the tension increases and decreases to keep the simulation numerically stable. A distance constraint is only defined between two body fixed nodes.

Note

Body fixed nodes can also be used as fixed connection points on bodies. They can be located at arbitrary relative positions to that body.

Note

If collisions between the rigid body and the wire are enabled, it is important to place the BodyFixedNode so that the whole wire is initially non-overlapping. This includes especially one extra wire radius around the attachment point. So:

```
agx::RigidBodyRef body = new agx::RigidBody(new agxCollide::Geometry(new agxCollide::Box(agx::Vec3(1, 1, 1))));
// ...
agxWire::WireRef wire = new agxWire::Wire(0.1, 10);
wire->add(new agxWire::BodyFixedNode(body, agx::Vec3(1 + 0.1, 0, 0))); // Note the extra 0.1
```

Eye nodes are nodes that can never start or end a wire. They consist of two points that both are fixed relative a rigid body which is NOT part of the wire. Their purpose is then to make the rigid body slide along the wire.

ShapeContact nodes work as eye nodes except for two major differences: they come and go as the wire collides and slips off geometries and they slide (with friction) along edges (see Section 15.8.1) of shapes.

Free nodes (could also be called *temporary lumped nodes*) are used for routing. They are kept as long as the wire is numerically stable. If the actual position of a free node along the wire will give us stability problems but still is considered necessary to keep to retain the orientation and most of the energy, the node may travel along the wire to a position where the tension could be handled by the solver. If the free node isn't necessary to keep it is removed forever. When a contact between the wire and geometry is supposed to be removed, the contact node is transformed into a FreeNode instead, to preserve the wire orientation.

15.7.1. agxWire::BodyFixedNode

BodyFixedNode is a node that attaches a wire to a point *relative* to a RigidBody.

```
// Arguments: the body and an attachment point relative the body
agx::BodyFixedNodeRef bf = new agxWire::BodyFixedNode( body, Vec3( 1,0,1 ));
```

15.7.2. agxWire::EyeNode

EyeNode is a node that can slide along a wire. It has two friction parameters, in two separate directions. *Positive* is specifying friction when a body is sliding along the wire towards the end of the wire, *Negative*, is when a body is sliding towards the beginning of a wire. The beginning of a wire is determined as the first point of routing.

An EyeNode can also specify a point and a normal, which makes it possible to attach a RigidBody and make it rotate as the wire moves and vice versa.

```
agxWire::EyeNodeRef eye = new agxWire::EyeNode( slidingObject.getRigidBody(),
                                                Vec3( 1,0,1 ),    // First point relative to body.
                                                Vec3( -2,0,0 ) ); // Vector pointing from first point
                                                                  // To the second point
```

The third argument is a vector pointing out the direction and distance to the second point of attachment between the wire and the rigid body.

An EyeNode can also have a radius which determines whether a Link with a specified radius can pass through or not.

To specify the friction of the wire sliding through the eye, use the wire material:

```
eyeNode->getMaterial()->setFrictionCoefficient( 0.5 );
```

It is also possible to specify different friction in different directions:

```
eyeNode->getMaterial()->setFrictionCoefficient( 0.5, agxWire::NodeMaterial::NEGATIVE );
eyeNode->getMaterial()->setFrictionCoefficient( 0.5, agxWire::NodeMaterial::POSITIVE );
```

To have the eye node constrained to an area instead of a point the node can be attached to an `agxWire::EyeNodeArea`. The eye node is in this case attached to a rigid body that is constrained to another rigid body.

```
// Rigid body that is associated with the area in which the eye node can move
agx::RigidBodyRef eyeNodeAreaBody = new agx::RigidBody( eyeNodeAreaGeometry );
simulation->add( eyeNodeAreaBody );

agx::RigidBodyRef eyeBody = new agx::RigidBody(); // Eye node body
eyeBody->setPosition( position ); // Initial position should be inside the constraining area.
simulation->add( eyeBody );

agxWire::WireRef wire = new agxWire::Wire( wireRadius, wireResolution );
wire->add( new agxWire::FreeNode( startPosition ) );
wire->add( new agxWire::EyeNode( eyeBody, agx::Vec3() ); // Body position should be the same as node position.
wire->add(  new agxWire::FreeNode( endPosition )  );
simulation->add( wire );

// An area definition that specifies the constraining area. It is specified in body coordinates.
agxWire::AreaDefinitionRef areaDefinition = new agxWire::CircularAreaDefinition( localPoint,
                                                                                 radius,
                                                                                 eyeNodeAreaBody,
                                                                                 localDir,
                                                                                 eyeNode );
```

```
agxWire::EyeNodeAreaRef eyeNodeArea = new agxWire::eyeNodeArea( areaDefinition );
simulation->add( eyeNodeArea );
```

The area definitions available are `agxWire::CircularAreaDefinition` and `agxWire::ConvexAreaDefinition`.

The circular area definition is defined by a local point on the constraining body that will be center of the circle, a local direction on the constraining body and a radius.

```
CircularAreaDefinition::CircularAreaDefinition( const agx::Vec3& localCenter,
                                                      agx::Real radius,
                                                      agx::RigidBody* constrainingBody,
                                                const agx::Vec3& localDir,
                                                      agxWire::EyeNode* eyeNode );
```

The convex area definition is defined by a local point on the constraining body, a local direction on the constraining body and a set of local points that is projected onto the plane specified by the point and direction.

```
ConvexAreaDefinition::ConvexAreaDefinition( const Vec3& localCenter,
                                                  Vector< Vec3 > localAreaCoordinates,
                                                  RigidBody* constrainingBody,
                                            const Vec3& localDir,
                                                  EyeNode* eyeNode )
```

It is also possible to create a custom area by creating a class that inherits from agxWire::AreaDefinition.

An example of how an eyeNodeArea is used is available in **tutorial_wire1.cpp**, Tutorial 8.

15.7.3. agxWire::FreeNode

FreeNode are nodes that can be used during the routing procedure of a wire. They are nodes that will disappear when the tension increases in the wire.

```
wire->add( new agx::FreeNode( Vec3( 52,2,10 ) );  // Position in world coordinates
```

15.8. Wire Material

The class **agxWire::Wire** stores an instance of an **agx::Material**. This material is used for two things:

Bulk attributes: Density is used to calculate the internal material properties for a wire. Whenever this material is updated with new properties, the physical properties are recalculated.

Surface attributes: The SurfaceMaterial part is used during contact processing to calculate restitution and friction like any other geometry collision. It is possible at any point in time to change the material of a segment:

```
const agx::Real resolution = 1.0;
const agx::Real radius = 0.01;
// Create a material
agx::MaterialRef wire_material = new agx::Material("wire_material");
simulation->add( wire_material );

agxWire::WireRef wire1 = new agxWire::Wire( radius, resolution )

// Associate the new material to this Segment
wire1->setMaterial( wire_material );

// Set properties
wire_material->getWireMaterial()->setYoungsModulusBend( 1E11 );
wire_material->getSurfaceMaterial()->setRoughness( 0.2 ); // Friction
```

It also possible to use the wire material when creating *explicit* ContactMaterials:

```
// Create an explicit contact material for this wire_material colliding
// with another material
agx::ContactMaterialRef wire_plastic_material = new agx::ContactMaterial( wire_material, plastic_material );

// Set the attributes of the explicit material
wire_plastic_material->setFrictionCoefficient( 0.5 );
```

```
simulation->add( wire_plastic_material );
```

The material used for a Wire is internally modified (for point friction) which makes it unsuitable for use with other geometries in a simulation.

Attention

Avoid using the same material for an agxWire::Wire as other agxCollide::Geometries.

15.8.1. Wire Friction

A wire can interact with other geometries in two ways:

1. A wire segment intersects a geometry and a ShapeContactNode is inserted, with an edge to move along. This when using the kinematic wire contact algorithm, which solve the dynamics, including friction, of shape contact nodes in a co-simulation.
2. A wire collides on the surface of another geometry. Either a rigid wire segment when using the dynamic wire contact algorithm, described in Section 15.17.2. or a BodyFixedNode.

Wire friction for BodyFixedNode's and for the DynamicWireContact is defined as friction for any two geometries colliding, see Section 11.15 about materials.

Wire friction for a ShapeContactNode is defined both along the wire and along an edge defined orthogonal to the wire direction. See Fig. 15.3 for an illustration. When the *wire* (iii) is in *contact* (v) with a *geometry* (iv) and sliding along its *edge* (vii) we find a friction *force* (ii) along the edge. If the *normal force* (i) is small enough we see sliding friction, otherwise the contact will not move. The algorithm for defining edges for a ShapeContactNode make use of the wire direction and the local curvature of the shape the wire is in contact with.

To distinguish friction along the wire and along the edge for a ShapeContactNode primary (along the wire) and secondary (along the edge) friction direction is specified to set them differently. By default the wire friction coefficients are the same as defined for the contact material for a wire-geometry contact. Creating an explicit contact material the default wire friction coefficients can be overridden by setting the wire friction coefficient.

```
// Create an explicit contact material for this wire_material colliding
// with another material
agx::ContactMaterialRef wire_plastic_material = new agx::ContactMaterial( wire_material, plastic_material );

// Set the wire friction coefficient along the wire
wire_plastic_material->setWireFrictionCoefficient( 0.5, ContactMaterial.PRIMARY_DIRECTION );
// Set the wire friction coefficient along the shape contact edge
wire_plastic_material->setWireFrictionCoefficient( 0.1, ContactMaterial.SECONDARY_DIRECTION);

simulation->add( wire_plastic_material );
```

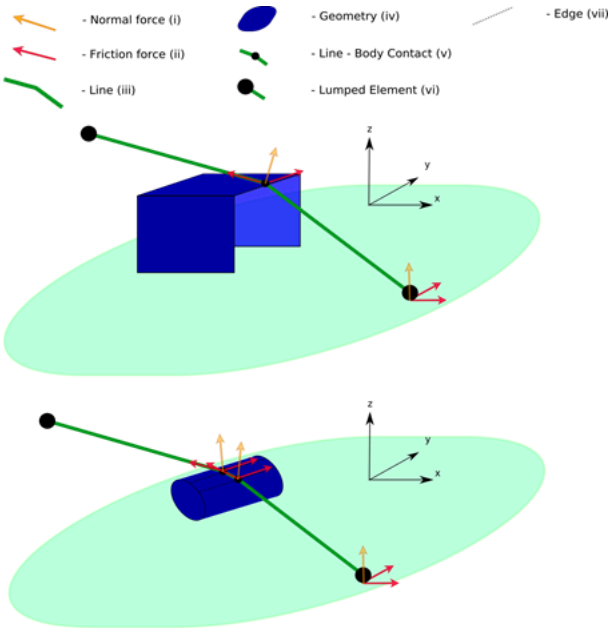


Fig. 15.3 Illustration of wire friction.

15.9. Creating a agxWire::Wire

Below is a description of the process of creating a wire.

15.9.1. Routing a wire

A wire is placed into the world by a routing process. During the routing, winch controllers and nodes of various types are created and positioned attached to rigid bodies. The nodes are then added to the wire.

During wire routing, the route nodes added are stored locally in the wire. I.e. no communication occurs with the constraints. The wire has not been added to a simulation at this point. The routing ends when the wire is added to the simulation.

Attention

This means that adding a wire to the simulation should be done *after* all the routing is done.

When the first node is added, a check for a possible winch controller at the beginning of the wire is done. If one is present, and it for example does not have auto-feed enabled, we have to take care of the rolled in length and count that to the total routed length.

For all nodes added after the first one, we calculate the current route length. If the route length does not exceed the total length of the wire, we add the node to the local list of routed nodes.

15.9.1.1. Process for routing a wire

1. Initialization
    1. Create all bodies (involved in routing)
    2. Create WinchControllers at relative position to bodies (or in the world)
    3. Create eyes, at positions relative to bodies
    4. Position bodies at their correct positions.
2. Routing:
    1. Route beginning of wire either through a winch, a BodyFixedNode, a ConnectingNode or a FreeNode.
    2. Route through WinchController (wire->add())
    3. Route through eyes (wire->add())
    4. Route through final WinchController (wire->add()) or
    5. You always have to end the route with a WinchController, BodyFixedNode, ConnectingNode or a FreeNode.
3. Add wire to simulation
4. Simulate.

It is always important to remember to always position the rigid bodies/Nodes before the routing process starts. This is the same requirement as when configuring other constraints in AGX.

15.9.2. Add to simulation

When a wire is added to the simulation it will be assumed that the routing is finalized.

Starting at the end we check for winch controllers, if present, one of the following scenarios may occur:

1. **Winch at both ends, both with auto-feed enabled.**

   Pulled in length for the end winch is set to zero and all superfluous wire is moved to the first winch.

2. **Winch at both ends, first with auto-feed enabled and second with auto-feed disabled.**

   If this is the case, errors (warning messages, return false and ignoring the last winch during routing) occurs if the wire outside the winch is not long enough. Other than that this is a well-defined case.

3. **Winches at both ends, first with auto-feed disabled and second with auto-feed enabled.**

   This means that auto-feed flag is used wrong. The purpose of the flag is to pull in all wire inside the first winch and then route while the length inside the first winch is changed during routing. If this case appears, all superfluous wire is put in the first winch, the one that does not have auto-feed enabled.

15.10. Rendering a agxWire::Wire

Rendering a wire is done using *RenderIterators*. Basically, a RenderIterator contains enough information to get a world position and tension of a control point in the wire. The code for agxOSG::WireRenderer can be found in <agx-dir>/agxOSG/src/WireRenderer.cpp,.h

```
agxWire::WireRef wire = createAWire();
simulation->add(wire);
```

```
simulation->stepForward();       // Take a time step
agx::Real refMaxTension = 1E4; // Just to get some reference for setting color/weight of a
                               // controlpoint in the spline

// Create a graphical representation of a spline in your rendering engine
Rendering::Spline *spline = new Rendering::Spline();

agxWire::RenderIterator it        = wire->getRenderBeginIterator();
const agxWire::RenderIterator end = wire->getRenderEndIterator();

// Loop through all the nodes in the wire
while ( it != end ) {
  const agxWire::Node* node = *it;

  // We can also read tension at the node to scale, control the rendering
  spline->add( node->getWorldPosition(), Real( 0.9 ) );
  ++it;
}
```

15.11. Wire operations

15.11.1. Cut a wire

To cut a wire there are two methods available:

```
// Cut a wire at the specified wire length
agxWire::Wire* agxWire::Wire::cut( const agx::Real& length , size_t minimumResolution, bool includeWinchPulledInLength =
true );

// Cut the wire at the closest point one the wire
// from the specified position
agxWire::Wire* Wire::cut( const agx::Vec3& position, size_t minimumResolution );
```

Both methods return a pointer to the second part of the cut wire. The wire returned is already inserted into the simulation and is ready to use. The two wires does not share any information, so they can be removed independently from the simulation.

15.11.2. Merge two wires

Two wires can be merged into one using the *merge* method:

```
bool agxWire::Wire::merge( agxWire::Wire* wireToMergeWith );
```

An example:

```
agxWire::WireRef a = createAWire();
agxWire::WireRef b = createAWire();

a->merge( b );
```

In the above example **b** will be added to the end of **a**. This method will only work if the last node in **a** and the first node in **b** are **FreeNodes**.

Attention

To be able to merge two wires wire *A* and wire *B*, the last node of *wire A* and the first node of *wire B* have to be FreeNodes.

If the two end points to be merged are not at the same position, additional *wire will be added to the last wire segment as a straight wire between the end of the first and beginning of the first*. This means that the resulting wire length after a merge is always >= length1+length2

The wire given as an argument to merge() will be deleted from the simulation. If no references are holding it, it will be deallocated.

15.12. Controlling wire resolution

The wire has an overall resolution controlled at construction or by using the method `Wire::setResolutionPerUnitLength( agx::Real resolutionPerUnitLength );` However, in some cases you might want to have different resolution along the wire. One typical scenario is where you have wire on a ship deck, and the wire goes into water for several kilometers and then up to another ship deck. It would be a waste of resources to have the same resolution along the whole wire. On deck, you might want to have higher resolution and in the water, it can be enough with a lot less nodes during simulation.

This can be controlled using the *HighResolutionRange* class that is available for nodes.

First you need to identify a node on the wire which acts as the reference. Usually the begin or the end of a wire but intermediate nodes, such as eye nodes, may also be reference nodes for higher resolution. The high resolution range may be at any distance before or after the reference node along the wire (not necessarily overlapping the reference node).

```
agx::Real default_resolution = 0.1; // One node every 10 meters
agxWire::WireRef wire = new agxWire::Wire(0.01, default_resolution);
agxWire::WireWinchControllerref winch = new agxWire::WireWinchController(winch_body, agx::Vec3(0,0,0), agx::Vec3(1,0,0));
wire->add(winch);

double start_distance = 2; // Start of the range with higher resolution
double end_distance = 10;  // End of the range with higher resolution
agx::Real new_resolution = 2; // Two nodes every meter

  // Set the high resolution range. In this range the wire will get higher resolution.
winch->getStopNode()->getHighResolutionRange()->set( start_distance, end_distance, new_resolution );
```

You can also have several high resolution ranges along a wire. Also, you cannot reduce the resolution, only increase.

15.13. agxWire::WireWinchController

The class `agxWire::WireWinchController` is an implementation of a winch that can pull in and winch out wire. It is based on a constraint (Prismatic) which means that it applies forces on a wire just as any other constraint or RigidBody in the system. It has functionality to haul in Links (which it then automatically disables when it is winched in the winch). A Winch can only be routed to the beginning or the end of a wire. A winch has a motor and a brake that can be controlled independently from each other. Remember that a motor cannot hold a load a specific position by setting the speed to zero. This is related to the problem of non-holonomic constraints that occurs with all velocity constraints. To hold a load a certain position, a holonomic (position) constraint should be used, in this case the winch brake.

For composite wires (wires with links in between) there is also the `agxWire::Winch` class. (see Section 15.21.3)

A winch is always attached to a RigidBody:

```
agxWire::WireWinchControllerRef winch = new agxWire::WireWinchController(
                            rigidBody,                              // RigidBody onto which sd is attached
                        agx::Vec3( 0, 0, -1 ),  // Position relative to rigidBody
                        agx::Vec3( 0, 0.0, -1); // Normal direction
```

A winch can be told to pull in any “loose” wire so that the wire is stretched after the routing:

```
winch->setAutoFeed( true );
```

Routing a wire to the winch is done by adding the winch to the wire:

```
// Add the winch to the wire, the second argument is the amount of wire pulled
// in to the winch at start.
wire->add( winch, wireLength );
```

The above call will result in that the winch will pull in all the resulting wire after it is routed.

A winch can be detached from a wire with a call to:

```
wire->detach(bool front);
```

where *front* has to be set to *true* if the winch is in the front end of the wire, *false* for the back.

The winching mechanism of a winch can be activated and controlled through the following API:

Set the maximum force for holding the wire (keeping velocity at 0):

```
void Winch::setBrakeForceRange( agx::Real brakeForce );
```

Set the desired winch in/out speed:

```
void Winch::setSpeed( agx::Real speed );
```

Set the maximum force by which the desired speed will be obtained:

```
void Winch::setForceRange( agx::Real winchForce );
```

The force applied by the winch motor and the winch brake can be accessed via:

```
auto winchForce = winch->getCurrentForce();

auto winchBrakeForce = winch->getCurrentBrakeForce();
```

Note

If all wire has been spooled out (`agxWire::WireWinchController::getPulledInWireLength()` reports zero), the forces reported will not be consistent. We recommend that you avoid spooling out all wire. Instead stop the winch before all wire is spooled out, or detach the wire from the winch.

15.14. Wire-wire interaction

Wires can also collide with other wires. This functionality has to be explicitly enabled between pairs of wires that are supposed to be able to interact. The class **agxWire::WireController** controls this behavior.

To enable wire-wire interaction between two instances of agxWire::Wire:

```
agxWire::WireRef wire1 = createWire(); // Create a wire
agxWire::WireRef wire2 = createWire(); // Create a wire

// Enable wire-wire collisions between the two wires
agxWire::WireController::instance()->setEnableCollisions(wire1, wire2, true);
```

For stability of wire-wire interaction, the coefficient of restitution for the contact material between the two wires should be set to 0.

```
auto* cm = simulation->getMaterialManager()->getOrCreateContactMaterial(wire1->getMaterial(),
                                                                        wire2->getMaterial()); // Obtain the contact
material
cm->setRestitution(0);
```

Also, the value of YoungsModulusBend for material of the respective wires should be set to realistic values.

```
wire->getMaterial()->getWireMaterial()->setYoungsModulusBend(1E9);
```

If a chain should be simulated instead of a relatively stiff wire, this can be achieved by turning off the bend resistance altogether, instead of setting it to zero:

```
wire->getMaterialController()->setIsBendResistant(false);
```

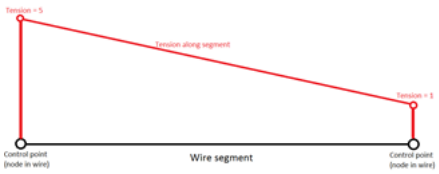
15.15. Reading tension/forces

After a time step (a call to simulation->stepForward()), the tension involving the wire is updated and available.

The tension at each node along the wire can be read through the specified node in the wire or through a point in space or a length along the wire.

15.15.1. Get tension given distance along wire or point in the world

The tension can be queried given a point in space, or a length from start along the wire. These two methods will do a linear search to find the correct point along the wire. The data will be returned into a struct: agxWire::WireSegmentTensionData containing various data. The data will be linearly interpolated between the nodes in the wire:

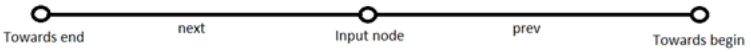


```
// Closest point to the wire will be used
double tension = wire->getTension( agx::Vec3(1,1,1) ).getAverageRaw();

// Get the tension in the wire a certain length along the wire
double tension = wire->getTension( 123 ).getAverageSmoothed(); // 123 m from start
```

15.15.1.1. Get tension given node

The node tension data, `agxWire::WireNodeTensionData`, contains data for two segments - the segment before and the segment after the node. I.e., the tension exactly before and exactly after the node.

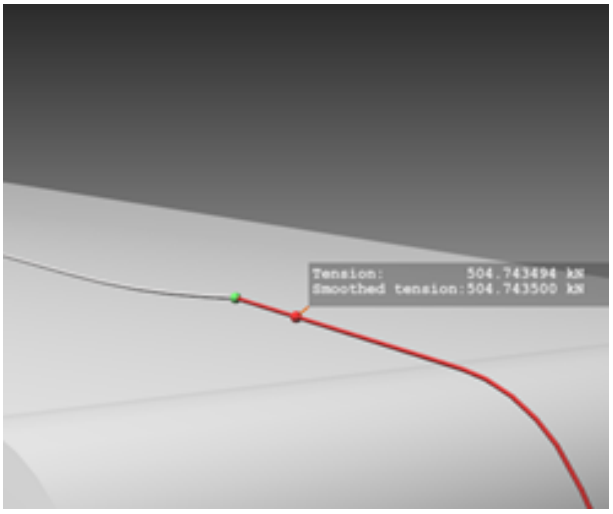


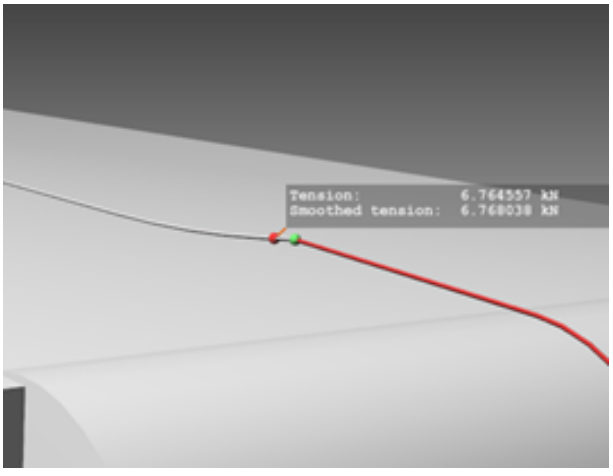

```
for ( agxWire::RenderIterator i = wire->getRenderBeginIterator(); i != wire->getRenderEndIterator(); ++i )
{
  double tension = 0;
  agxWire::WireNodeTensionData data = wire->getTension( *i ); // Get tension at the node
  tension = data.getAverageSmoothed();
}
```

In comparison to “get tension given distance along wire”, fetching data given node is computationally much cheaper.

15.15.1.2. Tension at nodes

The tension along a wire isn’t continuous - so beware of, and think of, what it means to read tension given nodes. Consider a closed Shark Jaw, as the two pictures below, modeled with an eye node (green sphere) that has infinite friction.





In this example, tension drops instantaneously from 504 kN to 7 kN over the eye node making it hard to interpret the exact tension value at the node. By definition (internally) the node at the end of a segment is carrying the tension data, i.e., if one ask the node what tension it feels it can either be 504 kN or 7

kN depending on in which way the wire is routed (or after reverse). It's important to be consistent and think twice which value one expects in this special case.

Of course it becomes more convenient with our data holder, `agxWire::WireNodeTensionData`, which contains data for the segment before and after the node. But following the above definition, `agxWire::WireNodeTensionData::getRaw()`, returns the raw tension value from the segment before the node.

15.15.2. Normal force between a geometry and the wire

To get the normal force between a specified geometry and a wire you first have to locate a shape contact node in the wire which is interacting with the specific geometry, then you can query the node's contact material for the normal force magnitude.

```
for ( agxWire::RenderIterator i = wire->getRenderBeginIterator(); i != wire->getRenderEndIterator(); ++i )
{
  double magnitude = 0;
  // Tension is only available at body fixed nodes
  agxWire::ShapeContactNode *cn = i.get()->getAsShapeContactContact();
  if (cn)
    magnitude = cn->getMaterial()->getNormalForceMagnitude();
}
```

15.16. Applying forces to a wire

If you want to simulate for example drag in water, wind, buoyancy or other external forces on a wire, you can do this with a ForceField. A *ForceField* will be updated inside the solve stage. This means that the callback in a user ForceField must never make any structural changes to the system. The only allowed operations is applying torque/forces onto rigid bodies, all other operations (except reading data) should be strictly avoided.

Below is a code snipped that creates a custom ForceField which applies forces to each rigid body of a wire.

```
// Derive from baseclass agx::ForceField
class MyForceField : public agx::ForceField
{
  public:
    MyForceField( agxWire::WireRef wire )
      : m_wire( wire ) {}

    // Virtual method called from within the solver
    virtual void updateForce( agx::DynamicsSystem* ) AGX_OVERRIDE
    {
      if ( m_wire == 0L )
        return;

      // Iterate over all nodes in the wire
      agxWire::RenderIterator it = m_wire->getRenderBeginIterator();
      while ( it != m_wire->getRenderEndIterator() ) {
        agxWire::Node* node = *it;
        // Check for internal, dynamic, mass nodes.
        if ( node->getType() == agxWire::Node::BODY_FIXED )
          node->getRigidBody()->addForce( calculateForce( node ) ); // Apply the force

        ++it;
      }
    }

  protected:

    // Some method for calculating a force
    // Could be calculating gravity, windforce, buoyancy or something else
    agx::Vec3 calculateForce( const agxWire::Node* node ) const
    {
      return agx::Vec3( agx::PI );
    }

  protected:
    agx::observer_ptr< agxWire::Wire > m_wire;
};

...
```

```
// Add the custom force field to a simulation
simulation->add( new MyForceField( wire ) );
```

15.17. Wire collisions and contacts

Up until this point the described agxWire::Wire contact behavior assumes that the default wire contact model is used. This model is called *"default kinematic contact model"*. There are two other models to consider, resulting in improved interaction fidelity.

- *Default kinematic contact model* – Default model with unconditionally stable behavior
- *Dynamics wire contact model* – Most dynamic/realistic behavior

15.17.1. Default kinematic contact model

One of the features of a kinematic model is that it is unconditionally stable. Wire collisions result in kinematic nodes (agxWire::ShapeContactNode) on the collided geometry.

To achieve highest performance combined with stability under high tension the *default kinematic contact model* is recommended.

When modeling of complex scenes with complex geometry wire collision can be computationally expensive. Collision between wires and complex geometry is not recommended when high performance is prioritized. Instead specific wire collision geometries are recommended to be used. I.e. position geometries containing one box or one cylinder to represent the complex geometry for the wire collisions (low interaction fidelity).

15.17.1.1. Wire kinematic model for AGX versions previous 2.19

For AGX versions previous 2.19 the kinematic contact model generated agxWire::ContactNodes at collision. The ContactNode is now deprecated and replaced with the ShapeContactNode which handle all shapes and has no restrictions on the number of shapes per geometry. New from 2.19 is that the wire has a class called WireShapeContactController replacing a previous class called WireContactController.

Restoring a wire stored from a version of AGX lower than 2.19 could include ContactNodes. The restored wire by default use the old wire contact controller. All contact nodes can be converted to shape contact nodes for a specific wire. For the wire to generate shape contact nodes from future interactions, the contact controller type must be changed from *OLD_CONTACT_CONTROLLER* to *SHAPE_CONTACT_CONTROLLER*.

```
// Saves simulation including a wire
agxStream::InputArchive& storedScene;
// Wire object to restore
agxWire::Wire* wire;

// Choose to use the shape wire contact controller,
// for shape contact node generation at collision
wire->setActiveContactControllerType(agxWire::SHAPE_CONTACT_CONTROLLER);

//convert all contact nodes to shape contact nodes
wire-replaceContactNodesWithShapeContacts();
```

15.17.2. Dynamic wire contact model

For a more realistic/fully dynamic behavior of wire/geometry interaction is required the *Dynamic wire contact model* is recommended. This model introduces new bodies, as part of the wire, wherever the wire has collided. It is therefore not expected to demonstrate the same stability under high tension as the kinematic model.

The dynamic wire contact model is enabled per geometry:

```
agxCollide::GeometryRef geometry = new agxCollide::Geometry();
agxWire::WireController::instance()->setEnableDynamicWireContacts(geometry, true);
```

This means that all wires will generate dynamic contacts when colliding with a geometry where the dynamic wire contacts are enabled. A limitation of the current implementation is that if different behavior is wanted from two different wires two different geometries must be used.

15.18. Automatic wire split for performance enhancement

agxWire::Wire support a notion of kinematic splitting during simulation. AGX Dynamics has an automatic partitioner which can analyze a system and split a simulation into subsystems (Section 38.6.2.3). This allows for the task/threading system to solve different parts in different threads. Thus improving performance in the overall system.

```
// Enable kinematic splitting for a wire
wire->setEnableSplitting( true );
```

By enabling the kinematic splitting for a wire, we allow for the system to introduce *kinematic* bodies along the wire, but only intermittent during one time step. Which of the bodies in the wire that are made kinematic will be determined by the splitting algorithm. To avoid artifacts, the place for splitting will be moved between time steps.

For wires under high tension it might not be possible for the splitting algorithm to introduce a kinematic body. This would potentially create unstable/unrealistic simulations.

15.19. Simulating a Pulley/Sheave/Gypsy Wheel

A sheave is a pulley with a grooved wheel for holding a belt, wire rope, or rope. The grooved wheel spins on an axle or bearing inside the frame of the block. This allows the wire or rope to move freely minimizing friction and wear on the cable. Sheaves can be used to redirect a cable or rope, lift loads, and transmit power. (from Wikipedia)

A Gypsy wheel is usually a motorized Sheave, meaning it is used for pulling in/out chain. The chain will have infinite friction due to mechanical jamming.

Commonly a cylinder Shape is used for modeling a Sheave. However a problem is that the wire can easily slip of the cylinder, especially if the cylinder is narrow in comparison to the wire radius. To assign a sheave geometry the “Pulley” property:

```
// Assign the "Pulley" property to a Geometry (which contains a Cylinder shape)
sheaveGeometry->getPropertyContainer()->addPropertyBool("Pulley", true);
```

This will tell the Wire contact algorithms to only create contact nodes at the center-line of a cylinder.

A Gypsy wheel, which is a motorized pulley can be defined in a similar manner:

```
// Assign the "Gypsy" property to a Geometry (which contains a Cylinder shape)
gypsyGeometry->getPropertyContainer()->addPropertyBool("Gypsy", true);
```

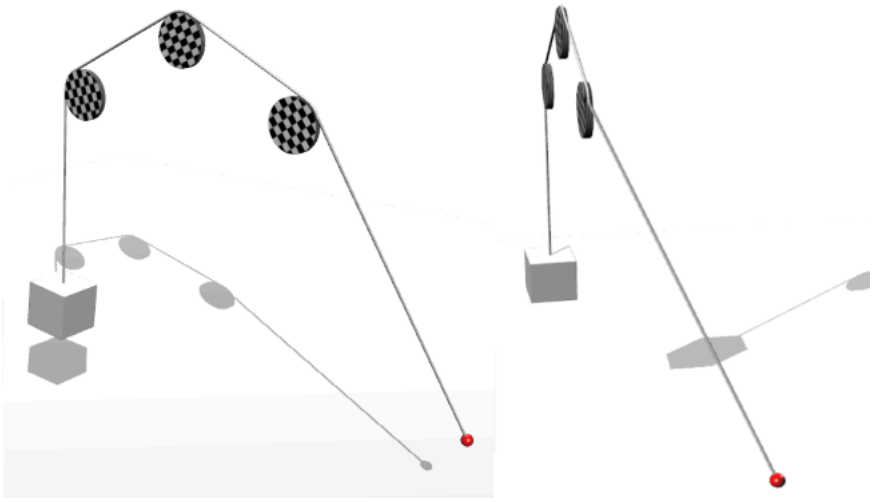


Fig. 15.4 Screenshot from the script wire_pulley.agxPy illustrating the use of the “Pulley” property. Here we can see that the wire is not slipping of the pulleys, even if the winch is pulling the wire sideways.

15.20. Hydro- and aerodynamics

A wire is affected by the fluid that surrounds it. A WindAndWaterController can be used to simulate the affects in water, air and other fluids, see Section 21.

15.21. Connecting wires, agxWire::Link

In general, an agxWire::Wire has no knowledge of neighboring connected wires. So, for example if one would like to connect a wire to a chain using a swivel, or wires of different radii using a shackle, or three wires connected through a tow plate – certain functionalities of the wire would not work as expected since all features are handled locally in the agxWire::Wire object.

agxWire::Link enables such functionality to work over connected wire segments. A link contains a user defined rigid body (with arbitrary geometry), a number of connecting wires and algorithms to transfer functionality between the wires.

15.21.1. Features

An agxWire::Link can have an arbitrarily number of connections. Typically, one or two connections is recommended for all functionality to be enabled.



Fig. 15.5 A link with a number of different wire segments connected to it.

The mass of a link is normally relatively small in relation to the tension the connected wires could have, so an agxWire::Link includes additional stabilization constraints when the link is exposed to large forces.

15.21.2. Configuration

For simplicity, assume this configuration:

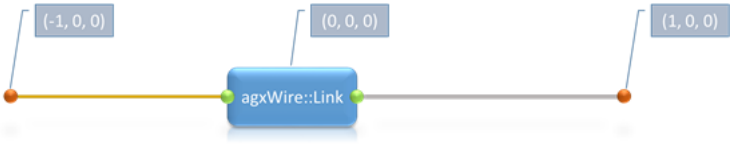


Fig. 15.6 The most common configuration where two wire segments are connected with a link in between.

Two wires connected to a link. Wire 1 has a free begin at (-1, 0, 0), the link positioned in the origin and Wire 2 ends in a free end at (1, 0, 0). Create the wires and the link:

```
agxWire::WireRef wire1 = new agxWire::Wire( wire1Radius, wire1Resolution );
agxWire::WireRef wire2 = new agxWire::Wire( wire2Radius, wire2Resolution );

// Link rigid body with box geometry.
agx::RigidBodyRef linkRb = new agx::RigidBody( new agxCollide::Geometry( new agxCollide::Box( linkHalfExtent ) ) );
linkRb->setPosition( 0, 0, 0 );
agxWire::LinkRef link = new agxWire::Link( linkRb );
```

The link *must* be positioned any time before the wires are initialized. When the wires are initialized the current position of the link defines the lengths of wires.

The link has to know where it's connected to the wires. A link is either connected to a wire's begin or end position of a wire and there are several ways of defining this.

Connection type from a links point of view:

```
/**
Connection type of the wires connected to this link, i.e., if this
link should be attached to begin or end of the wire.
*/
enum ConnectionType
{
  WIRE_BEGIN         = 0,    /**< Link attached at begin of wire. */
  WIRE_END           = 1,    /**< Link attached at end of wire. */
  INVALID_CONNECTION = 0xFF  /**< Invalid connection, e.g., if this link isn't connected to a give wire. */
};
```

Defining the connections to a link is only a mapping of how the wire composition relates. No nodes are added to the wires!

15.21.2.1. Routing with explicit connection type

Explicitly assigning the connection type:

```
// on the link is given in the link's coordinate system, and
// could in this case be: (-linkHalfExtent.x(), 0, 0)
link->connect( wire1, wire1ConnectionPosition, agxWire::Link::WIRE_END );

// The link is connected to Wire 2 begin. The connection position
// on the link is given in the link's coordinate system, and
// could in this case be: (linkHalfExtent.x(), 0, 0)
link->connect( wire2, wire2ConnectionPosition, agxWire::Link::WIRE_BEGIN );
```

The link now has two connections and now we can route the above system:

```
// Wire 1 first free node.
wire1->add( new agxWire::FreeNode( -1, 0, 0 ) );
// Add the link. The wire will find out how it relates to the
// link and add a node to the wire route. 'false' means that
// the wire shouldn't update the connection type and instead
// fail (return false) if the connection type is a mismatch.
wire1->add( link, false );

// Wire 2 starts at the link...
wire2->add( link, false );
// ...and ends at the free end (1, 0, 0).
wire2->add( new agxWire::FreeNode( 1, 0, 0 ) );
```

The above route is a valid route and the second argument to the add method also assures the connections are valid given the wanted route. If the connection type isn't that important, pass true as second argument (default is true) and the wire will match and update the connection type to the link. Example:

```
link->connect( wire1, wire1ConnectionPosition, agxWire::Link::WIRE_END );
print( link->getConnectionType( link->getConnectingNode( wire1 ) ) ); // "WIRE_END"
// Routing the other way around. Link is attached at wire begin.
wire1->add( link );
wire1->add( new agxWire::FreeNode( -1, 0, 0 ) );
print( link->getConnectionType( link->getConnectingNode( wire1 ) ) ); // "WIRE_BEGIN"
```

15.21.2.2. Routing with implicit connection pair

agxWire::Link has a static utility method to configure the most common configuration (Fig. 15.6), so configuring such setup can be done with one call:

```
// Wire 1 will get WIRE_END connection type and Wire 2 WIRE_BEGIN.
agxWire::Link::connect( wire1, wire1ConnectionPosition, link, wire2, wire2ConnectionPosition );
```

And the actual routing:

```
wire1->add( new agxWire::FreeNode( -1, 0, 0 ) );
wire1->add( link );

wire2->add( link );
wire2->add( new agxWire::FreeNode( 1, 0, 0 ) );
```

15.21.2.3. Routing and connecting – all in one call

When routing, a wire has the possibility to know, given nodes already added or not, which connection type a wire should have relative the link. Routing and connecting the system in (Fig. 15.6):

```
wire1->add( new agxWire::FreeNode( -1, 0, 0 ) );
// Will connect to the link and give Wire 1 connection type WIRE_END.
wire1->add( link, wire1ConnectionPosition );

// Will connect to the link and give Wire 2 connection type WIRE_BEGIN.
wire2->add( link, wire2ConnectionPosition );
wire2->add( new agxWire::FreeNode( 1, 0, 0 ) );
```

15.21.3. agxWire::Winch

The `agxWire::Winch` class is derived from the class agxWire::WireWinchController. It adds a few additional features:

1. Can pull in links.
2. Can pull out links.
3. Can store completely inactive wires and their rest lengths.

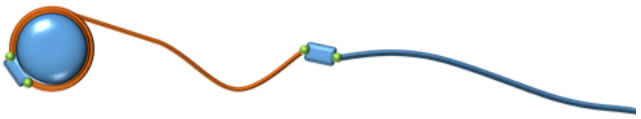


Fig. 15.7 Schematic figure of a wire composition with two wire segments. One completely pulled in segment (orange) connected to an (also inactive) pulled in link. The active wire (blue) is connected to the inactive wire through another active link.

An agxWire::Winch simulates a linear winch all units of forces are in Newton (N).

15.21.3.1. Length of a wire that hasn't been routed?

The **agxWire::Wire** object doesn't have a pre-defined length. The length of a wire is only defined by the pulled in length in a winches plus the active distance outside. So for a wire to have a length it must have at least two nodes.

A completely pulled in wire doesn't have active nodes. It only has a connection to a link – which also is inactive (i.e., no position). So for a winch to be able to properly configure a wire that never has been active, one has to 'report' the length of completely pulled in wire segments to the winch.

*How to create an **agxWire::Winch**, please have a look at the agxWire::WinchController section, the constructor and API are identical.*

```
// The completely pulled in wire segment.
agxWire::WireRef inactiveWire = new agxWire::Wire( inactiveWireRadius, inactiveWireResolution );
// The active wire segment that's connected to the winch.
agxWire::WireRef activeWire = new agxWire::Wire( activeWireRadius, activeWireResolution );

// Create the winch (API identical to agxWire::WireWinchController).
agxWire::WinchRef winch = new agxWire::Winch( winchRb, winchConnectionPoint, winchDirection );

// Report the inactive wire segment to the winch.
winch->add( inactiveWire, inactiveWireLength );
```

15.21.3.2. Routing with completely pulled in wire segments

The most important thing to remember when routing with inactive segments is not to forget to define the link connections. The following code snippet routes a system similar to Fig. 15.7 but with two inactive, completely pulled in, segments:

```
// The completely pulled in wire segments.
agxWire::WireRef inactiveWire1 = new agxWire::Wire( inactiveWireRadius, inactiveWireResolution );
agxWire::WireRef inactiveWire2 = new agxWire::Wire( inactiveWireRadius, inactiveWireResolution );
// The active wire segment that's connected to the winch.
agxWire::WireRef activeWire1 = new agxWire::Wire( activeWireRadius, activeWireResolution );
// The other active wire segment.
agxWire::WireRef activeWire2 = new agxWire::Wire( activeWireRadius, activeWireResolution );

// DEFINE THE CONNECTIONS:
// Inactive wire 1 is connected to inactive wire 2.
agxWire::Link::connect( inactiveWire1, wire1ConnectionPosition, link1, inactiveWire2, wire2ConnectionPosition );
// Inactive wire 2 is connected to active wire 1.
agxWire::Link::connect( inactiveWire2, wire1ConnectionPosition, link2, activeWire1, wire2ConnectionPosition );
// Active wire 1 is connected to active wire 2.
agxWire::Link::connect( activeWire1, wire1ConnectionPosition, link3, activeWire2, wire2ConnectionPosition );

// Create the winch (API identical to agxWire::WireWinchController).
agxWire::WinchRef winch = new agxWire::Winch( winchRb, winchConnectionPoint, winchDirection );

// Report the inactive wire segments to the winch.
winch->add( inactiveWire1, inactiveWire1Length );
winch->add( inactiveWire2, inactiveWire2Length );

// Begin route from winch to link2. Note that the initial pulled in
// length is how much of activeWire1 that's pulled in.
activeWire1->add( winch, initialPulledInLength );
activeWire1->add( link3 );

activeWire2->add( link3 );
activeWire2->add( new agxWire::FreeNode( wire2EndPosition ) );

// Only the active wires have to be added to the simulation.
```

```
simulation->add( activeWire1 );
simulation->add( activeWire2 );
```

Only active wires have to be added to the simulation!

15.21.3.3. Pulled in length

The agxWire::Winch and agxWire::WireWinchController return the same value for pulled in length – i.e., the amount pulled in of the current segment.

An agxWire::Winch has a few additional methods that support the inactive, totally pulled in, wire segments. Example:

```
agx::Real inactiveWireLength = 10.0;
agx::Real activeWirePulledInLength = 5.0;

// Add inactive wire, 10 m long.
winch->add( inactiveWire, inactiveWireLength );
// Add the winch to the active wire with 5 m pulled in length initially.
activeWire->add( winch, activeWirePulledInLength );

print( winch->getPulledInLength() );      // "5.0"
print( winch->getTotalPulledInLength() ); // "15.0"
print( winch->getNumPulledInWires() );    // "1"

// Remove the inactive wire.
winch->remove( inactiveWire );

print( winch->getPulledInLength() );      // "5.0"
print( winch->getTotalPulledInLength() ); // "5.0"
print( winch->getNumPulledInWires() );    // "0"
```

15.21.4. agxWire::Link:Algorithm

The link functionality is coupled to separate implementations of agxWire::Link::Algorithm's. A link algorithm receives callbacks with the link at different stages of a time step and when the link changes state from enabled to disabled, and vice versa.

The (current) default algorithms:

1. Detect approaching (sliding) nodes and spawn new algorithms given node type. E.g., from the link's point of view, when a winch is approaching.
2. Not fully pulled in or out link algorithm. The state before a segment is completely pulled in, the link is on a cylindrical joint, and this algorithm handles when the state changes to fully pulled in or out wire segments.
3. The first inactive link in a winch receives callbacks to detect when it's time to go active.
4. Enables high resolution ranges to spread over wire segments.

It's possible to implement your own or add extra (i.e., not default) link algorithms. If you are implementing your own link algorithm it's important to not store/keep a configuration dependent state. Configuration dependent states could be a wire, a start and an end node (which gives a direction). The reason for this constraint is because the link will never know if a wire has been reversed or cut in two, when a node has been removed, been merged with another wire, removed from the simulation etc..

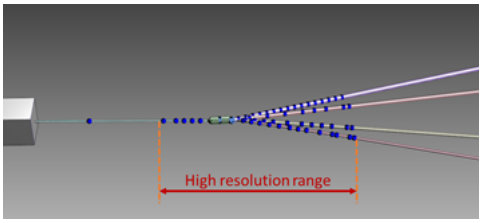


Fig. 15.8 High resolution range over link with five connections.

15.21.4.1. Optional link algorithms

For usability or performance reasons, some link algorithms are optional.

15.21.4.2. Contact stabilizing algorithm

In some cases, where the tension is high in two or more connections, the link could become unstable when in contact with other objects. This optional link algorithm monitors the tension in the connections and creates additional, stabilizing constraints when the link is in contact with another object.

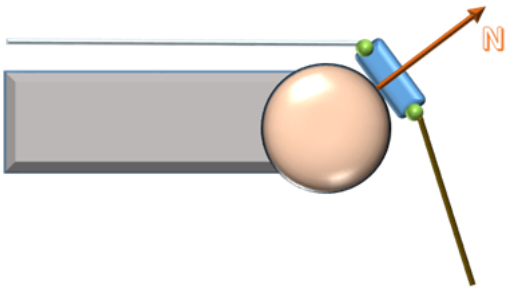


Fig. 15.9 Case where it could be suitable with a contact stabilizing algorithm.

The contact stabilizing algorithm uses its own version of a contact execute filter. It's easy to implement a filter, e.g., this filter that matches a rigid body with name 'sternRoller':

```
#include <agxWire/LinkAlgorithms.h>

class SternRollerMatcher : public agxWire::LinkObjectStabilizingAlgorithm::LinkExecuteFilter
{
  public:
    SternRollerMatcher( const agxWire::Link* link )
      : agxWire::LinkObjectStabilizingAlgorithm::LinkExecuteFilter( link ) {}

    // otherGeometry is overlapping the link - check if match.
    virtual bool matchOther( const agxCollide::Geometry* otherGeometry ) const
    {
      return otherGeometry->getRigidBody() != 0L && otherGeometry->getRigidBody()->getName() == "sternRoller";
    }
};
```

So when **matchOther** returns true AND the tension is high enough, extra constraints will be added to stabilize the link.

There are some already implemented filters. For example filters that match a property (in **agx::PropertyContainer**) in either a geometry or a rigid body. Example with geometry:

```
// Filter that checks a geometry's property container.
agxWire::LinkObjectStabilizingAlgorithm::GeometryPropertyBoolFilterRef geometryPropertyFilter =
  new agxWire::LinkObjectStabilizingAlgorithm::GeometryPropertyBoolFilter( "stabilize", link );

link->add( new agxWire::LinkObjectStabilizingAlgorithm( geometryPropertyFilter ) );

// Enables algorithm when the link is in contact with this geometry.
someGeometry->getPropertyContainer()->addPropertyBool( "stabilize", true );
```

Example with rigid body:

```
// Filter that checks a rigid body's property container.
agxWire::LinkObjectStabilizingAlgorithm::GeometryPropertyBoolFilterRef rbPropertyFilter =
  new agxWire::LinkObjectStabilizingAlgorithm::GeometryPropertyBoolFilter( "stabilize", link );

link->add( new agxWire::LinkObjectStabilizingAlgorithm( geometryPropertyFilter ) );

// Enables algorithm when the link is in contact with this rigid body.
someRigidBody->getPropertyContainer()->addPropertyBool( "stabilize", true );
```

15.21.5. Known limitations

15.21.5.1. Links and EyeNodes

A Link can not slide through an EyeNode. This is a feature that might be added in a future release of AGX Dynamics.

15.21.5.2. Loops in the configuration

A valid link-wire composition cannot have loops. There are recursive sections in the code, assuming a tree structure of the connection assembly. A link can only detect a loop if the user tries to connect the same wire twice to a link, and other than that, no runtime checks nor parsing of the current configuration to detect loops are made.

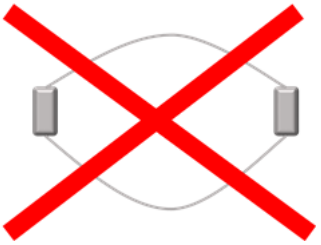


Fig. 15.10 Loops in the link connections is undefined.

This loop limitation prevents the user from creating a lasso using links and the connect-interface. But the reason for links is functionality such as pulling them into/out from winches, and pulling a lasso into a winch doesn't often make that much sense.

The more reasonable scenario when pulling a lasso into a winch is that the winch stops, due to jamming (or something similar). To achieve such behavior:

1. Connect the main wire to the link (i.e., not the lasso wire). The link should only have one mapped connection, making the link a so called 'end link'. I.e., the link defines the end of a link-wire composition.
2. Create and route the lasso wire only using explicit connections. Explicit connections could be separate bodies locked to the link body or body fixed/connecting nodes with the link body as the rigid body.

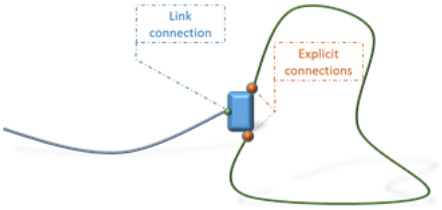


Fig. 15.11 Lasso configuration so that a winch will automatically stop when the lasso is about to be pulled in.

15.21.5.3. Length of connections and winches

The maximum distance from a connection point, on the link surface, to the link center of mass, must be less than one meter for the winches to work with the link. In principle, this means that a somewhat symmetric link can't be longer than two meters AND be pulled into or out from a winch. If winches aren't involved, there's no limitation in the connection lengths.

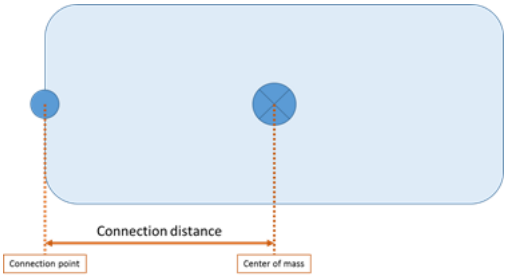


Fig. 15.12 Definition of the connection distance going from the connection point to the center of mass.

15.21.6. Controlling Link behaviour

15.21.6.1. Link and bend

Connecting wires to small link objects are often problematic when simulating offshore operations due to the wire tension to link mass ratio difference. Early, AGX have had support for such connections, supporting stable simulation with wires and links, even when the system involves extreme potential energies. The solution to the initial problem has been to have a variable stiffness of the bend constraint over the link, i.e., controlling more degrees of freedoms to filter out high frequencies.

This control of the bend stiffness has been, and will always be, handled internally. When the tension is low the bend was completely ignored making the connection look like a ball joint type of connection. I.e., the link could rotate freely about the connecting point. This type of connection behavior isn't always desirable and the user should be able to control the bend stiffness when the link isn't under extreme stress.

15.21.6.2. Link and twist

If the wire going out from a link connection defines an axis, the twist is defined to be the rotation about that axis.

Until now, this degree of freedom has been completely ignored by the wire. Mainly because there're absolutely no information about it because the wire uses 3 DOF (particle) bodies internally. Bending is still defined when the geometry of this exist given three control points along a wire.

Since twist is a completely new concept for the links, consider it experimental, and more work has to be done to figure out the API to control all desirable "effects" of twisting a link about a wire.

15.21.7. Introducing interface to a Link Node and Connection Properties

15.21.7.1. agxWire::ILinkNode

This new node is only an interface to the implementation of the special node used when connecting an agxWire::Wire to an agxWire::Link. At this moment, the interface is minimalistic but can be extended easily for future features.

```
/**
Interface class for nodes connected to links.
*/
class AGXPHYSICS_EXPORT ILinkNode : public agxWire::ConnectingNode
{
  public:
    /**
    \return the connection properties for the connection between the wire and the link
    */
    virtual agxWire::ILinkNode::ConnectionProperties* getConnectionProperties() const = 0;
};
```

15.21.7.2. agxWire::ILinkNode::ConnectionProperties

New object holding parameters and properties of a connection between a wire and a link. Currently it's small and it'll be extended/changed when we fully understand these connections.

```
/**
Object holding parameters, such as bend and twist stiffness,
related to wire to link connections.
*/
class AGXPHYSICS_EXPORT ConnectionProperties
{
  public:
    /**
    Default constructor, with default bend and twist stiffness disabled.
    */
    ConnectionProperties();

    /**
    Construct given bend- and twist stiffness.
    */
    ConnectionProperties( agx::Real bendStiffness, agx::Real twistStiffness );

    /**
    Assign bend stiffness of the link to wire connection. The bend stiffness is
    interpreted as Young's modulus of the wire. I.e., the final stiffness
    is dependent on the wire radius and segment lengths. Default: 0.
    \param bendStiffless - new bend stiffness for this connection
    */
    void setBendStiffness( agx::Real bendStiffness );

    /**
    \return the currently used bend stiffness of this connection (default: 0)
    */
    agx::Real getBendStiffness() const;

    /**
    Assign twist stiffness of the link to wire connection. The twist stiffness is
    interpreted as Young's modulus of the wire. I.e., the final stiffness
    is dependent on the wire radius. Currently, the segment length is assumed
    to be 1 meter and default value of the stiffness is 0.
    \param twistStiffness - new twist stiffness for this connection
    */
    void setTwistStiffness( agx::Real twistStiffness );

    /**
    \return the currently used twist stiffness of this connection (default: 0)
    */
```

```
    agx::Real getTwistStiffness() const;
};
```

Bend- and twist stiffness may be any real value >= 0. By default the values are all 0 to maintain the old behavior.

15.21.7.3. Default connections

Links carries a default connection property. I.e., all new connections inherits the values of this default property.

```
/**
Object that defines the relation between different types of wires (agxWire::Wire). I.e.,
this link object enables functionality to (for example) move nodes from one wire to
another.
*/
class AGXPHYSICS_EXPORT Link : public agx::Referenced, public agxStream::Serializable
{
  public:
    ...

    /**
    Access default wire connection properties for each new wire that's connected to this link.
    \note If any value is changed the current connections will not be updated with this value.
          Only new connection will receive this new default value.
    \return default connection properties
    */
    agxWire::ILinkNode::ConnectionProperties* getDefaultWireConnectionProperties();

    /**
    \return default connection properties
    */
    const agxWire::ILinkNode::ConnectionProperties* getDefaultWireConnectionProperties() const;

    ...
}
```

As stated in the API documentation, the changes are not propagated to all current connections. It'll only affect new connections, making it possible to have a default look, but custom if e.g., a chain, instead of a wire, is connected.

15.21.8. Twist again

The implementation of the link to wire twist constraint is basically a hinge axis. I.e., it's possible to control it within a range, motorize it or lock at certain angle. You'll have to test and see what the desirable behavior should be.

Currently there's no interface to change these behaviors. Default is a motor at zero speed, i.e., the twist stiffness controls the inertia to rotate the link about the twist axis.

15.21.9. Small Example

Example to change bend stiffness of individual nodes where the wires have identical radius.

```
void createScene( agxSDK::Simulation* simulation )
{
  const agx::Real radius = 0.015;

  agxWire::WireRef wire1 = new agxWire::Wire( radius, 2.0 );
  agxWire::LinkRef link1 = new agxWire::Link( createLink( simulation ) );

  agxWire::WireRef wire2 = new agxWire::Wire( radius, 2.0 );
  agxWire::LinkRef link2 = new agxWire::Link( createLink( simulation ) );

  agxWire::WireRef wire3 = new agxWire::Wire( radius, 2.0 );
  agxWire::LinkRef link3 = new agxWire::Link( createLink( simulation ) );

  link1->getRigidBody()->setPosition( 0, -1, 0 );
  link2->getRigidBody()->setPosition( 0,  0, 0 );
  link3->getRigidBody()->setPosition( 0,  1, 0 );

  simulation->add( new agx::LockJoint( link1->getRigidBody() ) );
```

```
  simulation->add( new agx::LockJoint( link2->getRigidBody() ) );
  simulation->add( new agx::LockJoint( link3->getRigidBody() ) );

  wire1->add( link1, agx::Vec3( 0.15, 0, 0 ) );
  wire1->add( new agxWire::FreeNode( 3, -1, 0 ) );

  wire2->add( link2, agx::Vec3( 0.15, 0, 0 ) );
  wire2->add( new agxWire::FreeNode( 3, 0, 0 ) );

  wire3->add( link3, agx::Vec3( 0.15, 0, 0 ) );
  wire3->add( new agxWire::FreeNode( 3, 1, 0 ) );

  simulation->add( wire1 );
  simulation->add( wire2 );
  simulation->add( wire3 );

  agxWire::ILinkNode* link1Node = link1->getConnectingNode( wire1 );
  agxWire::ILinkNode* link2Node = link2->getConnectingNode( wire2 );
  agxWire::ILinkNode* link3Node = link3->getConnectingNode( wire3 );

  // Closest wire in the picture.
  link1Node->getConnectionProperties()->setBendStiffness( 1.0E12 );
  // Middle wire.
  link2Node->getConnectionProperties()->setBendStiffness( 2.0E10 );
  // Farthermost wire.
  link3Node->getConnectionProperties()->setBendStiffness( 1.0E8 );
}
```

Result after some time. Wire 3 is swinging back and forth but the others are not moving.

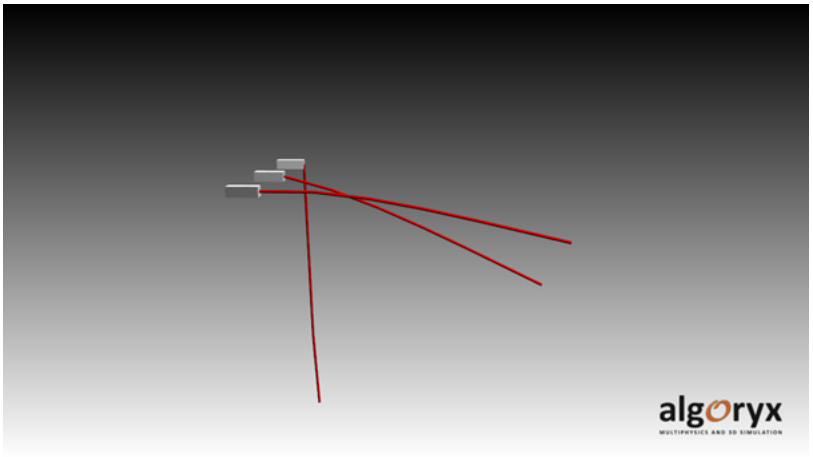


Fig. 15.13 Three links and wires where the wires has identical radius but the connection bend stiffness differs.

Another example where all wires has different radius and the links has identical default connection properties:

```
void createScene( agxSDK::Simulation* simulation )
{
  const agx::Real radius1 = 0.050;
  const agx::Real radius2 = 0.025;
  const agx::Real radius3 = 0.010;

  agxWire::WireRef wire1 = new agxWire::Wire( radius1, 2.0 );
  agxWire::LinkRef link1 = new agxWire::Link( createLink( simulation ) );

  agxWire::WireRef wire2 = new agxWire::Wire( radius2, 2.0 );
  agxWire::LinkRef link2 = new agxWire::Link( createLink( simulation ) );

  agxWire::WireRef wire3 = new agxWire::Wire( radius3, 2.0 );
  agxWire::LinkRef link3 = new agxWire::Link( createLink( simulation ) );

  // Before making any connections, set default bend stiffness.

  // Closest wire in the picture.
```

```
  link1->getDefaultConnectionProperties()->setBendStiffness( 1.0E10 );
  // Middle wire.
  link2->getDefaultConnectionProperties()->setBendStiffness( 1.0E10 );
  // Farthest wire.
  link3->getDefaultConnectionProperties()->setBendStiffness( 1.0E10 );

  link1->getRigidBody()->setPosition( 0, -1, 0 );
  link2->getRigidBody()->setPosition( 0,  0, 0 );
  link3->getRigidBody()->setPosition( 0,  1, 0 );

  simulation->add( new agx::LockJoint( link1->getRigidBody() ) );
  simulation->add( new agx::LockJoint( link2->getRigidBody() ) );
  simulation->add( new agx::LockJoint( link3->getRigidBody() ) );

  wire1->add( link1, agx::Vec3( 0.15, 0, 0 ) );
  wire1->add( new agxWire::FreeNode( 3, -1, 0 ) );

  wire2->add( link2, agx::Vec3( 0.15, 0, 0 ) );
  wire2->add( new agxWire::FreeNode( 3, 0, 0 ) );

  wire3->add( link3, agx::Vec3( 0.15, 0, 0 ) );
  wire3->add( new agxWire::FreeNode( 3, 1, 0 ) );

  simulation->add( wire1 );
  simulation->add( wire2 );
  simulation->add( wire3 );
}
```

Resulting in this steady state:

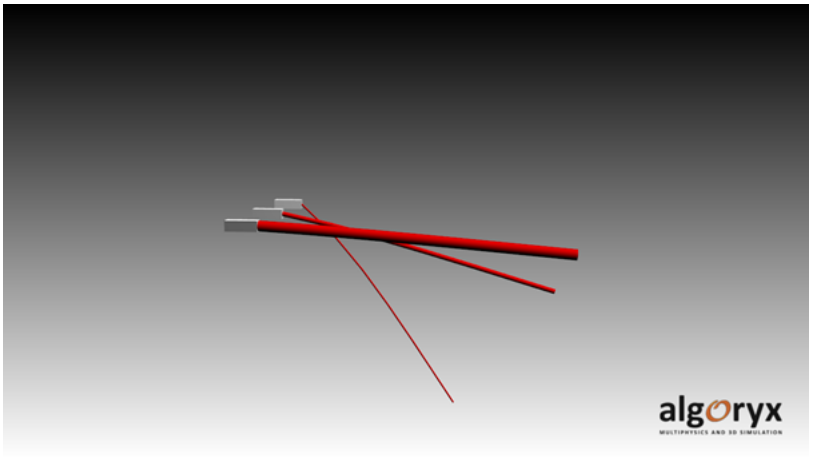


Fig. 15.14 Three links and wires where the connection bend stiffness is identical but the wires has different radii.

A trained eye can see that the relation between the radius and stiffness is proportional to r^4.

16. AGX Cable

The agxCable module is used to simulate cables, hoses, ropes, short wires and dress packs. These are long structures with a circular cross section that can be bent, stretched and twisted. The agxCable namespace provides the Cable class which uses a lumped elements approach to model such structures. By using **agxCable::Cable** it is possible to create simulations with cable structures that behave realistically in contacts and interaction with other simulated objects. This can be used, for example, for training operations of assembling scenarios, measurement of load and wear on cables in a robot scenario or for cable motion analysis in articulated devices.

16.1. Model

The lumped elements approach means that the cable is represented by a sequence of rigid bodies linked together using constraints. Each rigid body constitutes what is called a segment. Together the segments define the shape, extent and location of the cable. In addition to a rigid body, each segment has a geometry that allows it to collide with other geometries in the simulation, including other segments of the same cable. Each geometry contains a capsule and consecutive segments are placed with a bit of overlap so that the flat sides of facing hemispheres coincide in a straight cable.



Fig. 16.1 Two consecutive cable segments. Notice the overlap.

This makes the cable surface smoother at bends, compared to if cylinders had been used. The geometries for consecutive segments are exempted from contact generation. The cable is a one-dimensional structure and thus each constraint connects exactly two segments.

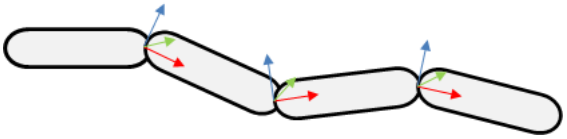


Fig. 16.2 A Cable consisting of four segments and three LockJoints (constraints).

Parameters and properties set on the cable are translated into parameters on these bodies and constraints. This makes it possible to create cables that range from easily bendable to completely stiff.

Both the bodies and the constraints operate on all six degrees of freedom, which makes it possible to track bending, twisting and stretching of the cable independently.

16.2. Cable vs Wire

agxWire is another very efficient AGX component that is used to simulate long, bendable structures. Whereas the cable has a *fixed resolution*, meaning that the number and size of the segments remain constant throughout the simulation, the wire has *dynamic resolution*. This is the main difference compared to agxWire, which can adjust the resolution locally in order to improve stability. This makes it possible to use agxWire in scenarios with extreme tension and in large scale scenarios with several kilometers of wire. On the other hand, the agxCable model supports modeling of plasticity and torsion, which are not available for agxWire. The wires, but not the cables, can be cut and merged, and also spooled in and out of winches.

Use Wire if you want to simulate:

- Very long wires/chains/ropes (50m or more) (Anchoring, hoisting, crane wires) with varying length.
- Wire winches (winching wire in and out)
- Be able to withstand high tension/large mass ratios
- Torsion is not relevant
- Typical scenarios: crane operations, lifting heavy containers, anchoring oil rigs, towing ships

Use Cable if you want to simulate:

- Hoses, cables, pipes of shorter distances with fixed length
- Limited tension
- High fidelity self interaction (collision)
- Requires torsion
- Requires plasticity
- Typical scenarios: cables on an industrial robot, hoses/cables in an assembly simulation of a car engine.

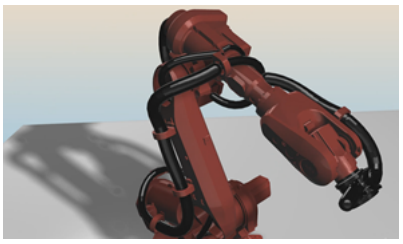


Fig. 16.3 Industrial robot with dresspack (cables) attached.

16.3. Creating a Cable

There are two parts to creating a cable: routing and properties configuration.

16.3.1. Routing

Cable routing is the process that defines the initial path of the cable. The user specifies the path using routing nodes. This section describe the default routing and segmentation algorithm, alternative algorithms are described in Direct segment routing and Path routing. A cable created from a collection of routing nodes will extend from node to node in the order that the nodes were added to the cable. Each such routing node pair is called a leg of the path. There are two types of routing nodes: FreeNode and BodyFixedNode. A FreeNode simply defines a location in space that the cable should pass through. This type of node produces no special behavior once the simulation has started.

```
agx::Real radius(0.05);
agx::Real resolution(3.35);
agxCable::CableRef cable = new agxCable::Cable(radius, resolution);
// Add 3 points in world coordinate system
cable->add(new agxCable::FreeNode(0.0, 0.0, 0.0));
cable->add(new agxCable::FreeNode(1.0, 0.5, 0.0));
cable->add(new agxCable::FreeNode(2.0, 0.3, 0.0));
```

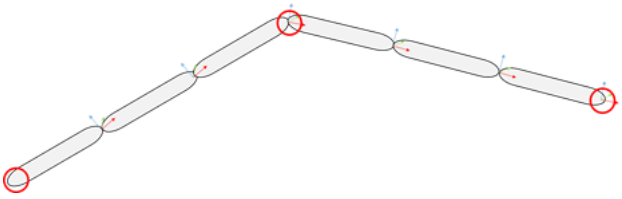


Fig. 16.4 Cable created from three FreeNodes indicated by red circles.

The points supplied defines the range of the cable in space, but does not include the capsule hemispheres. The cable will therefore be one cable diameter longer than the total distance between the routing nodes.

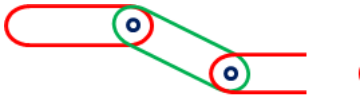


Fig. 16.5 Three cable segments created from three routing nodes.

The extra extent of the cable becomes important when an endpoint of the cable should be on the surface of some object. As shown in Fig. 16.6: Cable routed from the surface of a wall., placing the routing node on the surface will cause some initial overlap with the object. This will result in contacts and sudden cable movement when the simulation is started. Two possible solutions are to either place the routing node one cable radii away from the surface, or to disable collisions between the object and the first cable segment.

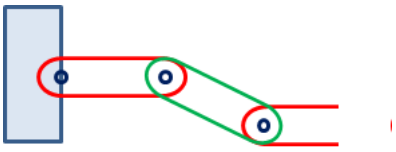


Fig. 16.6 Cable routed from the surface of a wall.

BodyFixedNodes are used to attach the cable to some other rigid body so the cable and the body will follow each other as the simulation progresses. The BodyFixedNode can be configured to either allow or prevent the body from rotating relative to the cable by providing different arguments to the BodyFixedNode constructor. By providing only an offset, in the form of an agx::Vec3, we specify that only that point on the body, in the body's model coordinate frame, should be attached to the cable with a Ball Joint. By providing a full transformation, in the form of an agx::AffineMatrix4x4, we specify how the whole body should be placed relative to the cable using a LockJoint.

```
agx::RigidBodyRef rotatingBody = new agx::RigidBody();
agx::RigidBodyRef fixedBody = new agx::RigidBody();

// Attach the rotatingBody with a point: results in a BallJoint attachment constraint
cable->add(new agxCable::BodyFixedNode(rotatingBody, agx::Vec3()));

// Attach the rotatingBody with an AffineMatrix4x4: results in a LockJoint attachment constraint
cable->add(new agxCable::BodyFixedNode(fixedBody, agx::AffineMatrix4x4()));
```

When using a transformation matrix, the transformation describes where the routing node should be placed and how the body should be oriented relative to the cable, where the Z axis is defined to be along the cable.

Table 16.1 Cable routing node types.

Class	Description
agxCable::FreeNode	Initial positioning of cable.
agxCable::BodyFixedNode	Attach cable to rigid body.

The routing phase ends when the cable is added to a simulation. At this point the routing is finalized and segmentation is performed. Segmentation is the process that creates and positions segments between the routing nodes. The routing nodes are strictly adhered to, meaning that, except for at the cable end points, the routing node defines the end of one segment and the start of another. It is the responsibility of the segmentation algorithm to find a segment length that as evenly as possible divides all the route node pairs. This makes the smallest inter-node distance a lower bound on the cable's resolution. The user supplied cable resolution is used as a start point for the search. It is not always possible to find a segment length that perfectly divides all path legs. In that case, some of the path legs will contain either small overlaps or small extra gaps. The intention is that these will be evened out during the first few time steps of the simulation. For this to work there must be some room for the cable to move. If two BodyFixedNodes are added next to each other then neither side can move and the cable may become unstable.

It is possible to dry-run the segmentation algorithm in order to gather information about the current routing without finalizing it. This is done by calling tryInitialize on the cable and returned is an initialization report that contains information such as the number of nodes created, the resolution of the cable, the largest segment error and the total length of the cable.

Additional routing nodes cannot be added to the cable after routing has been finalized. However, as an alternative to BodyFixedNodes the cable provides attachments that provide similar functionality. By attaching a rigid body to a cable segment the body will follow the cable as the simulation progresses. Attaching bodies this way follows the same translation/transformation rules as the BodyFixedNode. Note that adding attachments does not change the path of the cable, or the position of the attached object.

Attaching bodies requires a way to specify where along the cable the body should be attached. This is done using cable iterators, which are described in the Section 16.9.

```
agx::RigidBodyRef attachedBody = ...;
agxCable::CableIterator segment = ...;
// Attach a body with a BallJoint to the segment.
cable->attach(segment, attachedBody, agx::Vec3());
```

16.4. Direct segment routing

Direct segment routing is an an alternative to the segmentation process described in Section 16.3.1. Using direct segment routing it is possible to precisely control both the position and orientation of each individual cable segment during routing. This is useful when the cable path and segmentation is defined by an external tool or is known though some other means.

A cable with direct routing support is created by passing an instance of `agxCable::IdentityRoute` to the cable constructor instead of the resolution parameter. The routing nodes added to such a cable describe not only the path of the cable, but also the position and orientation of each and every segment. No additional segments will be created, and every segment will correspond to a routing node.

The transformation of the segments are configured by setting the transformation of the RigidBody that the routing node contains.

The segmentation algorithm handles the last routing node separately from the rest of the nodes. There is no such special handling when using direct segment routing. Just like every other segment, the last segment is placed with its start at the location of the corresponding routing node. When using the segmentation algorithm the last segment is placed with its end at the last routing node.

```
const agx::Real radius = agx::Real(0.04);
const agx::Real resolution = agx::Real(3.0);
agxCable::CableRef cable = new agxCable::Cable(radius, new agxCable::IdentityRoute(resolution));

agxCable::FreeNodeRef node1 = new agxCable::FreeNode(pos1);
node1->getRigidBody()->setRotation(rot1);
cable->add(node1);
```

```
agxCable::FreeNodeRef node2 = new agxCable::FreeNode(pos1);
node2->getRigidBody()->setRotation(rot2);
cable->add(node2);

agxCable::BodyFixedNodeRef node3 = new agxCable::BodyFixedNode(attachedBody, attachmentPointInBodyFrame);
node3->getRigidBody()->setPosition(pos3);
node3->getRigidBody()->setRotation(rot3);
cable->add(node3);
```

16.5. Path routing

Path routing is an alternative to the segmentation process described in Routing. Path routing work similarly to the default segmentation algorithm, but relaxes the requirements on individual segment placing. Instead of a sequence of routing legs the routing nodes are seen as a continuous path along which segments are placed.

16.6. Rebind

By default, a routed cable will try to straighten it self out, i.e. the resting state is a straight cable. But with the `Cable::rebind()` method, the current/initial state can become the resting state. Consider figure Fig. 16.7 where a cable is created as a spiral. By calling the rebind method, the current state of the cable will become the resting state. Any effort to straighten it out, will result in a restoration force to bring the cable back to its resting state. The example below is available as a python script in `<agx-dir>/data/python/torsionSpiral.agxPy`.

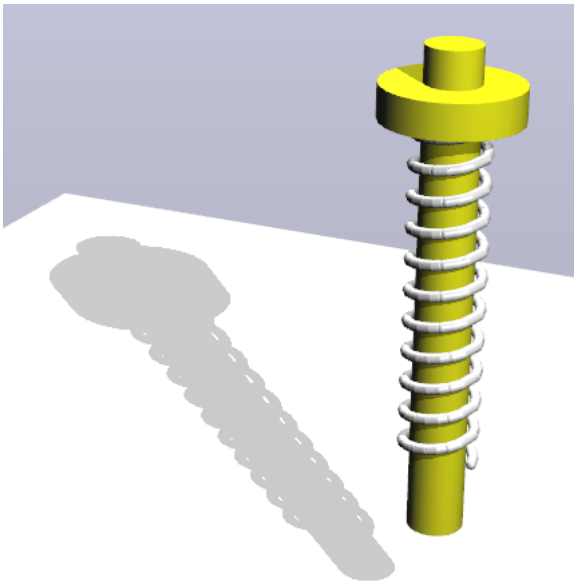


Fig. 16.7 Cable routed as a spiral using the rebind() method.

16.7. Properties

The cable provides several properties that control its behavior. These properties are accessed and modified via the `agxCable::CableProperties` class accessible using `Cable::getCableProperties`. Through the properties handle we can set things such as Young's modulus and damping. Several cables can share the same properties handle, making it possible to easily configure and modify many cables at once.

Property name	Unit	Description
Young's modulus	Pressure	Controls the stiffness of the cable.
Poisson's ratio	unitless	Ratio of Stress over Strain. Will affect rotational stiffness when twisting.
Damping	Time	Controls how quickly potential energy in cable deformations dissipates.

The parameters can be set independently along the three dimensions of the cable: bend, twist and stretch. The following code snippet demonstrates how to set Young's modulus in bend, twist and stretch directions.

```
agxCable::CableProperties* properties = cable->getCableProperties();
properties->setYoungsModulus(1e7, agxCable::BEND);
```

```
properties->setYoungsModulus(1e7, agxCable::TWIST);
properties->setYoungsModulus(1e7, agxCable::STRETCH);
```

Attention

Care should be taken when simulating cables with very low Young's modulus in the stretch direction. The cable geometries are not resized when the cable is deformed leading to gaps in the cable during extreme elongation.

A different set of properties of the cable are those that all geometries have and that are represented by an instance of the agx::Material class. By assigning a material to a cable all geometries that make up that cable are assigned that same material. This, along with the contact materials created for the material assigned, control much of how the cable interacts with other objects in the scene. Some parameters share the same name as parameters found in the cable properties. Despite this, these are separate values with different purposes. For example, the Young's modulus in the the geometry material control stiffness in contacts while the cable property with the same name control stiffness within the cable itself.

The density set in the geometry material is used to compute the mass of each cable segment, but due to the overlap between consecutive segments the total mass of the cable will be higher than what a density times volume computation would give. A higher resolution cable will have more segments and thus more overlaps and a larger mass discrepancy.

In addition to the internal damping in the cable, it is possible to set both linear and angular velocity damping of the cable using setLinearVelocityDamping and setAngularVelocityDamping.

16.7.1. Plasticity

A cable can be given plastic deformation properties by adding a *CablePlasticity* component to it. The plasticity parameters are configured on the *CablePlasticity* object using the same direction parameters as when configuring other direction dependent cable parameters.

```
agxCable::CableRef cable = ...;
agxCable::CablePlasticityRef plasticity = new CablePlasticity();
plasticity->setYieldPoint(1e9, agxCable::BEND);
cable->addComponent(plasticity);
```

The yield point specifies the torque required to permanently deform the cable, i.e., to cause a plastic deformation. Deformations in the stretch direction are always elastic.

16.8. Preventing tunneling using *CableTunnelingGuard*

If a simulation has issues with cables tunneling through each other, a direct solution is to increase the simulation frequency. In cases where this is not feasible, there is a *CableTunnelingGuard* component which can be added to one or multiple of the problematic cables to improve this behavior.

```
agxCable::CableRef cableExhibitingTunneling = ...;
cableExhibitingTunneling->addComponent(new CableTunnelingGuard(4.0));
```

This component creates a shell of sensors around the cable whose contacts are used for creating additional contacts between cable segments, attempting to predict and take into consideration a future contact.

Note

The component only takes other *CableTunnelingGuard* instances and the original cable geometries in account. *CableTunnelingGuard* interactions with additional geometries or shapes is not currently supported.

Some parameters are exposed that help further reduce tunneling behavior.

Parameter Name	Unit	Description	Default Value
Hull Scale	unitless	Controls the size of the surrounding sensor shell by applying the scale to the cables segment geometries.	None
Angle Threshold	Radians	Controls the minimum angle to the length axis of a segment at which a contact is considered.	PI/2*0.9
Leniency	unitless	Affects the required estimated depth for creating a supporting contact, calculated as otherCablesRadius * 0.5 * (1 - leniency) considering the symmetric nature of penetration depth.	0
Debounce Steps	amount	Controls the amount of steps for which the component continues to create geometry contacts after one has been created under normal conditions.	0

The component can also be configured to always add contacts while in range of another cable using *setAlwaysAdd* although this may lead to unexpected behavior in some situations, so using *setDebounceSteps* to temporally limit this behavior is instead preferred.

16.8.1. Interaction between *CableTunnelingGuard* instances

When the hull of two separate guards (or distant enough hulls along the same cable with a guard) come in contact, they are handled the same way as hull-segment contacts. This effectively means that the hull radius of these interactions will be as if one of the cable's guard had the combined hull radius of both the cable's guards.

Note

When two guards interact, their parameters are mediated to ensure consistent behavior. Besides the hull radius, the angle threshold of each guard is used for the respective angle checks, and only the maximum leniency and debounce are used.

A *CableTunnelingGuard* can also interact with itself to predict self-collisions between segments in a cable. Collisions between naturally overlapping hulls are disabled on creation, but other collisions between a cable's hulls do interact to prevent tunneling. To disable self-interaction, the *setSelfInteractionEnabled* method can be used.

16.9. Inspection using iterators

Some state of the cable, such as resolution and properties, apply to the whole cable while others, such as tension, vary along the cable. To inspect the latter the cable provides an iterator class that is used to iterate over the segments of the cable. Segmentation must have been performed on the cable before any iterators can be created. Using the iterator, we can find information such as the position of the segment, the rotational or translational tension and also get a hold of both the rigid body and the geometry that is used to represent the segment in the simulation.

```
for (agxCable::CableIterator segment = cable->begin();
!segment.isEnd();
++segment)
{
  // Use the accessor methods provided by CableIterator to inspect
  // dynamic cable state.
  agx::Real tension = segment->getStretchTension();
  agx::RigidBody* body = segment->getRigidBody();
  agxCollide::Geometry* geometry = segment->getGeometry();
}
```

The cable also supports range based for loops:

```
for (auto segment : *cable)
{
}
```

Once the cable has been initialized it is possible to create cable iterators directly from the routing nodes. This makes it possible to quickly get at the segment created near a particular node. This can be used, for example, to track the motion of some interesting point along the cable, or to find the geometry for a particular part of the cable in order to disable collisions between that geometry and geometries held by a body attached to the cable.

```
agxCable::CableRef cable = ...;
agx::RigidBodyRef body = ...;
agxCable::BodyFixedNodeRef node = new agxCable::BodyFixedNode(body, agx::Vec3());
cable->add(node);
simulation->add(cable);
agxCable::CableIterator segment(attachedNode);
agxUtil::setEnableCollisions(segment->getGeometry(), body, false);
```

A geometry, for example received as an argument to a contact event listener, can be passed to the static **Cable::getCableForGeometry** function and if that geometry is part of a cable then a pointer to that cable will be returned.

```
agxCollide::Geometry* geometry = ...;
// Return the Cable for which the geometry is part of. nullptr if the geometry is not
// part of any Cable.
agxCable::Cable* cable = agxCable::Cable::getCableForGeometry(geometry);
```

An initialized cable has two lengths: the rest length and the current length. The rest length is the sum och the lengths of the segments that make up the cable and the current length is the sum of the distances between the starting positions of consecutive segments, and the end position of the last segment. If there is no tension then the two lengths are the same, but if there is some load on the cable then the two lengths may differ. We say that the cable has become either stretched or compressed. The amount of stretch a given load produces is controlled by configuring Young's modulus on the cable.

16.10. Hydro- and aerodynamics

A cable is affected by the fluid that surrounds it. A WindAndWaterController can be used to simulate the affects in water, air and other fluids, see Section 21.

16.11. Known limitations

- Mass not properly computed from density because of hemisphere overlaps.
- Geometries are not resized to match the current stretching of the cable.
- Cables must be circular.
- Cables must be homogeneous, both in size and properties.

17. AGX Cable damage

An important factor in cable application design is cable wear. The cable damage component of the agxCable module provides a way to estimate damage caused to the cable during the simulation. Effects such as tension, deformation, and contacts are taken into account by the damage estimation model and applied to each cable segment individually.

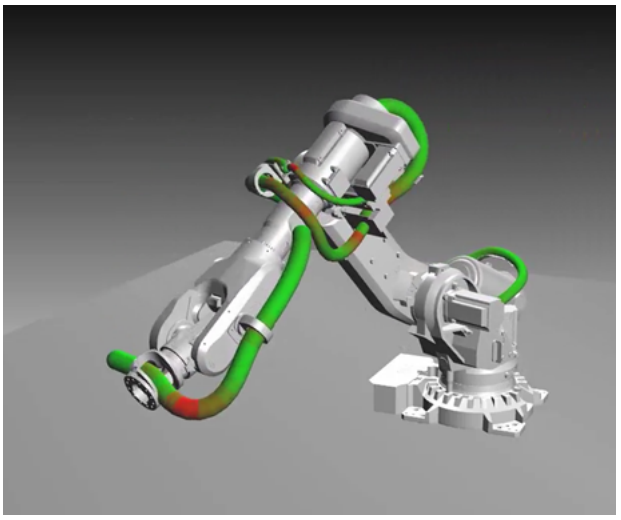


Fig. 17.1 Robot with several cables. Current cable damage visualized by color.

17.1. Basic usage

The damage estimation is performed by creating an instance of the `agxCable::CableDamage` class. It uses the simulation result of a cable to estimate damage and wear inflicted on the cable. The following code should be executed before the simulation starts to gather damage estimation for a cable named cable.

```
agxCable::CableDamageRef damage = new agxCable::CableDamage();
cable->addComponent(damage);
```

During or after the simulation the estimated damages can be accessed via instances of the *SegmentDamage* class, which the* CableDamage* instance provides. Damage is estimated per cable segment, and there is one *SegmentDamage* instance per segment of the simulated cable. We can read the damage accumulated over the simulation so far for a particular cable segment with the following code.

```
agx::Real wear = cableDamage->getAccumulatedDamageAt(segmentIndex).total();
```

The damage inflicted to a particular cable segment during the most recent simulation step is accessed using the following code.

```
agx::Real newWear = cableDamage->getCurrentDamageAt(segmentIndex).total();
```

In addition to the total damage, it is possible to read the contribution from the individual damage estimation sources for a particular segment. To better understand the available sources we must first understand the damage estimation model.

17.2. Damage estimation model

Cable damage is estimated each time step from a number of contribution sources. Each source is scaled according to a weight associated with the source and the weighted values are summed to form a total damage contribution for that time step.

The sources are categorized into four groups: deformation, deformation rate, tension, and contact force. The first three groups are further categorized based on the direction of deformation (or tension): bend, twist, and stretch. The fourth group, contact force, is split in normal force and friction force. This results in a total of eleven (3x3+1x2) damage estimation sources.

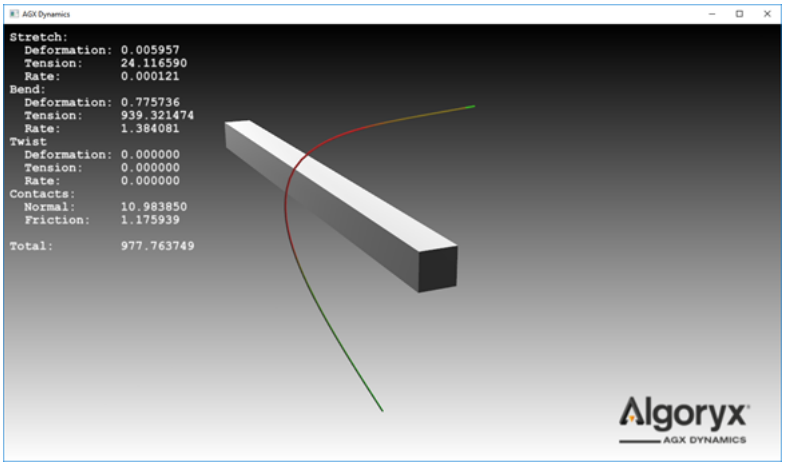
Deformation, deformation rate and tension have a threshold each, below which a deformation/tension gives no contribution to the damage estimation.

The complete damage estimation formula is as follows:

$$d = W_{T_b} T_b^2 + (W_{R_b} R_b + W_{b_0}) \max(0, b - b_0)^2$$
$$+W_{T_t} T_t^2 + (W_{R_t} R_t + W_{t_0}) \max(0, t - t_0)^2$$
$$+W_{T_s} T_s^2 + (W_{R_s} R_s + W_{s_0}) \max(0, s - s_0)^2$$
$$+W_n F_n^2 + W_f F_f^2$$

W	Weighting factor
T	Tension
R	Deformation rate
F	Contact force
b	Along bend direction
t	Along twist direction
s	Along stretch direction
n	Along contact normal
f	Along contact tangent
0	Deformation threshold

The following image shows the contribution of the different sources:



Damage estimation is tracked per cable segment. Dynamic cable properties that are defined as a relationship between two consecutive cable segments, such as tension and deformation, are distributed evenly over the two segments and thus the segment's damage contribution becomes the average of the damage contributions at the segment's two end points.

17.3. Parameters

The cable damage model contains two types of parameters: weights and thresholds. The weights control how much each individual damage estimation source should contribute to the final total. The thresholds control the smallest deformation required for the deformation related damage estimation sources to take effect. The following code shows an example of how to set some of these parameters. The damage estimation model section details the available weights and thresholds.

```
damage->setStretchDeformationWeight(100.0);
damage->setTwistDeformationWeight(30.0));
damage->setBendRateWeight(10);
damage->setTwistTensionWeight(0.04);
damage->setFrictionForceWeight(2.0);
damage->setStretchThreshold(0.0);
damage->setBendThreshold(0.1);
```

The patterns are *set[Bend|Twist|Stretch][Deformation|Rate|Tension]Weight*, *set[Normal|Friction]ForceWeight*, **and** *set[Bend|Twist|Stretch]Threshold*.

Changing the parameters will affect simulation steps taken after the change, but not damage estimates for time steps already completed. This makes it possible to alter the cable damage parameters over time to account for changes to the cable or the environment during the simulation.

The cable damage calculations and its parameters do not interact in any way with the cable material parameters and accumulated cable damage does not influence the simulation behavior of the cable.

17.4. Tutorial

The C++ tutorial *tutorial_cable_damage.cpp* shows more details on how to set up a scene with cable damage.

18. AGX PowerLine

In AGX Dynamics, a power line represents a collection of connected components that transport power across a system, typically from some kind of source to a set of consumers. Examples of power lines are a drive train moving torque from an engine to the wheels of a car or a truck, or a hydraulic circuit converting pressure from a pump into linear motion of a hydraulic cylinder.

Despite the name the structure of a power line does not need to be a simple line, but arbitrary graphs are supported. Using converter components, it is also possible to mix hydraulic and drivetrain components in the same power line. While the `agxPowerLine` namespace contains some basic components, most of the functionality is provided by the `agxDriveTrain` and `agxHydraulics` namespaces.

Using the AGX PowerLine library it is possible to share a power source among several consumers and thereby include the various system failures that can occur due to insufficient available power in the simulation.

18.1. Design philosophy

The most basic building blocks of the AGX power line are the one-dimensional bodies that represent the motion within each of the constituent components. For a drivetrain the motion is the rotation of the various shafts and gears, and for a hydraulic system it is the motion of fluid through pipes and valves. The toolkit can also represent linear motion. We call these the three dimension types. The one-dimensional bodies are connected together when the power line components are connected, which causes motion in one body to affect the neighboring bodies. The power line itself imposes no limitation in how components are connected, arbitrary graphs are allowed, but particular hydraulic or drivetrain components may limit how that particular component may be connected to other components.

Every component has an input and an output side. The sides may differ both in the number of connections that the side allows, and the type of dimension that it may connect to. The sides at which two components are connected define the direction of motion where output-to-input is the default and corresponds to positive-to-positive motion. Connecting output-to-output or input-to-input causes the two components to move in opposite local directions.

The power propagation can cross between the one-dimensional power line world and the six-dimensional mechanical world at specified transition points. A transition point is a pair consisting of an AGX constraint such as a hinge or a prismatic and a type of power line component called an Actuator.

The various components that make up a power line are owned by a particular PowerLine instance, which in turn is owned by the Simulation. To simulate a power line one first creates the PowerLine object and a collection of components. The components are added to the PowerLine and connected together, while the PowerLine is added to the Simulation.

18.2. Component types

The components that take part in a power line in AGX Dynamice are of one of two types: Unit or Connector. Every class representing a drivetrain or hydraulic component inherit from one of these two classes, either directly or indirectly. The two types are fundamentally different internally, but are used in similar ways. The create-add-connect pattern applies to both, but see the section titled “Connecting components” for more details.

18.2.1. Unit

A Unit holds the one-dimensional bodies that represents motion within the power line. The bodies are actually owned by instances of subclasses of the `PhysicalDimension` class; either `RotationalDimension`, `TranslationalDimension` or `FlowDimension` depending on which dimension type the Unit contains. The PhysicalDimension exposes the rotational-, translational or flow velocities in the component in a uniform, one-dimensional way. This is called the gradient of the PhysicalDimension. A PhysicalDimension also has a mass property that relates the load type of the dimension, e.g., force, torque or pressure, to acceleration within the Unit. The mass property is simply the mass or inertia in the case of translational and rotational dimensions, respectively, but more complicated for a hydraulic dimension.

The Unit defines connection points into the Unit. A connection point is a PhysicalDimension that other components may connect to. A Unit may have multiple connection points and may return a different PhysicalDimension instance depending on whether the connection concerns the input or output side.

18.2.2. Connector

A Connector represents transfer of power between Units. Whenever two or more Units are connected a Connector is used to link them together. It does this by creating a constraint between the bodies held by the PhysicalDimensions in the connected Units. The specifics of the constraint define the operation of the Connector and the Connector may provide configuration options that are used to control the constraint parameters. The configuration options may have been given component-specific names in order to be more domain oriented.

A Connector does not generally have any PhysicalDimensions of its own.

Much like the Unit, the Connector defines connection points for its inputs and outputs. The Connector specifies which type of dimension may be connected to each of the two sides. A connection between a Connector and a Unit can only be made if there are matching connection points for the sides being connected. More on making connections in the section titled “Connecting components”.

18.2.3. Composite components

Some components are more complicated than what a regular Unit or Connector can represent. In such cases the toolkit provides composite components, which are Units and Connectors that contains other components. All components have a primary type, so even if a composite component may contain both Units and Connectors, the component itself is always either a Unit or a Connector and can be used as-if it was a regular Unit or Connector.

A composite Unit may be empty in the sense that it doesn't contain any PhysicalDimensions of its own. Instead it contains internal Units that provide them. When asked for its connection point the Unit will return one of the PhysicalDimensions of the internal Units. A similar pattern may be used for Connectors and constraints.

Composite components commonly provide accessor methods for the internal components. Care should be taken when manipulating such components so that any invariants maintained by the composite component isn't broken. In general, connect calls involving internal components are advised against, unless explicitly suggested by the composite component's documentation.

18.2.4. Actuator

An Actuator is a type of Unit that forms a bridge between the one-dimensional power line and the six-dimensional mechanical world within the simulation. Examples of actuators are wheel axles on a vehicle and hydraulic cylinders on an articulated arm. Whenever there is motion within the power line there will be a corresponding motion in the mechanical system, and vice versa. There are two basic forms of Actuators: RotationalActuator and TranslationalActuator. There are also variants of these, such as PistonActuator that is a TranslationalActuator transforming hydraulic pressure and flow into translational motion.

Actuators are always created from a regular AGX constraint and for a 1-DOF constraint the effect of the power line onto the bodies is by default the same as-if the Motor1D of the constraint had been used. Creating a RotationalActuator for a hinge will cause rotation about the hinge's axis and a TranslationalActuator associated with a prismatic will cause translational motion along the prismatic's axis. The coupling is two-way, so the power source in the power line will feel the load from the mechanical bodies, including forces from other constraints, contacts and external forces such as gravity.

Constraints with a more complicated geometry than Constraint1DOF will need to provide additional information in order to connect to an Actuator. This information is supplied in the form of a ConstraintGeometry instance, which describe the location and direction on the rigid bodies at which the Actuator should act. An example of a such a constraint is the WheelJoint, which provide the corresponding `WheelJointConstraintGeometry` which makes it possible to attach an Actuator to either the wheel, steering, or suspension axes of a WheelJoint.

18.2.5. Connecting components

Components are connected using the various connect methods. The most basic form:

```
component1->connect(component2);
```



connects the *output* side of component1 to the *input* side of component2. The general rule is that a `connect` call must contain both a Unit and a Connector, but for the simplest cases the Connector can be created implicitly. The system may also chose to reuse a Connector that is already present at one of the connection points. This make it possible to connect two Units directly.

To create more complicated power lines it becomes necessary to specify on which side of a component that a connection should be made. This is done by passing side flags to the `connect` methods. The following example connects the input side of component1 to the input side of component2.

```
component1->connect(INPUT, INPUT, component2);
```



Units contain PhysicalDimensions of various types and the Connectors declare, for each side, which type or types of PhysicalDimensions the Connector can connect to. For a connection to be successful there must be at least one match between the dimension types that the Connector expects and the types that the Unit provides. Connectors can also have a maximum number of allowed connections for each side. Attempting to connect a Unit to a Connector that has reached the maximum number of connections at the given side will fail.

Units have a notion of forward, or positive, motion. It is defined as the direction moved when the value of the Unit's PhysicalDimension is positive. The normal case has two connection possibilities. Connecting two Units at different sides, i.e., output-to-input or input-to-output, causes the direction to be maintained, and connecting two Units at the same side, i.e., output-to-output or input-to-input, causes the motion to be reversed. Some Connectors, such as a gear box with a reverse gear in, can change this behavior.

18.2.6. Direct body manipulation

Since the PhysicalDimensions are one-dimensional it is possible to pack multiple dimensions into the same `agx::RigidBody`, which is done for performance reasons. A consequence of this is that care must be taken when manipulating the body since any change may influence other Units in unexpected ways. To avoid this issue for individual PhysicalDimensions it is possible to reserve the body for that PhysicalDimension only. This is done by calling `PhysicalDimension::reserveBody`. The body is then accessed by calling `PhysicalDimension::getOrReserveBody`.

In order to meaningfully manipulate a reserved power line body it is necessary to know its *direction*. For a rotational dimension the direction is the axis around which it rotates, and for translational and flow dimension it is the direction of motion. The world direction is provided by `PhysicalDimension::getWorldDirection`. The direction should always be fetched after the body has been reserved because the reserve process

may change the world direction. The direction should be taken into account when, for example, attaching a constraint to the reserved power line body. The constraint creation helper function `agx::Constraint::createFromWorld` can help with this.

The following code snipped demonstrates how to drive a shaft using the motor on a regular hinge.

```
agxDriveTrain::ShaftRef shaft = new agxDriveTrain::Shaft();
agxPowerLine::RotationalDimension* rotDim = shaft->getRotationalDimension();

agx::RigidBody* body = rotDim->getOrReserveBody();
agx::HingeRef hinge = agx::Constraint::createFromWorld<agx::Hinge>(
    body->getCmPosition(), rotDim->getWorldDirection(), body);
simulation->add(hinge);
hinge->getMotor1D()->setSpeed(agx::Real(2.0));
hinge->getMotor1D()->setEnable(true);
hinge->getMotor1D()->setForceRange(agx::RangeReal(agx::Real(100000.0)));
```

18.3. Rotational and Linear Actuators

By using the `RotationalTranslationalHolonomicConnector` class it is possible to connect rotational constraints to linear and vice versa. Fig. 18.1 illustrates a system where a hinge is used to power a prismatic constraint. A rotation of the hinge will lead to a linear motion in the prismatic.

For more illustrative examples, see `data/python/linear_actuator.agxPy` and `data/python/torque_drive_winch.agxPy`

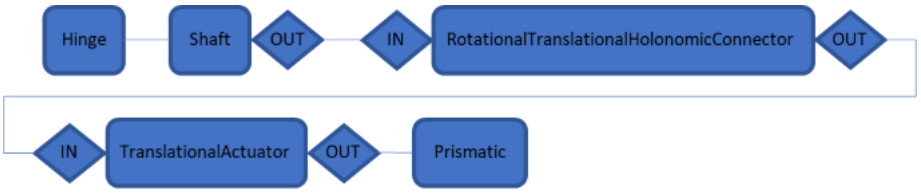


Fig. 18.1 A prismatic powered using a hinge through a drivetrain system.

```
// Build a drivetrain that connects a hinge to a prismatic constraint
agxDriveTrain::ShaftRef shaft = new agxDriveTrain::Shaft();
agxPowerLine::RotationalDimension* rotDim = shaft->getRotationalDimension();

agx::RigidBody* body = rotDim->getOrReserveBody();
agx::HingeRef hinge = agx::Constraint::createFromWorld<agx::Hinge>(
    body->getCmPosition(), rotDim->getWorldDirection(), body);
simulation->add(hinge);
hinge->getMotor1D()->setEnable(true);
hinge->getMotor1D()->setSpeed(0.5);

// Connect the prismatic constraint to the drivetrain using a TranslationalActuator
auto linearActuator = new agxPowerLine::TranslationalActuator(prismaticConstraint);
auto connector = new agxPowerLine::RotationalTranslationalHolonomicConnector();

shaft->connect(connector);
socket->connect(linearActuator);
```

19. AGX DriveTrain

AGX DriveTrain is a library within the AGX PowerLine framework designed to model the various components of a vehicle's drivetrain. It focuses exclusively on the rotational motion, meaning that all its components inherit from either `RotationalUnit` or `RotationalConnector`.

The following C++ example demonstrates how to build a simple AGX DriveTrain system using the AGX PowerLine framework.

```
// Create a ``PowerLine`` instance that will hold all drivetrain components
agxPowerLine::PowerLineRef powerLine = new agxPowerLine::PowerLine();
simulation->add(powerLine);

// Initialize an engine using predefined library parameters (e.g., Saab B234i).
agxDriveTrain::CombustionEngineParameters engineParameters;
agxDriveTrain::CombustionEngine::readLibraryProperties("Saab-B234i", engineParameters);
agxDriveTrain::CombustionEngineRef engine = new agxDriveTrain::CombustionEngine(engineParameters);
powerLine->add(engine);

// Create a simple shaft to transmit torque between engine and torque converter.
agxDriveTrain::ShaftRef engineShaft = new agxDriveTrain::Shaft();
powerLine->add(engineShaft);

// Add a torque converter to simulate automatic transmission behavior.
agxDriveTrain::TorqueConverterRef torqueConverter = new agxDriveTrain::TorqueConverter();
powerLine->add(torqueConverter);

// Create a shaft to transmit torque between torque converter and gear.
agxDriveTrain::ShaftRef torqueConverterShaft = new agxDriveTrain::Shaft();
powerLine->add(torqueConverterShaft);

// Add a gear component to model the transmission ratio.
agxDriveTrain::GearRef gear = new agxDriveTrain::Gear();
powerLine->add(gear);

// Create a shaft to transmit torque between gear and differential.
agxDriveTrain::ShaftRef gearShaft = new agxDriveTrain::Shaft();
powerLine->add(gearShaft);

// Add a differential to distribute torque between wheels.
agxDriveTrain::DifferentialRef differential = new agxDriveTrain::Differential();
powerLine->add(differential);

// Connect the differential to the wheel actuator (linked to a hinge joint in the scene).
agxPowerLine::RotationalActuatorRef wheelActuator = new agxPowerLine::RotationalActuator(hinge);
powerLine->add(wheelActuator);

// Connect all drivetrain components in sequence.
engine->connect(engineShaft);
engineShaft->connect(torqueConverter);
torqueConverter->connect(torqueConverterShaft);
torqueConverterShaft->connect(gear);
gear->connect(gearShaft);
gearShaft->connect(differential);
differential->connect(wheelActuator);
```

Note

- Ensure that each component is added to the same PowerLine instance before connecting them.
- Component order should reflect the actual power flow — from engine → wheels.
- A **detailed description** of **each drivetrain component** is provided in the subsequent sections.

19.1. Engines

An engine is a type of `RotationalUnit` that serves as the primary source of power within a drivetrain system. AGX DriveTrain provides three types of engines:

- **Combustion engine**: A conventional power source that converts fuel energy into rotational motion.
- **Torque curve based engine**: Uses a predefined torque-speed relationship.
- **Fixed velocity engine**: Maintains a constant output speed regardless of load.

Each type has distinct modeling characteristics depending on the simulation's goals and desired level of physical realism.

19.1.1. Combustion engine

The Combustion engine [13] is one of the most commonly used power source in vehicles, excavators, wheelloaders, aircraft, and marine propulsion systems. It consists of several key components (see Fig. 19.1), including

- **Cylinders and Pistons**: The main body of the engine where combustion occurs.
- **Crankshaft and Camshafts**: Convert the pistons' linear motion into rotational motion, transmitting power to the drivetrain;
- **Valves**: Control the intake and exhaust flow.
- **Fuel Injection and Ignition Systems**: Regulate and initiate the combustion process.
- **Air Intake and Exhaust Systems**: Manage airflow and remove combustion gases.

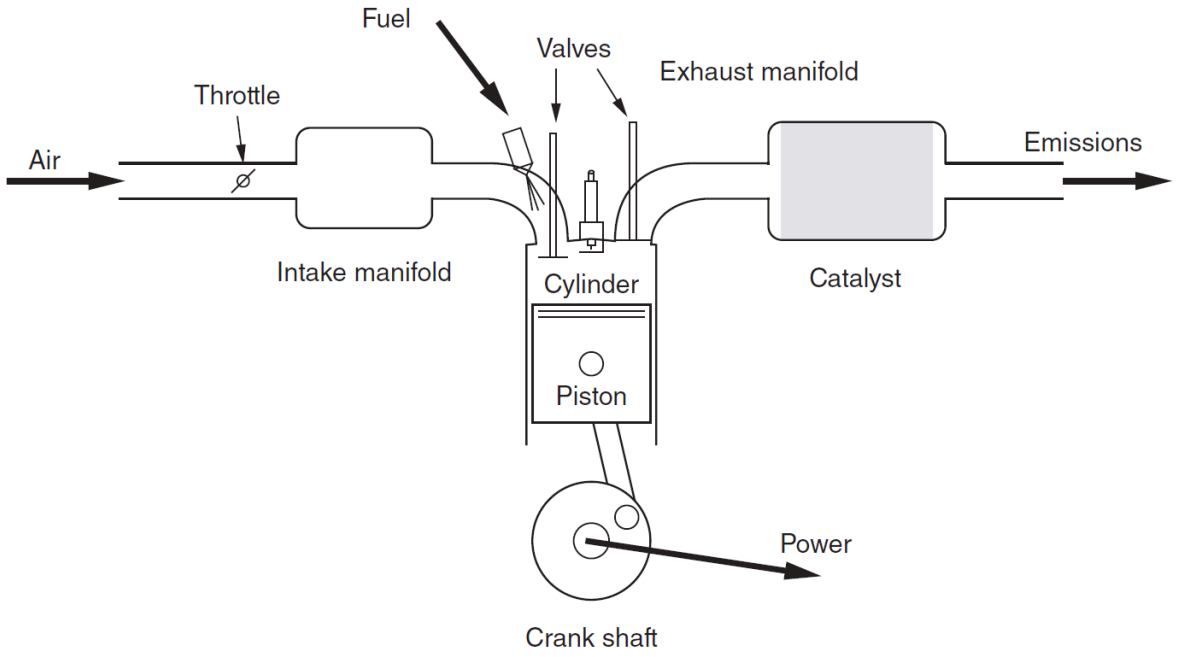


Fig. 19.1 Combustion engine schematic (source: [13]).

As a critical component of the vehicle, the combustion engine plays a decisive role in the vehicle's dynamic performance, including stability, handling, and ride comfort. A well-modeled engine provides valuable data such as:

- Engine speed
- Torque
- Power
- Fuel consumption

These outputs are essential for designing efficient powertrain systems and developing effective vehicle control strategies, meeting the increasing demands for accurate and efficient simulations.

Modelling of the combustion engine involves applying mathematical equations that describe its physical behavior, including:

- **Thermodynamics**: Determines engine efficiency and power output.
- **Fluid Mechanics**: Governs airflow and fuel delivery to the combustion chamber.
- **Rotational Dynamics**: Represents the crankshaft, connecting rods, and their conversion of linear piston motion into rotational motion.

19.1.1.1. Modelling theory

In AGX Dynamics, the mean value engine model (MVEM) [13] is implemented to represent the combustion engine. MVEM is a simplified, time-based model that predicts **averaged engine combustion variables** without simulating the detailed in-cylinder combustion process. It captures the dominant combustion characteristics averaged over one or several engine cycles.

The governing equations of the MVEM are two ordinary differential equations as shown below.

$$\begin{aligned} \frac{d\,p_{\text{im}}}{d\,t} &= \frac{RT_{\text{im}}}{V_{\text{im}}}(\dot{m}_{\text{at}} - \dot{m}_{\text{ac}}) \\ J\frac{d\,N}{d\,t} &= M_i - M_f - M_p - M_{load} \end{aligned}$$

The **first equation** describes the development of the intake manifold pressure p_{im}. Here, T_{im} is the intake temperature, V_{im} is the intake manifold volume, and R is the specific gas constant. The difference between throttle inflow $\dot{m}_{\text{at}}$ and cylinder outflow $\dot{m}_{\text{ac}}$ determines the rate of mass change in the intake system. This equation is derived from the ideal gas law $pV = mRT$.

The **second equation** governs the rotational dynamics of the engine crankshaft, with the moment of inertia J. N is the rotational speed of the crank shaft, M_i is the indicated torque, M_f is the internal friction torque, M_p is the pumping torque, M_{load} is the output(load) torque.

19.1.1.1.1. Throttle mass flow

The throttle air mass flow $\dot{m}_{at}$, depends not only on the throttle plate position but also on the intake manifold pressure. It can be expressed as [6],

$$\dot{m}_{\text{at}} = \frac{p_{\text{amb}}}{\sqrt{RT_{\text{amb}}}} A_{\text{eff}}(\alpha)\Psi_{\text{li}}(\Pi(\frac{p_{\text{im}}}{p_{\text{amb}}}))$$

where T_{amb} and p_{amb} are the ambient temperature and pressure, respectively. $A_{\text{eff}}(\alpha)$ is the effective throttle openning area, and $\Psi_{\text{li}}(\Pi(\frac{p_{\text{im}}}{p_{\text{amb}}}))$ is the flow function. Due to the large pressure difference across the throttle, the airflow velocity can reach 70–100 m/s. This flow is modeled as isentropic compressible flow.

19.1.1.1.2. Engine mass flow

The cylinder air mass flow $\dot{m}_{ac}$ depends on the engine speed N, intake pressure p_{im}, and temperature T_{im}. It is defined as,

$$\dot{m}_{\text{ac}} = \nu_{\text{vol}}(N, p_{\text{im}}, \ldots)\frac{V_D N p_{\text{im}}}{n_r RT_{\text{im}}}$$

where ν_{vol} is the volumetric efficiency, which represents the engine's ability to draw fresh air into the cylinders. It can be modeled as [14]

$$\nu_{\text{vol}} = c_0 + c_1 N + c_2 N^2 + c_3 p_{\text{im}}$$

Here, c_0, c_1, c_2, c_3 are pressure- and engine-speed-dependent coefficients.

19.1.1.1.3. Engine torques

The engine torques – indicated torque M_i, friction torque M_f, and pump torque M_p – are calculated as

$$\begin{aligned} M_i &= \frac{\dot{m}_f Q_{hv}\eta_t}{N} = \frac{\dot{m}_{ac} Q_{hv}\eta_t}{AF \cdot N} \\ M_f &= \frac{V_D}{2\pi n_r}(C_{fr0} + C_{fr1}\frac{N}{1000} + C_{fr2}(\frac{N}{1000})^2) \\ M_p &= \frac{V_D}{2\pi n_r}(p_{em} - p_{im}) \end{aligned}$$

where

- $\dot{m}_f$ – fuel flow rate
- Q_{hv} – fuel heat value
- η_t – thermal efficiency
- AF – stoichiometric air-fuel ratio
- C_{fr0}, C_{fr1}, C_{fr2} – friction torque coefficients [13]
- p_{em} – exhaust manifold pressure

Note

The default friction coefficients – $C_{fr0} = 0.97$, $C_{fr1} = 0.15$ and $C_{fr2} = 0.04$ – are suggested as representative values for engines with displacement volumes between 0.845L and 2.0L [13]. For other engines outside this range, these parameters should be calibrated against real engine test data.

19.1.1.2. Build combustion engine

The combustion engine can be parameterized, created and initialized as shown below:

```
// Initialize the engine as a Saab B234i engine.
agxDriveTrain::CombustionEngineParameters engineParameters;
agxDriveTrain::CombustionEngine::readLibraryProperties("Saab-B234i", engineParameters);
agxDriveTrain::CombustionEngineRef engine = new agxDriveTrain::CombustionEngine(engineParameters);

// Enable the engine before it can start.
engine->setEnable(true);
```

In this example, the Saab B234i engine configuration is used, which has a 2.3-liter displacement volume and delivers a maximum torque of 212 Nm at 4000 RPM.

An engine can also be reconfigured after creation:

```
// Assume that the engine has been initialized as a Saab B234i engine.
// Reconfigure it as a Volvo T5.
engine->loadLibraryProperties("Volvo-T5");
engine->setEnable(true);
```

19.1.1.2.1. Engine Parameters

The AGX Dynamics data directory includes a set of predefined engine models stored in JSON format. These files are located at: `<agx_dir>/data/MaterialLibrary/CombustionEngineParameters`.

Each file contains a parameter set (e.g., “Saab-B234i”, “Scania-K310UB4X2Euro5”) and the corresponding torque-friction coefficients. You can define a new engine model by copying an existing file and modifying the parameters as needed. Alternatively, parameters can be set explicitly in code:

```
agxDriveTrain::CombustionEngineParameters engineParameters;
engineParameters.displacementVolume = 0.003;
engineParameters.maxTorque = 300;
engineParameters.maxTorqueRPM = 3000;
engineParameters.maxPowerRPM = 4000;
engineParameters.idleRPM = 1000;
engineParameters.maxRPM = 9000;
engineParameters.crankShaftInertia = 0.5;
agxDriveTrain::CombustionEngineRef engine = new agxDriveTrain::CombustionEngine(engineParameters);
engine->setEnable(true);
```

The following table lists all available parameters for customizing a combustion engine:

Parameter	Unit	Description
displacementVolume	m^3	Total cylinder displacement volume of the engine.
maxTorque	Nm	Maximum rated torque the engine can produce over short or continuous operation.
maxTorqueRPM	RPM	Crankshaft speed at which maximum torque is achieved.
maxPowerRPM	RPM	Crankshaft speed at which maximum power is produced.
idleRPM	RPM	Crankshaft speed under idle conditions.
maxRPM	RPM	Maximum allowable crankshaft speed.
crankShaftInertia	kg m^2	Moment of inertia of the crankshaft.
cfr0		Coefficient defining the quadratic relationship between friction torque and engine speed.
cfr1		Coefficient defining the quadratic relationship between friction torque and engine speed.
cfr2		Coefficient defining the quadratic relationship between friction torque and engine speed.

throttlePinBoreRatio		Ratio between throttle pin and bore diameters (optional)
maxVolumetricEfficiency		Maximum volumetric efficiency; ratio of actual air intake volume to swept cylinder volume (optional).
airFuelRatio		Stoichiometric air–fuel ratio for complete combustion (optional).
heatValue	J/kg	Heat value of the fuel (optional).
nrRevolutionsPerCycle		Number of revolutions per cycle, e.g., `RevolutionsPerCycle::TWO_STROKE` or `RevolutionsPerCycle::FOUR_STROKE`

19.1.1.2.2. Throttle control

The engine's behaviour can be controlled via the throttle input. The throttle value ranges from 0 (idle) to 1 (fully open):

```
engine->setThrottle(throttle);
```

The available throttle-related parameters and methods are listed below:

Parameter	Method	Description
Throttle	setThrottle	Sets the throttle opening as a percentage between 0 (fully closed) and 1 (fully open).
Throttle Angle	setThrottleAngle	Defines the throttle plate position directly in radians rather than as a percentage.
Idle Throttle Angle	setIdleThrottleAngle	Specifies the minimum throttle plate angle when the engine is idling.
Max. Throttle Angle	setMaxThrottleAngle	Limits the maximum throttle plate angle during full throttle operation.

Additional engine settings, such as the number of revolutions per cycle and other configuration details, can also be adjusted to match specific engine types or performance requirements.

Detailed C++ and Python examples can be found in:

- `<agx_dir>/tutorials/tutorial_driveTrain_combustionEngine.cpp`
- `<agx_dir>/data/python/tutorials/tutorial_driveTrain_combustionEngine.agxPy`
- `<agx_dir>/data/python/tutorials/tutorial_driveTrain_torque_converter.agxPy`

19.1.1.3. Engine performance

Fig. 19.2 shows the torque and power curves for the Saab engine configuration. Both simulated and measured results are included to demonstrate and verify the correspondence between the model predictions and experimental data.

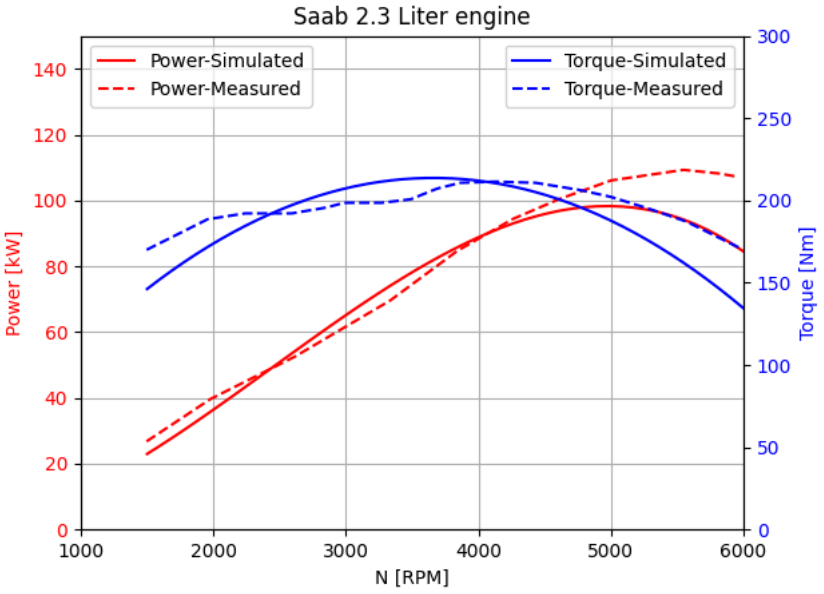


Fig. 19.2 Torque and power curves of Saab engine.

Fig. 19.3 illustrates the performance of the Volvo excavator engine. Similar to the Saab case, the figure compares simulated and measured data to validate the model's ability to reproduce the engine's real-world behavior.

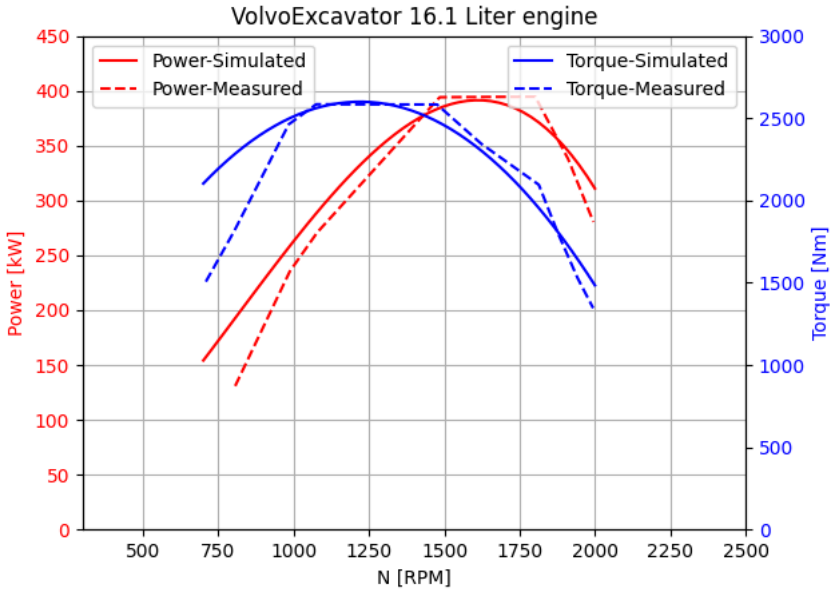


Fig. 19.3 Torque and power curves of the Volvo excavator engine.

19.1.2. Torque curve based engine

A **torque curve based engine** determines its output torque based on a predefined torque–speed relationship. It uses a torque curve to find the maximum available torque at the current engine speed (RPM), and a user-controlled throttle to scale that torque proportionally.

19.1.2.1. Example usage

A typical setup looks like this:

```
// Create an Engine instance.
agxDriveTrain::EngineRef engine = new agxDriveTrain::Engine();

// Access the engine's internal torque curve lookup table.
// This table defines how much torque the engine delivers at different speeds (RPM).
agxPowerLine::PowerGenerator* powerGenerator = engine->getPowerGenerator( );
agxPowerLine::PowerTimeIntegralLookupTable* torqueCurve = powerGenerator->getPowerTimeIntegralLookupTable();

// Insert torque-speed data points into the lookup table.
// Each pair (rpmX, torqueX) represents the engine's torque output at a given engine speed.
torqueCurve->insertValue(rpm0, torque0);
torqueCurve->insertValue(rpm1, torque1);
torqueCurve->insertValue(rpmN, torqueN);

// Set the throttle input that controls the engine's behavior (e.g., power output).
engine->setThrottle(throttle);
```

At each simulation step, the engine performs a lookup in the torque curve table to find the maximum available torque for the current RPM. It then multiplies this value by the current throttle (a scalar between 0 and 1) to determine the effective torque applied during that step.

The lookup table uses piecewise linear interpolation between inserted data points and linear extrapolation using the first or last two points when the RPM is outside the defined range.

An `Engine` also defines an idle RPM, along with a corresponding minimum torque value in its power generator. When the engine speed falls below the idle RPM, the generator applies at least this minimum torque to bring the engine back up to its idle speed.

19.1.2.2. PID-controlled engine

A variant of the `torque curve based engine` is the `PidControlledEngine`, which extends `Engine` with additional control capabilities.

It behaves similarly by default, but adds:

- Ignition control — start and stop the engine programmatically.
- Automatic throttle control — maintain a target RPM or implement custom throttle strategies.

Automatic throttle control is implemented via a `ThrottleCalculator` interface. The built-in `RpmController` is a PID-based implementation that regulates throttle input to maintain a desired RPM target.

```
engine = new agxDriveTrain::PidControlledEngine();

// Create and assign the engine's torque curve before enabling ignition.
agxPowerLine::PowerGenerator* powerGenerator = engine->getPowerGenerator( );
agxPowerLine::LookupTable* torqueCurve = powerGenerator->getPowerTimeIntegralLookupTable( );
torqueCurve->insertValue(rpm0, torque0);
torqueCurve->insertValue(rpm1, torque1);
torqueCurve->insertValue(rpmN, torqueN);

// Create an RPM controller that regulates the engine's throttle to maintain a target RPM.
// The PID gains (pGain, iGain, dGain) determine the control response, and iRange limits the integral term.
controller = new agxDriveTrain::RpmController(engine, targetRpm, pGain, dGain, iGain, iRange);

// Assign the controller to the engine's throttle calculator.
// This allows the engine's throttle to be automatically adjusted based on the PID control logic.
engine->setThrottleCalculator(controller);

// Enable the engine ignition to start the power generation process.
engine->ignition(true);
```

Here, it is worth noting that the idle control in a `PidControlledEngine` differs conceptually from that of the base `Engine` (which relies purely on a torque curve):

- When a `ThrottleCalculator` (such as `RpmController`) is assigned, it takes full control of the throttle to maintain the target RPM — including idle conditions.
- When no controller is set, the engine automatically adjusts torque using `setGradient` to reach and sustain its idle RPM.

19.1.3. Fixed velocity engine

Unlike combustion- or lookup-based engines, which calculate torque based on physical or empirical relationships, a fixed velocity engine imposes motion directly.

19.1.3.1. Principle of operation

The fixed velocity engine maintains a constant rotational speed by enforcing a velocity constraint on its output shaft.

$$0 = \omega_e - \omega_0 + \epsilon\tau$$

where ω_e refers to the angular speed of the engine output shaft, ω_0 is the target velocity, τ is the torque transfer bounded within $[-\infty, \infty]$, and ϵ is the regularization parameter of the velocity constraint.

Functionally, the fixed velocity engine behaves as an idealized actuator with effectively unlimited torque capacity. It instantly compensates for load variations or transient disturbances to maintain the prescribed speed.

Since torque is derived implicitly from the velocity constraint, the model excludes combustion, throttle control, and frictional effects. Its simplicity and determinism make it particularly useful for drivetrain validation, mechanical subsystem studies, and other cases where a constant input speed is required.

This approach guarantees a perfectly constant shaft velocity over time, making the model well-suited for testing, debugging, and validating drivetrain components under controlled input conditions.

19.1.3.2. Example usage

```
// Create a fixed velocity engine

agxDriveTrain::FixedVelocityEngineRef engine = new agxDriveTrain::FixedVelocityEngine();

// Set the target angular velocity (in radians per second)
engine->setTargetVelocity(100.0);
engine->setEnable(true);
```

This setup will maintain the output shaft velocity at 100 rad/s, independent of external load or resistance in the drivetrain.

19.2. ElectricMotor

An ElectricMotor is a type of Unit that, like an engine, acts as a source of power within a drivetrain. It models the conversion of electrical energy to mechanical energy, producing torque that drives the connected shaft.

19.2.1. Physical model

The behavior of the electric motor is defined by its resistance, inductance, torque constant, and back electromotive force (EMF) constant. These parameters describe how the motor responds to voltage and current, and how it converts electrical input into rotational motion.

The torque constant, α, defines the proportional relationship between the motor torque τ, and the current, I flowing through the windings:

$$\tau = \alpha I \, .$$

The back EMF constant, denoted β, defines the proportional relationship between the back EMF V, and the angular velocity ω of the rotor,

$$V = \beta \omega \, .$$

For an ideal motor, the constants are equal, $\alpha = \beta$. In practical systems, however, energy losses (primarily due to resistive heating in the armature) reduce efficiency, leading to $\beta \leq \alpha$. These losses become more significant at high current, where a portion of the electrical input energy is dissipated as heat rather than converted to mechanical work.

19.2.2. Example usage

When an ElectricMotor is created, the input voltage is initialized to zero. To start the motor, a nonzero voltage must be applied.

```
agxDriveTrain::ElectricMotorRef motor = new agxDriveTrain::ElectricMotor( resistance, torqueConstant,
                                                                           emfConstant, inductance );
motor->setVoltage( voltage );
```

The applied voltage can be updated at each simulation step, allowing the user to regulate the angular velocity or torque output of the motor in real time. This can be typically done by implementing a control system (for example, a PID controller) that adjusts the input voltage based on measured speed or torque targets.

Note

- Ensure the chosen constants accurately represent the physical motor being simulated. Incorrect values can cause unrealistic response or numerical instability.
- The motor can be integrated with other Unit such as gearboxes or clutches to build complete electric drive systems.

19.3. Shaft

A `Shaft` is a fundamental drivetrain Unit representing a rotating mechanical element that transmits torque and angular velocity between connected components in the power line.

As the rotational equivalent of a mechanical link, it forms the system's backbone, connecting components like engine, clutch, torque converter, and gearbox.

19.3.1. Example usage

```
agxDriveTrain::ShaftRef shaft1 = new agxDriveTrain::Shaft();
powerLine->add(shaft1);

engine->connect(shaft1);
shaft1->connect(clutch);
```

This example creates a new shaft and adds it to an existing AGX PowerLine. Once added, the shaft can be connected to other components such as engine, clutch, or gearbox through its connect methods.

19.3.2. Rotational Inertia

Each shaft possesses a physical property – rotational inertia – defines the resistance of the shaft to angular acceleration. It can rotate freely unless constrained by connected units. By default, a shaft has a rotational inertia determined by its assigned moment of inertia, which can be set explicitly to control how quickly it accelerates under torque:

```
shaft1->setInertia(0.25); // kg·m²
```

A larger inertia results in smoother, slower acceleration, while a smaller value makes the system respond more quickly to torque changes.

19.4. TorqueConverter

A torque converter is a critical component in an automatic transmission system. It consists of three main mechanical elements: the pump impeller, stator, and turbine runner, as illustrated in Fig. 19.4.

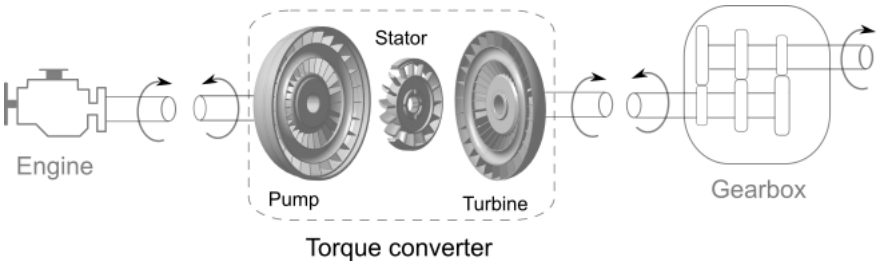


Fig. 19.4 Torque converter schematic [15].

The torque converter transfers rotational power from the engine to the transmission via fluid. The impeller, driven by the engine, accelerates the transmission fluid toward the turbine. The turbine then converts the fluid's momentum into mechanical torque, while the stator redirects the returning fluid to enhance torque multiplication and overall efficiency.

Functionally, the torque converter serves a role similar to that of a clutch in a manual transmission system. However, unlike a clutch — which disconnects the engine from the transmission when the vehicle stops — the torque converter allows the engine to continue spinning independently of the transmission. This enables the vehicle to remain stationary without stalling the engine.

19.4.1. Modelling strategy

In **AGX Dynamics**, the torque converter is modeled as such a **viscous coupling** between two rotating shafts, implemented through a **look-up table-based model**. When there is a velocity difference between the input and output shafts, the torque converter produces torque multiplication. This approach offers low computational cost and relies primarily on measured data, making it efficient and suitable for real-time simulation. It provides practical accuracy without the complexity or parameter sensitivity of dynamic or CFD-based models.

The performance of a torque converter depends on the internal design of its pump and turbine. Two key characteristic tables define its behavior:

- **Geometry Factor Table**
- **Efficiency Table**

The geometry factor, denoted as $\xi(\nu)$, is a function of the velocity ratio $\nu = \omega_t / \omega_p$ (the ratio of turbine to pump angular velocity). It is used to calculate the torque required to drive the pump:

$$M_p = \xi(\nu)\rho d_p^5 \omega_p^2,$$

where

- ρ – fluid density
- d_p – pump diameter

The efficiency, denoted as $\psi(\nu)$, defines the ratio of the turbine torque to the pump torque. The turbine torque M_t is derived from the pump torque as:

$$M_t = \psi(\nu)M_p,$$

This relationship describes how effectively the torque converter transfers energy between its components. The above-mentioned parameters can be adjusted to tune the torque converter's overall behavior.

19.4.2. Example usage

A torque converter can be created and added to a drivetrain as follows:

```
// Create a torque converter and attach it to the power line.
agxDriveTrain::TorqueConverterRef tc = new TorqueConverter();
powerLine->add(tc);

// Configure the geometry factor table, defining the relationship
// between the velocity ratio and the pump torque scaling.
TorqueConverter::setGeometryFactorTable(const agx::RealPairVector& velRatio_geometryFactor);

// Configure the efficiency table, defining how effectively the turbine
// transfers torque relative to the pump across different velocity ratios.
TorqueConverter::setEfficiencyTable(const agx::RealPairVector& velRatio_efficiency);

// The torque converter also includes a lock-up mechanism that can synchronize
```

```
// the input and output velocities for improved efficiency during steady driving conditions.
tc->enableLockUp(true);
```

By default, the torque converter is initialized with **generic parameters** suitable for typical use. It can be connected directly to the drivetrain and used without additional configuration.

Detailed C++ and Python examples with torque converter are available in

- <agx_dir>/tutorials/tutorial_driveTrain.cpp and
- <agx_dir>/data/python/tutorials/tutorial_driveTrain_torque_converter.agxPy

Note

- For more advanced applications, the **geometry factor** and **efficiency tables** can be customized to achieve specific performance characteristics or to replicate measured converter data.
- Fine-tuning of parameters ensures stable, realistic, and efficient drivetrain performance.

19.5. DryClutch

The **dry-clutch**, as shown in Fig. 19.5 a), is a mechanism that enables the engine to be connected to or disconnected from the rest of the transmission system without stalling the engine. This is achieved by pressing friction plates against each other, allowing controlled slip during the engagement phase.

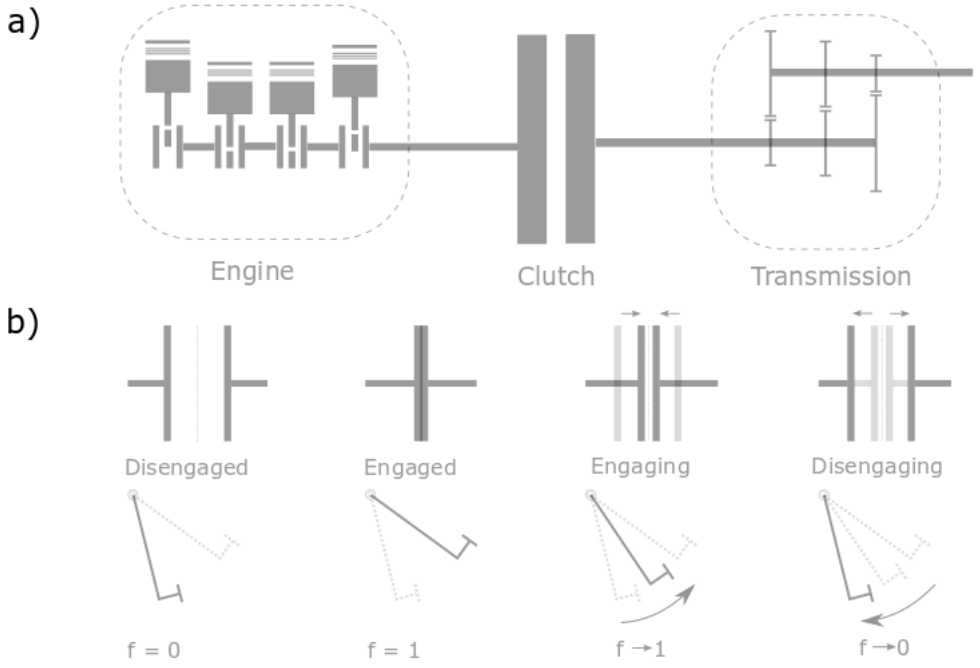


Fig. 19.5 Dry clutch schematic. a) Relative position; b) Working schemes.

Fig. 19.5 b) shows the operational principles of the clutch. A variable of fraction, denoted as f, represents the relative position of the clutch discs corresponding to the clutch pedal operation. When $f = 0$, the clutch is fully disengaged, meaning the clutch pedal is fully depressed. Conversely, $f = 1$ indicates a fully engaged clutch. During the engagement and disengagement process, f varies continuously within the range $[0, 1]$.

19.5.1. Governing equations

The dry-clutch is modelled based on dry friction between input and output shafts, combined with a locking mechanism that engages when the two shafts rotate at nearly the same angular velocity. Dry friction is represented as a **velocity (nonholonomic) constraint** which enforces that the relative velocities of the connected shafts remain equal as long as the torque required to maintain this condition stays below the slip threshold.

$$0 = \omega_e - \omega_s \perp \tau \in [-\tau_C, \tau_C]$$

where ω_e denotes the angular speed of the engine crackshaft, ω_s is the angular speed of the transmission input shaft, τ represents the torque transmitted through the clutch discs, τ_C is the maximum torque capacity of the clutch.

The perpendicularity condition $\perp$ indicates a **complementarity relationship** between the relative angular velocity and the transmitted torque. This means that:

- if $\omega_e - \omega_s = 0$ then $-\tau_C < \tau < \tau_C$ (no slip),
- if $\omega_e - \omega_s > 0$ then $\tau = -\tau_C$, and
- if $\omega_e - \omega_s < 0$ then $\tau = \tau_C$.

When the lock (a positional, or holonomic constraint) is activated, the following condition is imposed:

$$0 = \phi_e - \phi_s$$

where ϕ_e denotes the angular position of the engine crackshaft, ϕ_s is the angular position of the transmission input shaft.

This constraint models the clutch's locking mechanism, ensuring the engine crackshaft and the transmission input shaft rotate as a single rigid body. In the locked state, there are **no limits on the transmitted torque**, as the connection is fully rigid.

19.5.2. Parameters

As described above, the dry clutch model involves several physical parameters – such as the torque capacity and engagement fraction – used for both modelling and simulation. In addition, a number of simulation control parameters must be specified by the user, as listed in the table below.

Parameter	Method	Description	Default
f	setFraction	Fraction of the maximum distance between the clutch plates (or equivalently, the fraction of the maximum normal force between the clutch pads and the flywheel). The value varies between 0 and 1.	0
τ_C	setTorqueCapacity	Maximum torque transferable through the clutch plates (in N·m)	225
autoLock	setAutoLock	Enables or disables the auto locking feature	False
engageTimeConstant	setEngageTimeConstant	Time constant governing the transition from the disengaged to the engaged state. Its unit is consistent with the simulation time step. The time constant ranges within $(0, \infty]$, where ∞ indicates that the clutch pedal remains fixed in its current position.	2.5
disengageTimeConstant	setDisengageTimeConstant	Time constant governing the transition from the engaged to the disengaged state.	1.0
minRelativeSlip	setMinRelativeSlip	Threshold used to determine whether the locking criterion is satisfied. The parameter is dimensionless	1e-5
manual	setManualMode	Set the clutch control mode: manual or automatic (See Clutch manipulation)	False
engage	setEngage	Activate clutch engagement (True) or disengagement (False)	False

The default values of **torque capacity** τ_C and the **time constants** are adopted from [5], which correspond to the engine of a medium size passenger car. For heavy duty engine, the clutch torque capacity can reach up to **1200N·m** [6].

It is worth noting that all the parameters are **fully customizable**. Users can adjust them to reflect different clutch characteristics or to simulate varying driving conditions and operational scenarios, such as traffic-dependent behavior.

19.5.3. Build clutch

The clutch is constructed, configured and connected as follows:

```
// Create a power line to host drivetrain components.
agxPowerLine::PowerLineRef powerLine = new agxPowerLine::PowerLine();

// Create the dry clutch and set key physical/simulation parameters.
agxDriveTrain::DryClutchRef dryClutch = new agxDriveTrain::DryClutch();
dryClutch->setTorqueCapacity(400.0);
dryClutch->setEngageTimeConstant(2.5);
dryClutch->setDisengageTimeConstant(1.0);
dryClutch->setAutoLock(true);

// Add the clutch to the power line.
powerLine->add(dryClutch);

// Create an engine and a transmission-side shaft, then connect them via the clutch.
agxDriveTrain::FixedVelocityEngineRef engine = new agxDriveTrain::FixedVelocityEngine();
agxDriveTrain::ShaftRef transmissionShaft = new agxDriveTrain::Shaft();

// Connect engine to the input side of the clutch and the output side to the transmission.
```

```
dryClutch->connect(engine);
dryClutch->connect(transmissionShaft);
```

Once the clutch is built, it can be connected to the engine (See Engines) and transmission shafts (See Shaft).

19.5.4. Clutch manipulation

The clutch supports two control modes: **manual** and **automatic**:

- In manual mode, the user drives the engagement directly via `setFraction()` with a value in $[0, 1]$.
- In automatic mode, the user toggles engagement with `setEngage(bool)`, while the time constants govern the ramp behavior.

Manual mode example – gradually engage over 1 s, then linearly disengage over 2 s:

```
class ManualClutchListener : public agxSDK::StepEventListener
{
public:
  // Constructor: takes a pointer to the DryClutch instance
  ManualClutchListener(agxDriveTrain::DryClutch* clutch) :
  m_clutch(clutch)
  {
    // Ensure the clutch is in manual mode so setFraction() is honored.
    m_clutch->setManualMode(true);
  }
  void pre(const agx::TimeStamp& t)
  {
    // Example schedule:
    // 0–1 s   : ramp fraction from 0 -> 1 (engaging)
    // 1–3 s   : ramp fraction from 1 -> 0 (disengaging)
    if (t > 0 && t <= 1.0) {
      m_clutch->setFraction(t); // linearly 0 -> 1 over 1 s
    }
    else if (t > 1.0 && t <= 3.0) {
      m_clutch->setFraction(1 - (t-1) / 2);  // 1 -> 0 over 2 s
    }
private:
  agxDriveTrain::DryClutch* m_clutch = nullptr;
}
```

Automatic mode example — use `setEngage()` with time constants:

```
class AutoClutchListener : public agxSDK::StepEventListener
{
  public:
    AutoClutchListener(agxDriveTrain::DryClutch* clutch) :
    m_clutch(clutch)
    {
      // Ensure automatic mode so setEngage() is effective.
      m_clutch->setManualMode(false);
    }
    void pre(const agx::TimeStamp& t)
    {
      // Example schedule:
      // 3–5 s   : engage over 2 s
      // 7–9 s   : disengage over 1 s
      // else    : hold current state (infinite time constants)
      if (t > 3.0 && t <= 5.0)
      {
        m_clutch->setEngage(true);
        m_clutch->setEngageTimeConstant(2.0);     // 0 -> 1 in 2 s
      }
      else if (t> 7.0 && t <= 9.0)
      {
        m_clutch->setEngage(false);
        m_clutch->setDisengageTimeConstant(1.0); // 1 -> 0 in 1 s
      }
      else {
```

```
        // "Hold" behavior: changes are effectively frozen.
        m_clutch->setEngageTimeConstant(agx::Infinity );
        m_clutch->setDisengageTimeConstant(agx::Infinity);
      }
  private:
    agxDriveTrain::DryClutch* m_clutch = nullptr;
  }
```

Detailed C++ and Python examples with DryClutch are available in

- `<agx_dir>/tutorials/tutorial_driveTrain.cpp` and
- `<agx_dir>/data/python/tutorials/tutorial_driveTrain_dryClutch.agxPy`.

Note

- Use `setManualMode(true)` when driving `setFraction()`.
- Use `setManualMode(false)` when driving `setEngage()`.
- The “hold” behavior is achieved by setting time constants to `infinity`, which maintains the current state.

19.6. Gear

The `Gear` is the most fundamental drive train Connector. It transfers rotational motion between the `Unit`-based drive train components. A `Gear` represents the interaction between two geared wheels – one on each connected `Unit` – and is characterized by a *gear ratio*.

The gear ratio defines the number of gear teeth on the output shaft relative to the input shaft.

- A gear ratio **greater than 1** causes the output gear to rotate slower than the input,
- A gear ratio **less than 1** makes the output gear rotate faster than the input shaft,
- A **negative** ratio reverses the rotation direction of the output shaft.

Mathematically:

$$\omega_{out} = \frac{\omega_{in}}{g}$$

where g is the gear ratio.

A pair of shafts can be connected with the gear using the following example code:

```
agxDriveTrain::ShaftRef inputShaft = new agxDriveTrain::Shaft();
agxDriveTrain::ShaftRef outputShaft = new agxDriveTrain::Shaft();

// Create a reference to a new gear that will connect the two shafts
agxDriveTrain::GearRef gear = new agxDriveTrain::Gear();

// Add the created components to the power line system
powerLine->add(inputShaft);
powerLine->add(outputShaft);
powerLine->add(gear);

// Set the gear ratio to 2.0
// This means the output shaft rotates at half the speed of the input shaft,
// but with double the torque (typical gear ratio effect)
gear->setGearRatio(agx::Real(2.0));

// Connect the input shaft to the gear, and the gear to the output shaft
// This completes the mechanical power transmission path: input → gear → output
inputShaft->connect(gear);
gear->connect(outputShaft);
```

19.6.1. GearBox

A `GearBox` is a gear-like connector containing a collection of selectable gear ratios. It is initialized with a vector of ratios and provides methods for switching between them.

```
// Create a vector to store the gear ratios for the gearbox
agx::RealVector gearRatios;
```

```
// Add different gear ratios to the vector
// Higher ratios provide more torque but less speed (lower gears)
// Lower ratios provide less torque but more speed (higher gears)
gearRatios.push_back(agx::Real(10));
gearRatios.push_back(agx::Real(7));
gearRatios.push_back(agx::Real(5));
gearRatios.push_back(agx::Real(3.5));
gearRatios.push_back(agx::Real(2));
gearRatios.push_back(agx::Real(1));

// Create a gearbox using the specified gear ratios, and add it to the power line
agxDriveTrain::GearBoxRef gearBox = new agxDriveTrain::GearBox(gearRatios);
powerLine->add(gearBox)

// Set the current gear of the gearbox to the first gear (index 0)
// This means the gearbox will initially use the 10:1 ratio
gearBox->setGear(0);
```

The above example creates a six-speed gear box with decreasing gear ratios – meaning increasing output velocity – and set the current gear to the first one.

19.6.2. SlipGear

A `SlipGear` behaves like a regular gear but includes configurable compliance. By adjusting this compliance, you can allow controlled slip under load — useful for simulating flexible couplings, belt drives, or non-rigid gear connections.

```
// Create a slipGear, and add it to the power line
agxDriveTrain::SlipGearRef slipGear = new agxDriveTrain::SlipGear();
powerLine->add(slipGear)

// Set the gear ratio to 5.0
slipGear->setGearRatio(5.0);

// Adjusting the compliance enables the gear to slip under load, from slight to significant.
slipGear->setViscousCompliance(1e-3);
```

19.6.3. HolonomicGear

While a `SlipGear` introduces flexibility, a `HolonomicGear` does the opposite — it enforces perfect synchronization between connected shafts.

The `HolonomicGear` employs a **holonomic constraint**, ensuring that the two connected shafts remain perfectly synchronized with no slippage.

```
// Create a holonomicGear, and add it to the powerline
agxDriveTrain::HolonomicGearRef holonomicGear = new agxDriveTrain::HolonomicGear();
powerLine->add(holonomicGear);

// Set the gear ratio to 5.0
// The holonomic constraint guarantees perfect angular synchronization
holonomicGear->setGearRatio(5.0);
```

19.6.4. MultiGear

A `MultiGear`, as illustrated in Fig. 19.6, allows multiple connected shafts, each with its own gear ratio. When one connected shaft rotates, an internal hidden shaft will rotate at a velocity determined by that shaft's gear ratio. All other connected shafts then rotate according to their respective gear ratios.

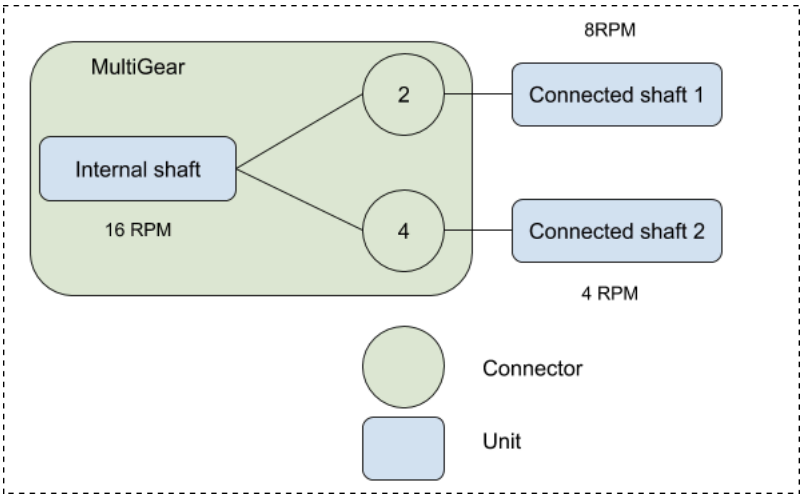


Fig. 19.6 MultiGear schematic.

```
// Create an internal shaft that acts as the input to the multi gear
agxDriveTrain::ShaftRef internalShaft = new agxDriveTrain::Shaft();

// Create two output shafts that will receive power from the multi gear
agxDriveTrain::ShaftRef outputShaft1 = new agxDriveTrain::Shaft();
agxDriveTrain::ShaftRef outputShaft2 = new agxDriveTrain::Shaft();

// Create a MultiGear, and add it to the power line
agxDriveTrain::MultiGearRef multiGear = new agxDriveTrain::MultiGear();

powerLine->add(internalShaft);
powerLine->add(outputShaft1);
powerLine->add(outputShaft2);
powerLine->add(multiGear)

// Set the gear ratio for each output shaft
multiGear->setGearRatio(outputShaft1, 2.0);
multiGear->setGearRatio(outputShaft2, 4.0);

// Connect the shafts to the multi gear
// This completes the power transmission paths:
// internalShaft → multiGear → outputShaft1 / outputShaft2
internalShaft->connect(multiGear);
multiGear->connect(outputShaft1);
multiGear->connect(outputShaft2);
```

Note

A `MultiGear` only supports connections on the **output side**.

19.7. Differential

A `Differential` is a Connector that distributes torque evenly over multiple output shafts. It ensures that the angular velocity of the input shaft equals the average of the velocities of the output shafts.

The differential can be locked, forcing all outputs to rotate at the same speed. Additionally, it supports **a limited slip torque mode**, which allows only a limited amount of torque to be transferred between outputs. This helps maintain equal output shaft velocities while still permitting slight differences when necessary.

19.7.1. Example Usage

The example below shows how to create a differential and connect it to two wheel actuators, enabling torque distribution between the left and right wheels so they can rotate at different speeds for smooth and realistic cornering behavior.

```
// Create an input shaft that delivers power into the differential
agxDriveTrain::ShaftRef inputShaft = new agxDriveTrain::Shaft();

// Create a differential, which splits torque between two output shafts (wheels)
```

```
// allowing them to rotate at different speeds – essential for smooth cornering
agxDriveTrain::DifferentialRef differential = new agxDriveTrain::Differential();

// Create two rotational actuators that will drive the left and right wheel hinges
agxPowerLine::RotationalActuatorRef leftWheelActuator = new agxPowerLine::RotationalActuator(left_wheel_hinge);
agxPowerLine::RotationalActuatorRef rightWheelActuator = new agxPowerLine::RotationalActuator(right_wheel_hinge);

// Add all drivetrain components to the power line
// The power line manages the dynamic interaction between these components
powerLine->add(inputShaft);
powerLine->add(differential);
powerLine->add(leftWheelActuator);
powerLine->add(rightWheelActuator);

// Connect the drivetrain components to form the complete power flow:
// inputShaft → differential → leftWheelActuator / rightWheelActuator
inputShaft->connect(differential);
differential->connect(leftWheelActuator);
differential->connect(rightWheelActuator);
```

19.8. Brake

The `Brake`, as shown in Fig. 19.7, is a mechanism that applies a counteracting torque to a rotor, gradually reducing its rotational motion until it comes to a complete stop.

A typical brake assembly consists of the caliper, brake pads, and hydraulically actuated pistons. The pistons press the brake pads firmly against the rotating disc, generating friction and slowing down the rotor. Fig. 19.7 (b) shows the operational sequence of the brake, detailing both the engagement and disengagement processes. It illustrates how the components interact as the brake is applied and released.

The fraction, denoted as f, represents the position of the brake pedal relative to the motion of the brake pads during braking operations.

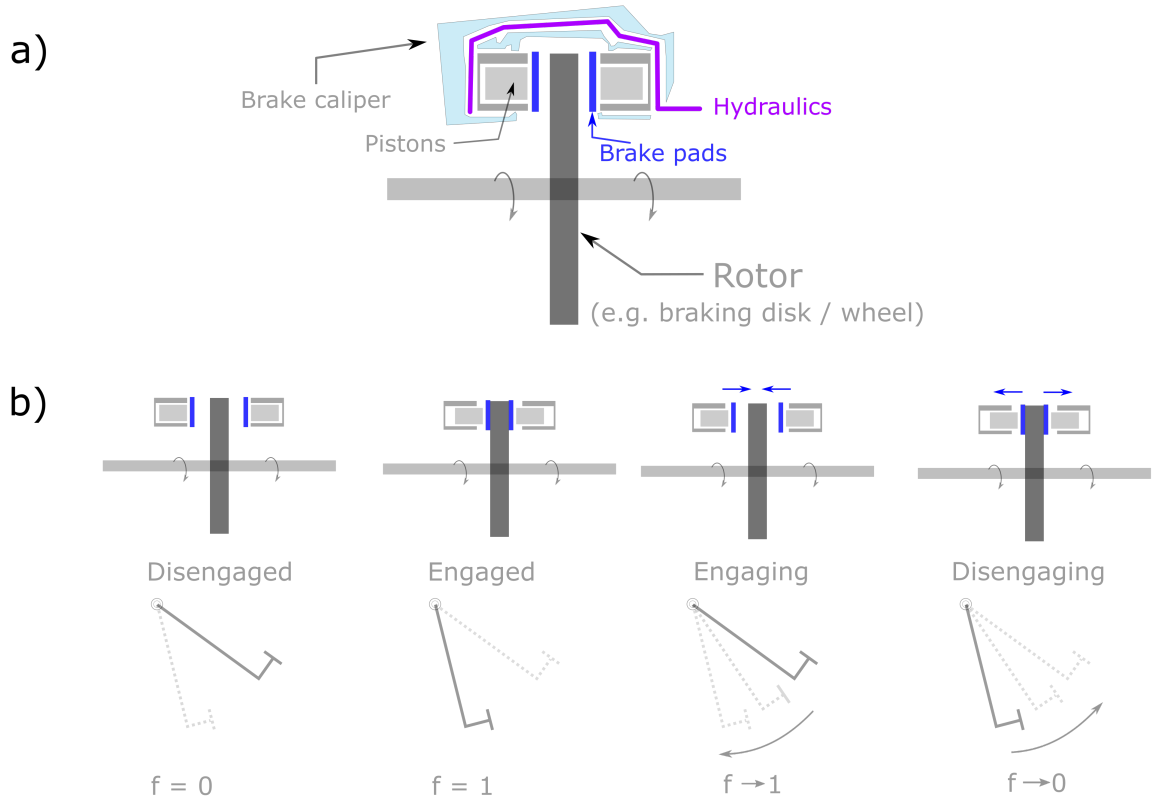


Fig. 19.7 Brake schematic. (a) Typical assembly; (b) Working mechanism.

A value of 0 indicates that the brake is fully disengaged – the pedal is fully released, and no braking torque is applied (free rotation). A value of 1 means the brake is a fully engaged – the pedal is fully pressed, and maximum braking torque transfer is applied. During the engaging and disengaging process, f varies within $[0, 1]$, simulating the varying braking force based on the pedal position.

The brake operates similarly to a DryClutch, but in reverse. While a clutch engages when the pedal is released, a brake engages when the pedal is pressed. Because of their mechanical similarity, both the Brake and DryClutch share a unified API interface, reflecting their closely related operational principles.

19.8.1. Governing equations

In AGX, the brake is modelled as a **velocity (nonholonomic) constraint** attached to a rotational Shaft. The constraint is expressed as

$$0 = \omega \perp \tau \in [-\tau_C, \tau_C],$$

where - ω refers to the angular speed of the shaft, - τ is the torque transmitted through the brake pad, - τ_C is the maximum torque capacity.

The perpendicularity condition ($\perp$) implies that either:

- $\omega = 0$ and $-\tau_C < \tau < \tau_C$, or
- $\omega > 0$ and $\tau = -\tau_C$, or
- $\omega < 0$ and $\tau = \tau_C$.

This constraint regulates the shaft's rotation by controlling the magnitude of the torque applied, with two key configurable parameters: the fraction f and the torque capacity τ_C.

The torque capacity τ_C depends on the properties of the brake pads and defines the maximum amount of transferable torque. A higher torque capacity leads to a stronger braking effect, enabling quicker deceleration as the brake pedal position increases. By adjusting the brake pedal position and torque capacity, the brake model can simulate different braking behaviors – from gradual deceleration to a more immediate stop.

When the lock (the positional, or holonomic constraint) is activated, the following constraint is enforced:

$$0 = \phi,$$

where ϕ refers to the angular position of the shaft. This models a parking brake mechanism, keeping the shaft stationary with no limits on the applied torque.

19.8.2. Parameters

The physical parameters of the Brake are listed below.

Parameter	Method	Description	Default
f	setFraction	Fraction of the brake pedal position (or equivalently, the fraction of the maximum counter torque applied by the brake pads). Ranges from 0 to 1.	0
τ_C	setTorqueCapacity	Maximum torque that the brake pads can transfer (N·m)	2000
autoLock	setAutoLock	Enable or disable automatic locking	True
engageTimeConstant	setEngageTimeConstant	Time required to transition the brake from disengaged to engaged. The unit matches the user-defined time step. A value of $(0, \infty]$ indicates that the brake pedal remains in its current position.	2.5
disengageTimeConstant	setDisengageTimeConstant	Time required to transition the brake from engaged to disengaged.	1.0
minRelativeSlip	setMinRelativeSlip	Threshold for checking whether the lock criterion is fulfilled (dimensionless).	1e-5
manual	setManualMode	Set brake to manual or automatic control mode (See Brake manipulation)	False
engage	setEngage	Activate (True) or deactivate (False) the braking engagement operation	False

Users can adjust these parameters to achieve desired braking performance, responsiveness, and driving characteristics.

19.8.3. Build brake

A brake can be created and configured as follows:

```
agxPowerLine::PowerLineRef powerLine = new agxPowerLine::PowerLine();
agxDriveTrain::BrakeRef brake = new agxDriveTrain::Brake();
brake->setTorqueCapacity(1000.0);
brake->setEngageTimeConstant(0.2);
brake->setDisengageTimeConstant(1.0);
brake->setAutoLock(true);
powerLine->add(brake);
```

Once built, the brake can be connected to other DriveTrain components.

19.8.4. Brake manipulation

The brake can operate in two modes: **manual** and **automatic**.

In the manual mode, the user can in great detail control the operation of the brake using the `setFraction()` method, see the code snippet below:

```
class ManualBrakeListener : public agxSDK::StepEventListener
{
  public:
    ManualBrakeListener(agxDriveTrain::Brake* brake) : m_brake(brake)
    {
    }
    void pre(const agx::TimeStamp& t)
    {
      // Disengaged
      if (t < 1.0) {
        m_brake->setFraction(0.0);
      }
      // Engaging
      else if (t <= 1.1) {
        m_brake->setFraction(10 * (t-1));
      }
      // Disengaging
      else if (t <= 3.0) {
        m_brake->setFraction((3.0-t)/(3-1.1));
      }
  private:
    agxDriveTrain::Brake* m_brake;
  }
```

In this example, the brake begins engaging rapidly at t=1 s and is fully engaged at t=1.1 s. It then disengages gradually, becoming fully disengaged at t=3.0 s.

In the **automatic** mode, the brake can be controlled using `setEngage()` method:

```
class AutoBrakeListener : public agxSDK::StepEventListener
{
  public:
    AutobrakeListener(agxDriveTrain::Brake* brake) : m_brake(brake)
    {
    }
    void pre(const agx::TimeStamp& t)
    {
      // Engaging
      if (t > 3.0 && t <= 5.0)
      {
        m_brake->setEngage(true);
        // Engage goes from 0 to 1 in 2 seconds.
        m_brake->setEngageTimeConstant(2.0);
      }
      // Disengaging
      else if (t> 7.0 && t <= 9.0)
      {
        m_brake->setEngage(false);
        // Disengage goes from 1 to 0 in 1 seconds.
        m_brake->setDisengageTimeConstant(1.0);
      }
      // Brake on-hold
      else {
        m_brake->setEngageTimeConstant(agx::Infinity );
        m_brake->setDisengageTimeConstant(agx::Infinity );
      }
  private:
    agxDriveTrain::Brake* m_brake;
  }
```

Comprehensive C++ and Python examples with `Brake()` can be found in:

- `<agx_dir>/tutorials/tutorial_driveTrain_combustionEngine.cpp`,
- `<agx_dir>/data/python/tutorials/tutorial_driveTrain_brake.agxPy`, and
- `<agx_dir>/data/python/tutorials/tutorial_driveTrain_combustionEngine.agxPy`.

19.9. Actuator

An Actuator is a `PowerLine` component that transfers motion between a powerline `Unit` and a standard AGX constraint such as a `Hinge` or `Prismatic`.

Since a `Hinge` represents rotational constraint and a `Prismatic` represents translational constraint, there are two specialized subclasses of the `Actuator` class:

- `agxPowerLine::RotationalActuator` – connects to rotational constraints (e.g., hinges).
- `agxPowerLine::TranslationalActuator` – connects to translational constraints (e.g., prismatic joints).

19.9.1. Example Usage

If a wheel body is attached to a chassis via a hinge named hinge, the wheel can be powered through the drivetrain as follows:

```
// Create a rotational actuator linked to the hinge constraint
agxPowerLine::RotationalActuatorRef actuator = new agxPowerLine::RotationalActuator(hinge);

// Connect an existing drivetrain (shaft) to the hinge via the RotationalActuator.
shaft->connect(actuator);
```

This establishes a link between the drivetrain shaft and the hinge, allowing torque and rotational motion to be transmitted through the PowerLine system.

19.10. WireWinchActuator

A winch is a rotating drum used to spool wire in or out. Its effective radius changes depending on how much wire is currently wound on the drum. To simulate a torque-driven winch, it must be connected to a rotational power source, such as a motorized Hinge or an Engines.

The agxWire::WireWinchController manages winch operation by controlling the motor and brake. It acts as a linear constraint, applying a linear force (in Newtons) along the wire to simulate real winching behavior.

Since the winch mechanism operates linearly while the power source operates rotationally, AGX uses `agxPowerLine::RotationalTranslationalHolonomicConnector` to convert rotational motion to linear motion.

To connect this converter to the winch, use `agxPowerLine::WireWinchActuator`, which bridges rotational and linear power transmission, ensuring physically consistent torque-to-tension conversion.

By combining:

- a **motorized hinge** (rotational input),
- a **RotationalTranslationalHolonomicConnector** (conversion element), and
- a **WireWinchActuator** (link to the winch),

AGX provides a robust framework for simulating realistic wire and rope systems that accurately transform motor torque into wire tension.

19.10.1. Example Usage

The following example demonstrates how to integrate a `WireWinchActuator` into a AGX PowerLine system. It shows how rotational motion from a motorized hinge is converted into linear motion to drive a winch through a `RotationalTranslationalHolonomicConnector`.

```
// PowerLine chain:
//   [Hinge]
//      ↓
//   [Shaft]
//      ↓
//   [RotationalTranslationalHolonomicConnector]
//      ↓
//   [WireWinchActuator]
//      ↓
//   [Winch]

// Assume we have a pointer to a winch object with a routed wire.
```

```
// Create a shaft that transfers rotational motion from the motorized hinge to the connector.
auto shaft = new agxDriveTrain::Shaft();
auto winch_socket = new agxPowerLine::RotationalTranslationalHolonomicConnector();

// Define the effective radius of the winch drum.
// This determines the relationship between the hinge's rotational motion and the winch's linear motion.
// A larger radius produces higher linear speed but lower pulling force.
winch_socket->setShaftRadius( radius );

// Connect the shaft to the winch_socket
shaft->connect( winch_socket );
shaft->setInertia( 1e-4 ); // Approximate inertia of the rotating drum

// Add the shaft to the PowerLine system.
power_line->add( shaft );

// Create a hinge aligned with the shaft's rotational axis.
auto frame = new agx::Frame();
frame->setLocalRotate( agx::Quat( agx::Vec3::Z_AXIS(), shaft->getRotationalDimension()->getWorldDirection())  );

// Initialize the hinge constraint.
auto winch_hinge = new agx::Hinge( shaft->getRotationalDimension()->getOrReserveBody(), frame );

// Initially, keep the hinge locked and disable the motor.
// To start winching, unlock the hinge and enable the motor with a target speed.
winch_hinge->getMotor1D()->setEnable( false );
winch_hinge->getMotor1D()->setSpeed( 0 );

// Note: Lock1D is not a brake – it behaves like a spring when its target position is exceeded.
// To prevent "springback", either update the lock target position dynamically
// or replace it with a friction-based controller.
winch_hinge->getLock1D()->setEnable( true );
simulation->add( winch_hinge );

// Create an actuator that transfers linear motion from the connector to the winch.
auto winch_actuator = new agxPowerLine::WireWinchActuator( winch );

// Connect the actuator to the connector and register it in the PowerLine system.
winch_socket->connect( winch_actuator );
power_line->add( winch_actuator );
```

For a more complete example, see `<agx_dir>/data/python/torque_driven_winch.agxPy`.

19.11. Clutch (deprecated)

Note

The class Clutch has been deprecated. It is recommended to use DryClutch instead.

A clutch is a Connector with configurable efficiency and torque transfer characteristics.

The efficiency parameter ranges from 0 to 1, where:

- 0 corresponds to no torque transfer (fully disengaged), and
- 1 represents a rigid coupling (fully engaged).

The torque transfer constant define the shape of the torque transfer curve as the efficiency varies between 0 and 1. A larger constant results in a steeper torque response, meaning torque increases more rapidly as efficiency rises.

20. AGX Hydraulics

agxHydraulics is a library for modeling a hydraulic system. For code examples, look into the C++ (tutorials/agxOSG) and Python (data/python/tutorials) tutorials.

Hydraulics is the technology that deals with the generation, control and transmission of power using pressurized liquids. The AGX Hydraulics library allows for fast, accurate and stable simulation of hydraulics components in real time. It provides hydraulic components such as pumps, pipes, flow control valves, restrictions and actuators.

The hydraulics library is built upon the AGX PowerLine framework and designed to integrate with both the drive train and the mechanical system. It provides new types of Units and Connectors that can be inserted into the same power line graph as the Units and Connectors already available.

Power can be generated using an electric motor or combustion engine and transmitted through a drive train to a hydraulic pump which converts torque in the drive train to pressure in the hydraulics circuit. The pressure can then be transmitted through the hydraulics to actuators that interact with the mechanical system, or to hydraulic motors in order to transition back to a drive train.

Using the combined functionality of the power line, hydraulics and mechanics of AGX Dynamics it is possible to model and simulate complex systems such as an excavator, a crane, an offshore winch or similar.

A number of quantities can be read from the simulation, including pressure at junctions and couplings, flow rates through pipes and other components, fluid compression in pipes and junctions, and torques at pumps and motors.

20.1. Components

Table 20.1 Hydraulic components

Name	Type	Description
Accumulator	FlowUnit	Spring loaded fluid container that only accepts connections on the input side. Is filled with fluid when the input pressure is higher than the internal pressure and emptied when the input pressure is lower. Often used in conjunction with a stop valve to control filling/emptying.
CheckValve	FlowUnit	Allows flow in one direction only. Flow in the other direction is blocked. Acts like a regular pipe when the flow is in the allowed direction. Can be used to prevent actuator collapse when the working pressure is not enough to lift or hold a load.
ConstantFlowValve	FlowUnit	Has two modes. Limits the flow rate, in absolute value, to be less than some given maximum, or enforces a given flow rate. Acts like a regular pipe when in the limiting mode and the flow rate is within bounds. Acts like a pressure source when in the pumping mode.
FlowConnector	Pressure Connector	The basic hydraulic junction. Connect any type of FlowUnit and it will distribute the incoming flow among all connected FlowUnits.
Motor	Rotational Flow Connector	Connector that converts hydraulic flow into shaft rotation. The FlowUnit connected to the input becomes the fluid chamber and the RotationalUnit connected to the output becomes the motor shaft.
NeedleValve	FlowUnit	A controllable orifice opening. Used much like a pipe, but can be used to dynamically and continuously open and close a restriction. The continuous version of the stop valve.
Pipe	FlowUnit	The most basic FlowUnit. Moves fluid between other components.
PistonActuator	Translational Actuator	Models a hydraulic piston. Forms the interface between the 1-DoF hydraulics and the 6-DoF mechanical world.
Pump	Connector	Connector that converts shaft rotation into hydraulic flow. The RotationalUnit connected to the input becomes the pump shaft and the FlowUnit connected to the output becomes the pump chamber.
ReliefValve	Unit	Multiple orificed component with one input, one output and a number of drains. The drains remain closed until the pressure in the relief valve exceeds the configured cracking pressure, at which point a number of drains are opened depending on the actual pressure in the relief valve. If the pressure is at or above the configured fully open pressure then all drains open.
SpoolValve	Connector	Discrete directional control valve. Accepts any number of connected FlowUnits and can create links between them in arbitrary ways. Used to direct flow from (multiple) sources to (multiple) users.
StopValve	FlowUnit	A valve that is either open or closed. The binary version of the needle valve.
Variable Displacement Pump	Pressure Connector	A pressure regulated pump. Has a default displacement that is in effect as long as the pressure at the pump is below the configured deadhead pressure. The displacement decreases as the pressure increases from the deadhead pressure to the cutoff pressure. At and above the deadhead pressure the displacement becomes zero and the pump passive.

20.2. Class structure

The hydraulics library is built upon the power line framework and the abstractions that it provides. The hydraulics components are implemented as subclasses of `agxPowerLine::Unit`, `agxPowerLine::Connector` or `agxPowerLine::Actuator`. Many hydraulic units inherit from `agxPowerLine::Unit` via the `agxHydraulics::FlowUnit` class. Similarly, many connectors in the AGX Hydraulics library inherit from `agxPowerLine::Connector` via `agxHydraulics::PressureConnector`.

20.2.1. FlowUnit

A power line is a graph of Units connected by Connectors. Units carry motion, which is managed by PhysicalDimensions. The motion that the hydraulic units carry is the motion of the fluid itself. We call the hydraulic units FlowUnits, and the physical dimension of flow is defined by the FlowDimension class. It represents the actual flow in and out of a FlowUnit.

As described in the DriveTrain section, all PhysicalDimensions define a value, a gradient, and a mass property. The fluid flow rate, measured in units of volume per time, is the gradient of the flow dimension. The flow dimension's value is the total volume that has passed through the unit. This value may be inaccurate and should not be used. The mass property of a FlowDimension is not the mass of the fluid itself, but rather the resistance to change in flow rate given a pressure differential over the Unit. It increases with the length of the component and decreases with the area.

FlowUnits have a direction and an input and an output side. Forward, or positive, flow is defined as fluid entering at the input side and leaving at the output side. It is not required that FlowUnits are connected output-to-input. The effect of connecting input-to-input or output-to-output is that the two FlowUnits will have flow rates with opposite signs. Some hydraulic components behave differently depending on if it receives flow at the input or the output side, so it is important to keep track of the flow direction through each component.

The most basic FlowUnit is the Pipe, but several other hydraulic components are implemented as subclasses of FlowUnit as well. Hydraulic components that are not FlowUnits tend to inherit from PressureConnector instead.

20.2.2. PressureConnector

Connectors that propagate pressure through the FlowUnit graph are called pressure connectors. They all have a getPressure method, which returns the last pressure that the connector exerted on the connected FlowUnits.

An important subclass of PressureConnector is the FlowConnector. A FlowUnit can connect to many PressureConnectors, but no more than one FlowConnector per side. FlowConnectors are used to move fluid between FlowUnits. It ensures that all fluid that enters the FlowConnector from some connected FlowUnit is accounted for as flow through the other FlowUnits connected to the same FlowConnector.

20.2.3. Actuator

Actuators form the bridge between the one-dimensional world of the power line and the six-dimensional mechanical world of regular AGX simulations. An Actuator is always associated with a regular AGX Dynamics constraint which defines the operation of the Actuator. An Actuator associated with a Hinge will cause rotation around the hinge axis and an Actuator associated with a Prismatic will cause translational motion along the prismatic axis.

20.3. Building circuits

In this section a simple example circuit is built and the required steps are demonstrated. The construction is split into three parts: flow control, translational actuation and pressure generation.

20.3.1. Foundation Setup

The components of a hydraulic circuit must be part of a PowerLine object, which must be added to a simulation.

```
agxSDK::SimulationRef simulation = new agxSDK::Simulation();
agxPowerLine::PowerLineRef powerLine = new agxPowerLine::PowerLine();
simulation->add(powerLine);
```

20.3.2. Connecting FlowUnits

The agxHydraulics library provides a number of hydraulic components that can be connected together to create the hydraulic circuit. The simplest one is the pipe. The agxHydraulics::Pipe constructor takes the pipe's length and area as arguments, along with the fluid density.

This example uses the SI unit system, but that is not a general requirement. See separate section below.

```
agx::Real length(1);
agx::Real area(0.005);
agx::Real density(800);
agxHydraulics::PipeRef pipe = new agxHydraulics::Pipe(length, area, density);
```

Hydraulic components are joined together using FlowConnectors. A FlowConnector represents a junction where multiple pipes and other FlowUnits join. It cannot be connected to any other type of unit.

```
agxHydraulics::FlowConnectorRef junction = new agxHydraulics::FlowConnector();
```

We attach the pipe to the junction using the FlowConnector::connect method. The connector side argument is irrelevant for this particular type of connector, but the unit side argument is important. It specifies the side of the FlowUnit that is to be connected and thus also the direction of flow through the unit. In the following example we connect the junction's input side to the output side of the pipe.

```
junction->connect(agxPowerLine::INPUT, agxPowerLine::OUTPUT, pipe);
```

Another type of FlowUnit is the stop valve. It is a manually operated valve that is either fully open or fully closed.

```
agxHydraulics::StopValveUnitRef stop =
    new agxHydraulics::StopValveUnit(length, area, density);
```

By connecting the input side of the stop valve to the same junction that the output side of the pipe is connected to we have told the system that any flow that leaves the output side of the pipe should enter the input side of the stop valve.

```
junction->connect(agxPowerLine::OUTPUT, agxPowerLine::INPUT, stop);
```

By creating additional units and connectors it is possible to create arbitrary graphs; including forks, loops, joins and connections to tank. A requirement is that a FlowUnit may only have at most a single FlowConnector at each side. Any FlowUnit side that is not connected to anything, such as the pipe input in the example, is implicitly connected to the tank and will be subjected to tank pressure. The tank pressure is set globally.

```
agxHydraulics::utils::setTankPressure(agx::Real(1e5));
```

The components must be added to the power line for the circuit to be included in the simulation. Adding one unit will bring with it all connected units into the PowerLine, and additional Units connected later will be added as well.

```
powerLine->add(pipe);
```

20.3.3. Coupling to mechanical constraints

The flow control part of the circuit is described above. The next part is actuation, which in the example is a hydraulic cylinder. Actuation, in this context, refers to the transition from the one dimensional hydraulics domain to the six dimensional mechanics domain. The interface for this transition is always a regular AGX dynamics constraint. The presence of pressure in the hydraulics at the actuator is translated into a force or torque on the mechanical bodies according to the configuration of the constraint, and vice versa. The force or torque acts in the non-constrained degree of freedom of the AGX Dynamics constraint.

The example at hand contains a hydraulic cylinder. The class in the AGX Hydraulics library that provides that functionality is agxHydraulics::PistonActuator. It is a type of translational actuator that is created from a prismatic or a distance joint. Here it is assumed that a prismatic has been created previously.

```
agx::PrismaticRef prismatic = ...;
```

The details of the PistonActuator creation are not required for this discussion.

```
agxHydraulics::PistonActuatorRef cylinder =
                new agxHydraulics::PistonActuator(prismatic, ...);
```

Actuators are connected to the hydraulics circuit in the same way as Units. Instead of explicitly creating and connecting a FlowConnector we make use of implicit connector creation. When two FlowUnits are connected in the following manner the system either creates a new FlowConnector or uses one that has already been created for the FlowUnit sides being connected. If both sides already have a FlowConnector then they are merged into one, moving all connections from one of them to the other.

```
stop->connect(agxPowerLine::OUTPUT, agxPowerLine::INPUT, cylinder);
```

The second argument controls which side of the actuator the unit on which the call is made should be connected to. For a PistonActuator this selects one of the two chambers. Flow into the input side extends the piston, while flow into the output side compresses it. The coupling is two-way, so any force acting along the cylinder direction on the two bodies that the constraint is attached to will cause a pressure increase in one of the cylinder chambers, and any relative movement of the two bodies in the same direction will cause a flow of fluid through the cylinder.

20.3.4. Coupling to the power line

The hydraulics library provides a number of Connectors that bridge between a drive train and the hydraulics. The Pump is one such component. It accepts a RotationalUnit at its input and a FlowUnit at its output and converts torques on the input into pressures at the output. The hydraulic motor performs the opposite operation. It accepts a single FlowUnit input and a single RotationalUnit output.

Both pumps and motors have a property that defines the conversion ratio when translating between angular velocity and flow rate, called the displacement. The displacement describe the volume of fluid moved per radian of rotation of the RotationalDimension. A pump with a displacement of one will produce one unit of pressure for every unit of torque supplied at the input, and it will move, on average, one unit of volume for every radian that the input unit is rotated. With a displacement $\propto$ the pump will produce $\propto$ units of flow for every radian of rotation at the input, and generate $\frac{1}{\propto}$ units of pressure for every unit of torque that is applied at the input. When using the SI unit system, it is common for pumps to have a displacement less than one.

The example circuit contains an engine and a pump, but no motor yet. Details for how to construct and configure an engine is provided in the AGX DriveTrain section.

```
agxModel::HighLevelEngineRef engine = ...;
```

The pump constructor takes as its only argument the pump displacement.

```
agx::Real displacement(0.01);
agxHydraulics::PumpRef pump = new agxHydraulics::Pump(displacement);
```

The pump is connected to the engine and the pipe in a way similar to the way the FlowConnector was connected between the pipe and the stop valve, with the difference that here we don't explicitly suppliy the sides to connect. By default the output side of the first component is connected to the input side of the second component.

```
engine->connect(pump);
pump->connect(pipe);
```

The pipe has now, in essence, been turned into a pump. The pipe itself represents the fluid chamber inside the pump.

20.4. Run time system manipulation

Many of the hydraulic components in the library can be manipulated during runtime through their respective APIs. For some components this is an essential part of their operation. This is exemplified in the example circuit in the previous section by the stop valve whose main purpose is to open and close based on user input or some form of control logic. Examples of other properties that can be changed during runtime is pump and motor displacement, check valve orientation, relief valve pressure settings and constant flow valve target flow rate.

20.5. Data extraction

The state of each individual component can be inspected using each component's respective API. The example circuit constructed in the Building circuits section is used below.

20.5.1. Pressure

Similar to how flow rate is a property of the FlowUnits, pressure is a property of the Connectors. The toolkit does not track pressures along a pipe, only at its end points. To read the pressure at a junction use the FlowConnector's `getPressure` method.

```
agx::Real sourcePressure = junction->getPressure();
```

The call above will return the pressure at the pump outlet, i.e., the pressure where the stop valve is connected to the pump pipe. The pressure at any FlowUnit side can be read by first fetching the FlowConnector that the side is connected to. The following statement is equivalent to the previous one.

```
agx::Real sourcePressure = stop->getInputFlowConnector()->getPressure();
```

This technique can be used to read the pressure at locations where the FlowConnector was implicitly created, such as between the stop valve and the hydraulic cylinder in the example.

```
agx::Real workingPressure = stop->getOutputFlowConnector()->getPressure();
```

Be aware that not all FlowUnit sides has a connected FlowConnector and in these cases `nullptr` will be returned.

The pump connector is another subclass of PressureConnector and can therefore also return a pressure. The pressure returned is the pressure that the pump contributes to the system and not the pressure within the FlowUnit that the pump is connected to which will include pressures from connected neighboring pipes as well.

20.5.2. FlowRate

Flow rate is a property of the FlowUnits and are accessed using the `getFlowRate` method.

```
agx::Real flowRate = pipe->getFlowRate();
```

The flow rate is measured in units of volume per unit time. The toolkit does not track the flow velocity of the fluid, but the average flow velocity can be calculated by dividing the flow rate with the area of the FlowUnit.

There is no flow rate associated with FlowConnectors and, unlike the corresponding situation with pressure, there is no obvious way to find a FlowUnit to read the value from since a FlowConnector may be attached to any number of FlowUnits and the FlowUnits may be attached in any direction.

20.6. Units of measurement

It is possible to use any set of units as long as the choice is consistent across entire simulation spanning drive train, hydraulics and mechanics. This means that if geometry dimensions and rigid body positions are given in meters then FlowUnit areas must be given in meters as well, and reported flow rates will be in cubic meters per unit time.

Some helper functions, such as the RPM conversions, assume that SI units are used.

20.7. Parameter configuration

- Density - The density of the fluid, in units of mass per unit volume.
- Viscosity - The kinematic viscosity of the fluid, in units of area per time. The viscosity has a strong influence on the pressure loss over pipes, with zero viscosity resulting in no pressure loss at steady state.
- FlowConnector compliance - The compliance of the constraint held by the FlowConnector controls the compressibility of the system. The FlowConnectors will flex under pressure, which represents the flexibility of the junction itself, the expansion of the neighboring pipes and the compressibility of the fluid in those pipes. It is measured in units of volume per unit pressure.
- FlowConnector damping - Parameter that controls the rate at which pipe, junction and fluid deformations due to pressure variations are created and restored.
- Displacement - Pumps and motors have a displacement parameter that defines the relationship between flow rate in the chamber and the rotation of the shaft.
- Flow coefficient - The flow coefficient is currently only configurable for the needle valve. It is a measure of how efficiently fluid can pass through a component. The smaller this value is, the greater effect from the needle valve.
- Tank pressure - The pressure that any unconnected, i.e., tank connected, FlowUnit side is subjected to.
- …

20.8. Known limitations and assumptions

- Cavitation/vacuum inside pipes. The model assumes that all pipes and other components are always full with fluid and the constraints will ensure that this is the case. This may produce unrealistic pressures in cases where a real-world system would have produced cavitation and this may show up as negative pressures in the FlowConnectors. Common cases where this occurs is when the pump and load is working in the same direction and the pump is unable to keep up, e.g. a hydraulic cylinder is being extended faster than the pump can supply fluid and cavitation would be formed in the cylinder. Another scenario is a stop valve being closed while there is a large flow rate going through it.
- Gravity is ignored. Components don’t have elevation and all pipes are laid out horizontally.
- Assumes circular pipe cross sections and uniform area over the entire pipe.
- Does not take geometry of junctions into account. Cannot model bends or multi-pipe junctions where the angles between the pipes are important.
- No heat simulation.

20.9. Troubleshooting

The following is a listing of common causes of malfunctioning hydraulic simulations.

- Missing PowerLine::add.
- Misconfigured input/output side of unit.
- Valve mistakenly closed.
- Illegal connection (pump or motor backwards, connecting to nullptr, …).
- Zero displacement for pump/motor. Watch out for integer division.
- PistonActuator created from Prismatic without a properly configured range.
- Always check return values.

21. AGX Hydrodynamics

When an object is moving through air, water or another fluid it is affected by hydrodynamic effects such as lift, drag and added mass. These effects can be simulated using a WindAndWaterController. Only one (1) controller is needed for each simulation.

```
// Create wind and water controller and add it to the simulation.
agxModel::WindAndWaterControllerRef controller = new agxModel::WindAndWaterController();
simulation->add( controller );
```

The `WindAndWaterController` performs hydro- and aerodynamic calculations on the following objects:

- Shapes
  - Based on a mesh
  - Boxes
  - Spheres
  - Capsules
  - Cylinders
  - Cones
  - Hollow cones
  - Hollow cylinders
- Wires
- Cables

For shapes, the effects hydro- and aerodynamic are calculated for each triangle in a mesh. If it is a primitive shape, e.g. a sphere, the calculations are made on a tessellation of that primitive shape.

Wires and cables are not tessellated but divided by the segments. For each segment, the calculations are performed assuming the segment consists of the curved surface of a cylinder.

Hydrodynamic effects will be calculated if the object is in contact with water, otherwise aerodynamic calculations will be performed. For a shape this means that each triangle will be tested. If the triangle is partially submerged the triangle will be clipped creating new triangles so that each triangle is considered to be either completely in air or completely in water. For a wire or cable each segment is tested and if it is partially submerged the segment is divided in two parts, one for hydrodynamics and one for aerodynamics.

The effects that can be calculated is:

- Buoyancy - a floating force in negative gravity direction.
- Pressure drag – a force depending on the normal relative velocity.
- Viscous drag – a force depending on the tangential relative velocity.
- Lift – a force in the normal direction depending on the tangential relative velocity.
- Added mass – from the surrounding fluid. (Only for shapes, not for wires or cables)

It is possible to access the calculated forces from the different effects in the following way:

```
// Access all the forces/torques added together for a geometry
agx::Vec3 totalForce = windAndWaterController->getForce(geometry);
agx::Vec3 totalTorque = windAndWaterController->getTorque(geometry);

// Access the buoyancy force on a geometry
// Changing the ForceType to PRESSURE_DRAG, VISCOUS_DRAG, or LIFT will give us those forces instead
agx::Vec3 buoyancyForce = windAndWaterController->getForceComponent(
  geometry, agxModel::WindAndWaterController::ForceType::BUOYANCY);
// In the same way we can access torque components from different effects
agx::Vec3 buoyancyTorque = windAndWaterController->getTorqueComponent(
  geometry, agxModel::WindAndWaterController::ForceType::BUOYANCY);
```

Note

For hydrodynamic calculations the lift forces of wires and cables will be zero due to symmetry and will therefore be omitted.

Note

For the aerodynamic calculations, buoyancy, center of buoyancy and added mass will be omitted due to the low density of air.

The default values for relevant constants are specified in Table 21.1.

Table 21.1 Default values used in the simulation

Name	Value
Water density	1000
Air density	1.2
Pressure drag coefficient	0.6
Viscous drag coefficient	0.1
Lift coefficient	0.01

By changing the density of the fluid and the coefficients of the object, the `WindAndWaterController` can simulate many fluids other than water and air.

Attention

For multi threaded applications the results of hydro- and aerodynamics may be nondeterministic.

21.1. Parameters

For each shape, there are hydro- and aerodynamic parameters. These can be used to calibrate the behavior of objects in air or water. Note that the interaction between an object and a fluid is defined only by these parameters and not contact materials.

For a shape or wire, these can be reached in the following way:

```
// Hydrodynamic parameter for a shape
agxModel::WindAndWaterParametersRef hydrodynamicParameters = controller->
  getOrCreateHydrodynamicsParameters( shape );

// Aerodynamic parameters for a shape
agxModel::WindAndWaterParametersRef aerodynamicParameters = controller->
  getOrCreateAerodynamicsParameters( shape );

// Hydrodynamic parameter for a wire
agxModel::WindAndWaterParametersRef hydrodynamicParameters = controller->
  getOrCreateHydrodynamicsParameters( wire );

// Aerodynamic parameters for a wire
agxModel::WindAndWaterParametersRef aerodynamicParameters = controller->
  getOrCreateAerodynamicsParameters( wire );
```

For a cable, parameters can be stored both by cable and by segment of the cable.

```
// Hydrodynamic parameter for a cable
agxModel::WindAndWaterParametersRef hydrodynamicParameters = controller->
  getOrCreateHydrodynamicsParameters( cable );

// Aerodynamic parameters for a cable
agxModel::WindAndWaterParametersRef aerodynamicParameters = controller->
  getOrCreateAerodynamicsParameters( cable );

// Hydro- and aerodynamic parameters by segment of cable
agxCable::CableIterator iterator = cable->begin();
while ( !iterator.isEnd() )
{
  agxCollide::ShapeRef shape = iterator.getGeometry()->getShape();
  agxModel::WindAndWaterParametersRef aerodynamicParameters = controller->
    getOrCreateAerodynamicsParameters( shape );

  // Do something with the parameters
```

```
  iterator.advance();
}
```

The priority of parameters for a cable is as follows:

1. If parameters are set for a segment these parameters will be used.
2. If no parameters exists for the segment the parameters for the cable it belongs to will be used.
3. If no parameters exists for the cable default parameters will be used.

For a shape, the hydro- and aerodynamic effects will be calculated for each triangle in a mesh. If it is a primitive shape, e.g. a sphere, the tessellation can be chosen to be low, medium, high or ultra high.

```
// Tessellation level set to hydro- or aerodynamic parameters
parameters->setShapeTessellationLevel(
  agxModel::WindAndWaterParameters::ShapeTessellation::ULTRA_HIGH );
```

For a wire or a cable, the segments for calculation are defined by the resolution.

The hydro- and aerodynamic coefficients of pressure drag, viscous drag and lift can be set in the parameters, as well as a scaling parameter for buoyancy. Note that lift will not be calculated for wires and cables due to symmetry.

```
// Change the pressure drag coefficient to 0.5
parameters->setCoefficient( agxModel::WindAndWaterParameters::PRESSURE_DRAG, 0.5 );

// Change the viscous drag coefficient to 0.5
parameters->setCoefficient( agxModel::WindAndWaterParameters::VISCOUS_DRAG, 0.5 );

// Change the lift coefficient to 0.5
parameters->setCoefficient( agxModel::WindAndWaterParameters::LIFT, 0.5 );

// Change the buoyancy scaling to 0.8
parameters->setCoefficient( agxModel::WindAndWaterParameters::BUOYANCY, 0.8 );
```

To change a coefficient for all shapes of a rigid body the following methods can be used:

```
//Setting the hydrodynamic lift coefficient to zero.
agxModel::WindAndWaterParameters::setHydrodynamicCoefficient( controller, rigidBody,
  agxModel::WindAndWaterParameters::LIFT, 0 );

//Setting the aerodynamic lift coefficient to zero.
agxModel::WindAndWaterParameters::setAerodynamicCoefficient( controller, rigidBody,
  agxModel::WindAndWaterParameters::LIFT, 0 );
```

The default values for the coefficients are specified in Table 21.1. Since the viscous drag becomes negligible compared to pressure drag with increasing velocity it might be a good idea to have the viscous drag coefficient decrease, and even become zero, at high velocities.

21.2. Hydrodynamics

To enable hydrodynamic calculations a water geometry has to be defined. The geometry has to contain exactly one shape that can be either a box, a plane, a height field or a sphere.

```
// Let the water geometry to be an box of size 60x60x30 meters.
agxCollide::ShapeRef waterShape = new agxCollide::Box( 30, 30, 15 );
agxCollide::GeometryRef waterGeometry = new agxCollide::Geometry( waterShape );

// Mark as water geometry. Contacts will be disabled for the water geometry, i.e.,
// no "normal" contacts will be created, only hydrodynamic forces.
agxModel::WindAndWaterControllerRef controller = new agxModel::WindAndWaterController();
controller->addWater( waterGeometry );
```

The default value for the water density is 1000 kg/m3. By assigning a material to the geometry this value can be changed.

```
// Create material to define the density of water.
agx::MaterialRef waterMaterial = new agx::Material( "waterMaterial" );
```

```
waterMaterial->getBulkMaterial()->setDensity( agx::Real( 900 ) );

waterGeometry->setMaterial( waterMaterial );
```

When a water geometry is defined all geometries, wires and cables that overlap the water will be subject to hydrodynamic calculations which consist of buoyancy, pressure drag, viscous drag and for geometries also lift.

The `WindAndWaterController` can contain several water geometries.

Examples of hydrodynamic simulations can be seen in **tutorial_hydrodynamics**.

21.2.1. Added mass

To enable added mass calculations for a shape, a file containing static data of the shape has to be read:

```
// Initiate added mass.
hydrodynamicParameters->initializeAddedMassStorage( "fileName.dat" );
```

If there is no file that matches the specified file name an analysis of the shape will be performed creating a file with that name. Note that this analysis can be very time consuming, but will only be performed the first time.

For a wire or a cable there will be no added mass calculation.

21.2.2. Enabling Added mass for a rigid body

When a body moves in a fluid (e.g. water), parts of the fluid has to move around the body. When the body accelerates, so must also the fluid. Therefore, the force required to give a body certain acceleration is larger than it would have to be in vacuum. AGX supports the notation of added mass. The added mass enters the equations of motion except for when calculating the gravitational force, i.e. the gravitational mass is unchanged. It is also used in the mass matrix when AGX solves for constraint forces. The added mass can be specified for each axis through the API calls:

```
rigidBody->getMassProperties()->setMassCoefficients( const Vec3& coefficients );
rigidBody->getMassProperties()->setInertiaTensorCoefficients( const Vec3& coefficients );
```

These coefficients will enter the velocity update equation in the C_A matrix. Given the 6x6 mass matrix M_0 (linear mass and inertia) and the added mass coefficients C_A, the velocity from the previous time step V_k, the constraint Jacobian G and the Lagrange multipliers λ, time step h and external forces (including gravity) f_e we write the velocity update as:

$$V_{k+1} = V_k + [(1 + C_A)M_0]^{-1}[G^T\lambda + hf_e]$$

Gravity will be calculated with the original gravitational mass as: $F_g = mg$, where m is the original mass.

Note that C_A matrix with the added mass coefficient is a scale of the linear mass M_0.

21.2.3. Water flow

Water flow can be added by creating a water flow generator. The interface can be found in `agxModel::WaterFlowGenerator`.

```
/**
Interface for generating water flow and wind.
*/
class AGXMODEL_EXPORT WindAndWaterFlowGenerator : public agx::Referenced, public agxStream::Serializable
{
  public:
    /**
    Default constructor disables serialization. Any implementation
    has to enable this explicitly.
    */
    WindAndWaterFlowGenerator();

    /**
    Calculate and return wind at a given position in world.
    \param worldPoint - point in world
    \return wind velocity in world coordinates
    */
    virtual agx::Vec3 getVelocity( const agx::Vec3& worldPoint ) const = 0;
};
```

```
/**
Interface for generating water flow.
*/
class AGXMODEL_EXPORT WaterFlowGenerator : public WindAndWaterFlowGenerator
{
  public:
  /**
  Default constructor disables serialization. Any implementation
  has to enable this explicitly.
  */
  WaterFlowGenerator();
};
```

An example of how to implement a `WaterFlowGenerator` can be seen in **tutorial_hydrodynamics.cpp**.

A water flow created with an implementation of this interface can then be added to the controller.

```
// Add a water flow generator belonging to a given waterGeometry.
controller->setWaterFlowGenerator( waterGeometry, waterFlowGenerator );
```

If there are several water geometries in the simulation, the water flow generator has to be added for each geometry, otherwise that water geometry will have no water flow. It is also possible to have different water flow generators for each water geometry.

Use `ConstantWaterFlowGenerator` to create constant, uniform water flow.

```
// Create a constant water flow over a given water geometry
const agx::Vec3 flowVelocity( 5, 0, 0 );
agxModel::ConstantWaterFlowGenerator waterFlowGenerator =
  new agxModel::WaterFlowGenerator( flowVelocity );
controller->setWaterFlowGenerator( waterGeometry, waterFlowGenerator );
```

21.2.4. Water wrapper

The `WaterWrapper` is used to extract information about the water in the simulation based on the water geometry and, if present, a `FlowGenerator`. It is also possible to use it as a base class to be able to modify how some of this information is extracted. The water wrapper controls velocity, density and surface height of the water. For full extent, see **WindAndWaterController.h**.

The water wrapper can then be assigned to a water geometry in the `WindAndWaterController`.

```
windAndWaterController->setWaterWrapper( waterGeometry, waterWrapper );
```

For instance, the height of the water surface is by default considered to be the same as the surface of the water geometry. Overriding this enables the use of complicated water surfaces, such as waves, without the need of a dynamic height field. An example of this can be seen in **tutorial_hydrodynamics.cpp**.

Note

For the collision detection to work correctly it is important that the water surface never exceeds the extents of the water geometry.

21.2.5. Center of buoyancy

The center of buoyancy describes the center of mass of the fluid displaced by a geometry or body. It is also the point where the buoyancy forces should act. The center of buoyancy calculations are done separately from the buoyancy ones, and are automatically turned off to save some computational time. To enable the center of buoyancy calculations and get the value of it and/or of the corresponding volume:

```
// Enable center of buoyancy for a geometry
windAndWaterController->setEnableCenterOfBuoyancy(geometry, true);
// Access center of buoyancy value. The value in the Vec3 will be overwritten with center of
// buoyancy, if there is any value to set. The method returns true when it has set the value.
agx::Vec3 centerOfBuoyancy;
bool valueWasSet = windAndWaterController->getCenterOfBuoyancy(geometry, centerOfBuoyancy);
// The volume parameter will be overwritten with the volume of the displaced fluid.
agx::Real volume;
bool volumeWasSet = windAndWaterController->getSubmergedVolume(geometry, volume);
```

Note that when a geometry or body is not in the water, there is no valid value for center of buoyancy, and the method to get it cannot set any value. The calculated volume, however, will be valid as long as center of buoyancy calculations are enabled.

21.3. Aerodynamics

Aerodynamics can be enabled to get the effect of air resistance or wind. By default, these calculations are disabled but they can be enabled for any geometry, rigid body, wire or cable.

```
// Enable aerodynamics for a geometry.
controller->setEnableAerodynamics( geometry, true );

// Enable aerodynamics for a rigid body.
controller->setEnableAerodynamics( rigidBody, true );

// Enable aerodynamic calculations for a wire.
controller->setEnableAerodynamics( wire, true );

// For a cable aerodynamics can be enabled by cable or by segment
// Enable aerodynamics for a cable
controller->setEnableAerodynamics( cable, true );

// Enable aerodynamics for segment of a cable.
agxCable::CableIterator iterator = cable->begin();
while ( !iterator.isEnd() )
{
  agxCollide::GeometryRef geometry = iterator.getGeometry();
  controller->setEnableAerodynamics( geometry, true );

  iterator.advance();
}
```

It is also possible to enable aerodynamics for all objects in the simulation.

```
// Enable aerodynamics for all objects in the simulation
controller->setEnableAerodynamics( true );
```

Note that this can be time consuming. For performance reasons it is best to limit the objects enabled for aerodynamics.

The aerodynamic calculations consist of pressure drag, viscous drag and for geometries also lift. Added mass and buoyancy are omitted due to the low density of air.

The default value for air density is 1.2 kg/m^3 and can be changed in the WindAndWaterController.

```
// Set the air density to 1.4 instead of the default value of 1.2
controller->setAirDensity( agx::Real( 1.4 ) );
```

21.3.1. Wind

Wind can be added to the simulation by creating a wind generator. The interface is found in **agxModel::WindGenerator**.

```
/*
Copyright 2007-2025. Algoryx Simulation AB.

All AGX source code, intellectual property, documentation, sample code,
tutorials, scene files and technical white papers, are copyrighted, proprietary
and confidential material of Algoryx Simulation AB. You may not download, read,
store, distribute, publish, copy or otherwise disseminate, use or expose this
material unless having a written signed agreement with Algoryx Simulation AB, or having been
advised so by Algoryx Simulation AB for a time limited evaluation, or having purchased a
valid commercial license from Algoryx Simulation AB.

Algoryx Simulation AB disclaims all responsibilities for loss or damage caused
from using this software, unless otherwise stated in written agreements with
Algoryx Simulation AB.
*/

/**
Interface for generating water flow and wind.
*/
```

```
class AGXMODEL_EXPORT WindAndWaterFlowGenerator : public agx::Referenced, public agxStream::Serializable
{
  public:
    /**
    Default constructor disables serialization. Any implementation
    has to enable this explicitly.
    */
    WindAndWaterFlowGenerator();

    /**
    Calculate and return wind at a given position in world.
    \param worldPoint - point in world
    \return wind velocity in world coordinates
    */
    virtual agx::Vec3 getVelocity( const agx::Vec3& worldPoint ) const;
};

/**
Interface for generating wind effects.
*/
class AGXMODEL_EXPORT WindGenerator : public WindAndWaterFlowGenerator
{
  public:
    /**
    Default constructor disables serialization. Any implementation
    has to enable this explicitly.
    */
    WindGenerator();

    /**
    Called before update of \p geometry.
    \param geometry - geometry to be updated
    */
    virtual void prepare( const agxCollide::Geometry* geometry );
};
```

A wind created with an implementation of this interface can then be added to the controller.

```
//Adding a windGenerator to the WindAndWaterController
  controller->setWindGenerator( windGenerator );
```

Use ContantWindGenerator to create a constant, uniform wind.

```
// Create a constant wind, with a given wind direction and magnitude.
const agx::Vec3 windVelocity = agx::Vec3( -15, 0, 0 );
agxModel::ConstantWindGeneratorRef windGenerator = new agxModel::ConstantWindGenerator( windVelocity );
controller->setWindGenerator( windGenerator );
```

An example of using a custom wind to create shadowing can be seen in **tutorial_wind.cpp** together with other examples. There is also an example of a windmill that is affected by aerodynamics in **windmill.agxPy**.

21.4. Wires and Cables

As explained above the hydro- and aerodynamic effects can be applied to wires with the effects of buoyancy, pressure drag and viscous drag. For a full example with wires and cables, see **tutorial_wireWindAndWater.cpp** and **tutorial_cableWindAndWater**.

21.5. Pressure field renderer

The pressure field renderer is used to visualize the pressure acting on a shape by the hydro- and aerodynamic forces.

```
// Create a pressure field renderer and give it to the controller
agx::Real scale = 1.01;
agxOSG::PressureFieldRendererRef pfr = new agxOSG::PressureFieldRenderer( root, scale );
controller->registerPressureFieldRenderer( pfr, shape );
```

The pressure field renderer is not available for wires and cables.

21.6. Known limitations

- The object will not affect the water in any way. This means that:
    - An object will not be affected by another objects' movement nearby.
    - Object movement in water will not produce waves.
- All parts below the water surface will be subject to hydrodynamic calculations. This means that e.g. an empty bowl floating in water will experience a downward pressure from water if the inside of the bowl is beneath the surface level.
- Any geometry or wire segment can only be affected by one water geometry at any given time.
- Since these effects are calculated per shape, an object consisting of several shapes or geometries can lead to unwanted behavior.
- Wires and cables are considered as lines with a thickness. This means that a segment crossing the water surface will have correct behavior but a segment laying along the water surface will not.

22. AGX Vehicle

The AGX Vehicle module provides various functionality for simulating vehicles and machines.

- Tracks
  - Full-Degree Tracks Model
  - Low-Degree Tracks Model
  - agxVehicle::Wheel
  - agxVehicle::TrackWheel
  - agxVehicle::Track
- WheelJoint
  - Connecting to a power-line
- Steering
  - Geometrical parameters
  - Building steering
  - Steering manipulation

22.1. Tracks

A tracked vehicle is an object containing one or more tracks. Each track contains two or more wheels and an arbitrary number of treads.

AGX Dynamics provides two models for simulating tracked vehicles, each suited for different use cases.

- **Full-Degree Tracks Model** – detailed per-node rigid body simulation, high realism but computationally heavy.
- **Low-Degree Tracks Model** – simplified, wheel-based model, faster and more stable.

The Track class has two constructors, and the choice of constructor determines which track model is active. The recommended constructor is the one that takes a *reference rigid body* (typically the vehicle chassis when simulating vehicles). Using this constructor will initialize the *Low-Degree Tracks Model* by default. If desired, this model can be switched to the *Full-Degree Tracks Model* by explicitly enabling it with *setEnableFullDegreeModel(true)*.

If instead you use the constructor without a reference rigid body, then only the Full-DoF Model is supported, and that will be the model used during simulation.

```
agx::RigidBodyRef referenceRigidBody = new agx::RigidBody();
agxVehicle::TrackRef track = new agxVehicle::Track(referenceRigidBody,
                                                   numNodes,
                                                   width,
                                                   thickness);
```

This sections below describes how to create and configure tracks, add wheels and route them, set initial tension and node geometry, tune friction and material parameters, merge nodes for performance, and adjust track properties such as track stiffness, damping, and hinge range.

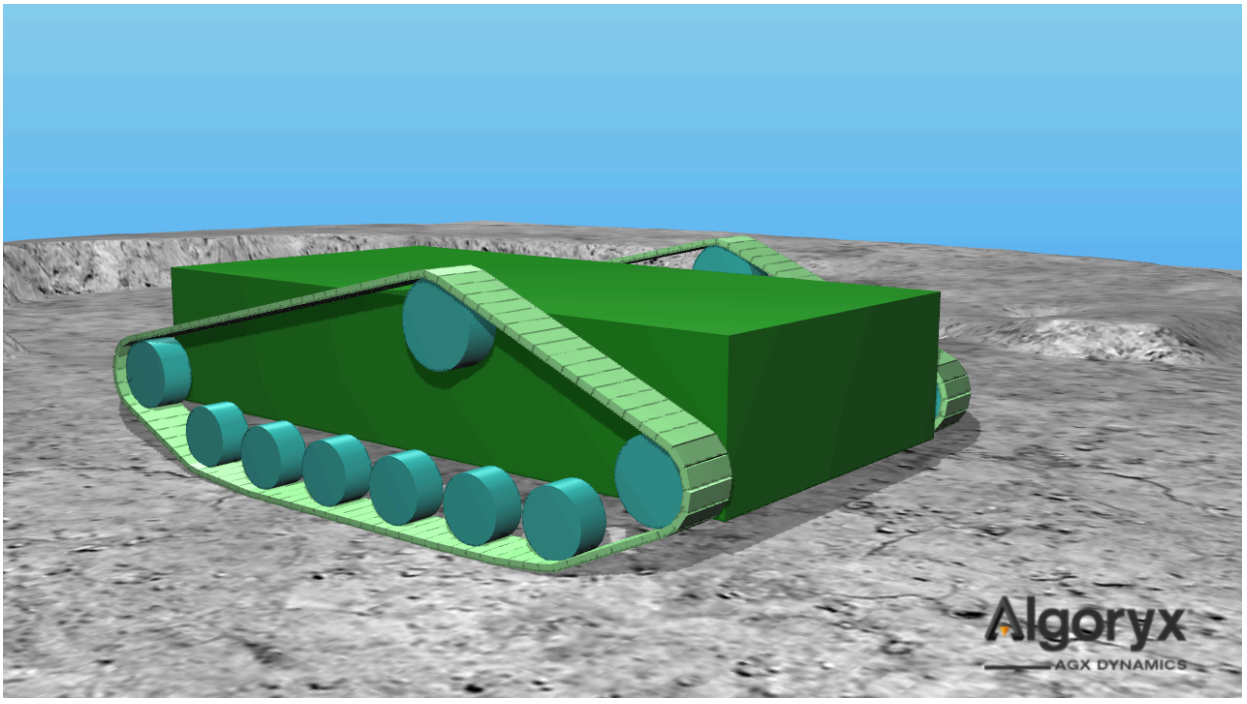


Fig. 22.1 Tracked vehicle with two tracks, each track has nine wheels and 100 treads.

22.1.1. Full-Degree Tracks Model

The **Full-Degree Tracks Model** (`agxVehicle::Track`) is based on a lumped-element approach with multiple degrees of freedom. Each track shoe is represented by a rigid body connected through rotational joints, accurately capturing the physical constraints between components and their interaction with terrain and other bodies. This model delivers high physical realism and is well suited for applications where detailed track behavior is important.

However, its complexity can make setup challenging, especially for fast-moving vehicles. High track speeds require small simulation time steps to maintain constraint accuracy, which can lead to numerical instabilities in demanding scenarios.

22.1.2. Low-Degree Tracks Model

The **Low-Degree Tracks Model** is optimized for high-speed vehicle simulations and uses a reduced number of degrees of freedom. In this model, the track nodes and hinge joints are replaced by a smaller set of bodies and constraints, with degrees of freedom proportional to the number of wheels rather than the number of track nodes.

Between each wheel pair, a sensor box body is added and prismatic-constrained to the chassis. Track tension is modeled using massless wire constraints: a global tension element wraps around all wheels, and local tension constraints connect each pair of track-contacting wheels. Track-ground interaction is approximated using the wheels and sensor boxes, which together define the track topology and tension forces.

This approach improves numerical stability and performance for high-speed simulations while retaining the essential dynamics of tracked vehicles.

22.1.2.1. agxVehicle::TrackImplementation

The default model depends on what constructor you use when you create your tracks model but it is recommended to use the constructor described below with a given reference rigid body, this will initiate the *Low-Degree Tracks Model* (vs the *Full-Degree Tracks Model*).

Creating Tracks

```
agx::RigidBodyRef referenceRigidBody = new agx::RigidBody();
agxVehicle::TrackRef track = new agxVehicle::Track(referenceRigidBody,
                                                   numNodes,
                                                   width,
                                                   thickness);
```

Switching Between Models

You can switch between the Full-Degree Tracks Model and the Low-Degree Tracks Model at runtime:

```
// Enable or disable the full degrees-of-freedom model
track->setEnableFullDegreeModel(true);   // Activates full DOF
track->setEnableFullDegreeModel(false);  // Activates low DOF (default)
```

```
bool isFullDofActive = track->getEnableFullDegreeModel();
```

Using Custom Track Implementations

You can also explicitly assign a custom track implementation. In the example below we use the default Low-Degree Tracks Model:

```
// Assign a runtime implementation
agxVehicle::TrackImplementation* customImpl =
        new agxVehicle::LowDofTrackImplementation(referenceRigidBody);
track->setTrackImplementation(customImpl, true);
```

This allows fine-grained control over how the track is simulated, making it possible to use a custom implementation or switch between implementations dynamically.

22.1.2.2. agxVehicle::LowDofTrackImplementation

The **LowDofTrackImplementation** is the default implementation when `setEnableFullDegreeModel(false)` is used.

In this model:

- Track node bodies and hinge DOFs are removed and the track nodes are approximated with boxes between the wheels.
- The number of DOFs is proportional to the number of wheels rather than the number of nodes.
- Track nodes are animated based on wheel geometries and rotation speeds, enabling realistic visual rendering.

22.1.3. agxVehicle::Wheel

Base class of an abstraction of a wheel of arbitrary geometry. An `agxVehicle::Wheel` is constructed given *radius*, *rigid body* and a *relative frame*.

Definition:

> A wheel has a center, a radius and a rotation axis.

Since the given rigid body can be any user defined object, the *relative frame* defines the center and rotation axis of the wheel and is given in the frame of the wheel body.

The resulting frame should have the **rotation axis** about *y* and **up axis** along *z*.

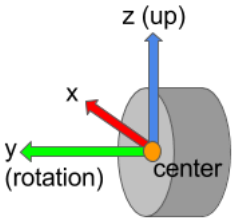


Fig. 22.2 Cylindrical wheel with wheel frame defining center position, rotation and up axes.

22.1.4. agxVehicle::TrackWheel

An `agxVehicle::TrackWheel` is an `agxVehicle::Wheel` and inherits the definitions of center, radius and reference frame. The track wheel comes with a few more important features; *model* and *property*. This means that each track wheel is of a certain model and contains a set of properties.

Model	Description
Sprocket	Driving wheel - this wheel's shaft is connected to the powerline/motor. Sprockets are normally geared where the treads wont slip while in good touch with the surface of the sprocket.
Idler	Geared, non-powered wheel. Similar surface properties as the sprocket but this model isn't powered.
Roller	Intermediate track-return or road wheel.

The properties associated to the track wheels are by default assigned given the model.

Property	Description	Model
Merge nodes	Treads/nodes are merged to the wheel when in contact, mimicking geared interaction (or infinite friction).	Sprockets and idlers

Property	Description	Model
Split segments	Splits merged segments of treads/nodes when in contact with wheels with this property.	Optional
Move nodes to rotation plane	If enabled - when a node is merged to the wheel, move the node into the plane defined by the wheel center position and rotation axis. This will prevent the tracks from sliding of its path but all wheels with MERGE_NODES must be aligned.	Optional
Move nodes to wheel	Similar to **move nodes to rotation plane** but with this property enabled the node will also be placed on the surface of the wheel.	Optional

Example of a cylindrical track wheel, trivially created with *reference frame* oriented with the rigid body. The wheel is attached in world with a hinge about the rotation axis and zero rotation axis offset:

```
const auto wheelRadius = 0.45;
const auto wheelHeight = 0.35;

agx::RigidBodyRef wheelRb = new agx::RigidBody( "wheelRb" );
wheelRb->add( new agxCollide::Geometry( new agxCollide::Cylinder( wheelRadius, wheelHeight ) ) );

agxVehicle::TrackWheelRef wheel = new agxVehicle::TrackWheel( agxVehicle::TrackWheel::SPROCKET,
                                                              wheelRadius,
                                                              wheelRb );
// Attaching the wheel with a hinge named "hinge" in world (nullptr) and
// 0.0 meters offset along the rotation axis.
auto hinge = wheel->attachHinge( "hinge", nullptr, 0.0 );
hinge->getMotor1D()->setEnable( true );
hinge->getMotor1D()->setSpeed( 1.0 );

simulation->add( wheelRb );
simulation->add( hinge );
```

It's hard to create an example with a more complex setup with non-identity reference frame. To verify the wheel frame, from case to case, render the frame using `wheel->getTransform()`, for example:

```
#include <agxRender/RenderSingleton.h>
...
  agxRender::debugRenderFrame( wheel->getTransform(), 0.25, agxRender::Color::Red() );
```

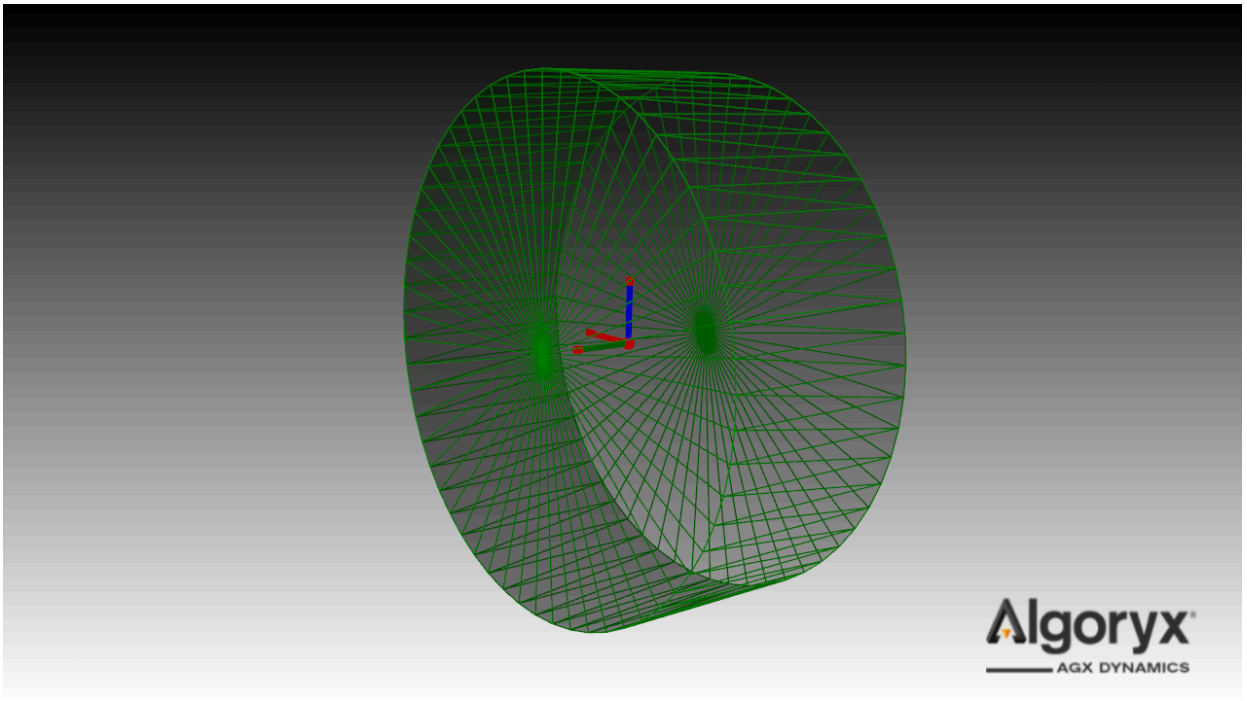
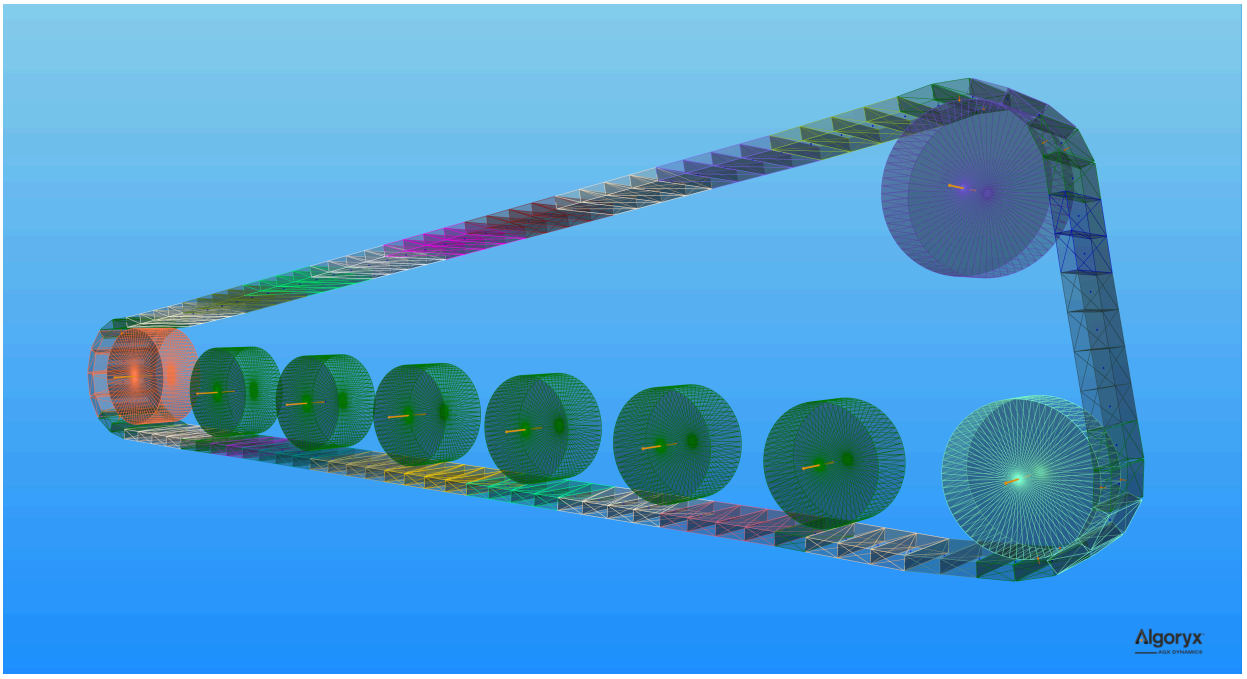


Fig. 22.3 Track wheel with radius 0.45 m and height 0.35 m. Hinged in world with hinge axis along the green (wheel) rotation axis.

22.1.5. agxVehicle::Track

Assembly object representing a continuous track with a given number of treads/nodes.



22.1.5.1. Initialization

An *agxVehicle::Track* is instantiated given the *referenceRigidBody* desired *number of nodes*, *width*, *thickness* of each node and *initialTension*. The width and thickness are both reference values - i.e., not actual, meaning the geometry may have different dimensions. The thickness value is used place the nodes as close as possible to the surface of the wheels.

The *initialTension* defines the tension in the track after it is initiated. This gives an amount by which each node is shortened to generate tension in the track and wheel system. Ideally, the track tension equals the shortening distance multiplied by the track tensile stiffness. Because contacts and other dynamic effects are present, the exact tension after initialization cannot be predicted. A scale factor of four is applied to approximately compensate for frictional effects, and the initial node distance is determined from this scale factor, the specified tension, and the tensile stiffness of the track.

The *referenceRigidBody* is the reference body that this track is attached to, normally the chassis of a vehicle. Note that the track is routed around track wheels that are connected directly or via links to this rigid body.

The ‘‘unknown’’ parameter is the length of each node. The length of the nodes is calculated during initialization to properly fit the track round the wheels.

22.1.5.1.1. Routing with wheels

Routing of the tracks is made by adding wheels to the track. Given number of nodes in the track, thickness of the nodes and wheel configuration (transform and radius), the track routing algorithm will calculate the length each node should have for the track to go outside all wheels.

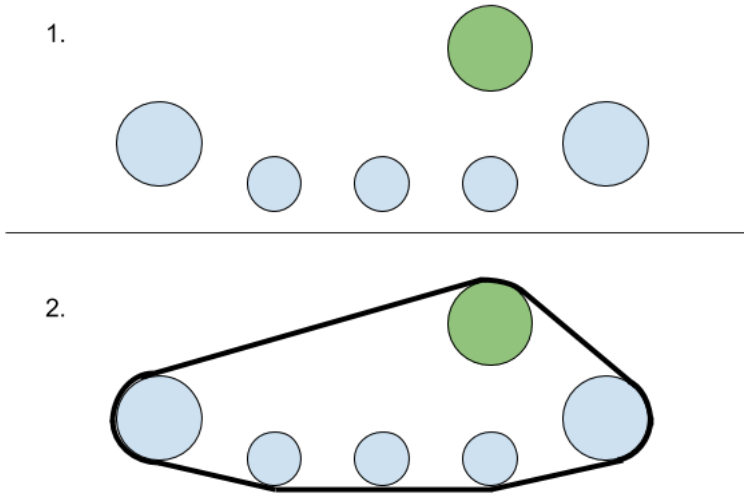


Fig. 22.4 Initial wheel configuration (1) and resulting track route (2).

Note

It doesn’t matter in which order the wheels has been added to the tracks.

Note

It’s currently only possible to initialize the tracks in a convex shape.

Small example given an array of wheel bodies that has been positioned and an array of wheel radii. The first wheel will be set to be the sprocket, all others are rollers:

```
// Use your chassis as the rigid body, or nullptr if the track is attached to the world
agx::RigidBodyRef referenceRigidBody = new agx::RigidBody();
agxVehicle::TrackRef track = new agxVehicle::Track( referenceRigidBody,
                                                    numNodes,
                                                    width,
                                                    thickness,
                                                    InitialTrackTension { 2e4 } );
for ( agx::UInt i = 0; i < wheelBodies.size(); ++i ) {
  agxVehicle::TrackWheelRef wheel = new agxVehicle::TrackWheel( i == 0 ?
                                                                  agxVehicle::TrackWheel::SPROCKET :
                                                                  agxVehicle::TrackWheel::ROLLER,
                                                                wheelRadius[ i ],
                                                                wheelBodies[ i ] );
  track->add( wheel );
}

// The track will be initialized when added to a simulation if
// agxVehicle::Track::initialize hasn't been called explicitly.
simulation->add( track );
```

22.1.5.1.2. Reference wheel

The reference wheel is used as reference during initialization of the tracks. The nodes will be positioned in the plane defined by the reference wheel position and rotation axis.

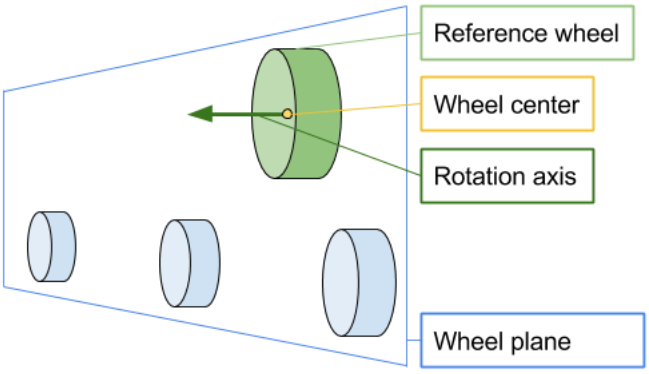


Fig. 22.5 Plane defined by the reference wheel center position and rotation axis.

The reference wheel is defined to be the first sprocket wheel (if any), otherwise first idler wheel (if any), otherwise the first roller.

Definition:

The **reference wheel** is the first sprocket wheel added to a track.

22.1.5.1.3. Initial tension

If the track finds a solution to the initial configuration and the actual geometry of the nodes matches the input thickness - the tension on the tracks will be close to zero. If the vehicle/object doesn’t have e.g., a tensioner, it’s possible to initialize the tracks with tension.

The initial tension parameter can be specified as either a distance or a force. Using a force value is less precise, as it is difficult to establish an initial state where the resulting tension exactly matches the specified force without simulating the system to equilibrium. When a force is provided, the initial node distance is approximated from the given tension and the tensile stiffness of the track, scaled by a factor of four.

A positive initial node distance shortens the spacing between nodes during initialization, increasing tension. A negative value increases the spacing between nodes, reducing or eliminating tension.

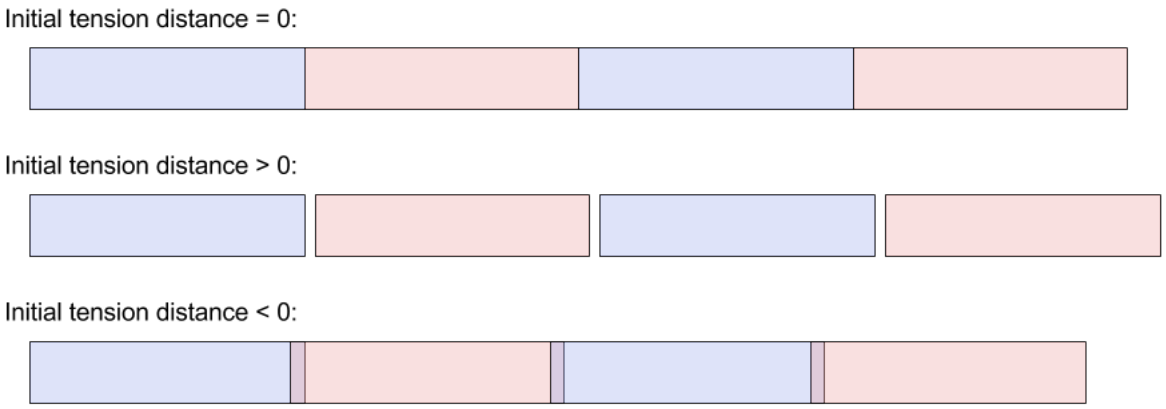


Fig. 22.6 Initial tension distance when zero (no tension), positive (initially stretched) and negative (initially compressed).

22.1.5.1.4. Node geometry

As for the wheels, the geometry of the nodes are completely arbitrary. Box is used by default.

Create geometry callbacks are fired when the node has been positioned. The z-axis of the node body is pointing in the direction of the track and the model center must be at the begin position of the node. The default ‘createGeometryBox’’ function looks like this:

```
void createGeometryBox( const TrackNode& node )
{
  // Node half extents = (0.5 * thickness, 0.5 * width, 0.5 * length)
  agxCollide::BoxRef box = new agxCollide::Box( node.getHalfExtents() );
  agxCollide::GeometryRef geometry = new agxCollide::Geometry( box );
  // Model center is begin of node, translate geometry forward 0.5 * length (along z).
  const auto transform = agx::AffineMatrix4x4::translate( 0,
                                                          0,
                                                          node.getHalfExtents().z() );
  geometry->setLocalTransform( transform );
  node.getRigidBody()->add( geometry );
}
```

Adding the track to the simulation without calling initialize beforehand is equivalent to call initialize given ‘’createGeometryBox’’ and then adding the track to the simulation:

```
// Default initialization...
simulation->add( track );
// ...is equivalent to:
anotherTrack->initialize( agxVehicle::utils::createGeometryBox );
simulation->add( anotherTrack );
```

Example of two tracks where one is initialized with several sphere shapes as geometry (initialize function) and the other with capsules (interface):

```
class NodeCapsuleInitializer : public agxVehicle::TrackNodeOnInitializeCallback
{
  public:
    NodeCapsuleInitializer() {}

    virtual void onInitialize( const agxVehicle::TrackNode& node ) override
    {
      agxCollide::CapsuleRef capsule = new agxCollide::Capsule( 0.5 * node.getWidth(),
                                                                node.getLength() );
      node.getRigidBody()->add( new agxCollide::Geometry( capsule ),
                                agx::AffineMatrix4x4::rotate( agx::Vec3::Y_AXIS(),
                                                              agx::Vec3::Z_AXIS() ) *
                                agx::AffineMatrix4x4::translate( 0, 0, 0.5 * node.getLength() ) );
    }

  protected:
    virtual ~NodeCapsuleInitializer() {}
```

```
};

const agx::Real sphereRadius = 0.1;
// Creates track with five wheels, 60 nodes and width = thickness = 2.0 * sphereRadius.
agxVehicle::TrackRef sphereTrack = createTrack( 2.0 * sphereRadius, 2.0 * sphereRadius );
agxVehicle::TrackRef capsuleTrack = createTrack( 2.0 * sphereRadius, 2.0 * sphereRadius );

sphereTrack->setPosition( 0, -1, 0 );
capsuleTrack->setPosition( 0, 1, 0 );

// Initializing the sphere track with several spheres as geometry.
sphereTrack->initialize( [sphereRadius]( const agxVehicle::TrackNode& node )
{
  const auto offset = 0.5 * sphereRadius;

  const auto numSpheres = std::max( agx::UInt( node.getLength() / offset +
                                               agx::Real( 0.5 ) ),
                                    agx::UInt( 1 ) );
  const auto dl = node.getLength() / agx::Real( numSpheres );
  agxCollide::GeometryRef geometry = new agxCollide::Geometry();
  auto pos = agx::Vec3( 0, 0, 0 );
  auto dir = agx::Vec3::Z_AXIS();
  for ( agx::UInt i = 0; i < numSpheres; ++i ) {
    geometry->add( new agxCollide::Sphere( sphereRadius ),
                   agx::AffineMatrix4x4::translate( pos ) );
    pos += dl * dir;
  }

  node.getRigidBody()->add( geometry );
} );

// Initializing the capsule track with one capsule per node.
capsuleTrack->initialize( new NodeCapsuleInitializer() );

simulation->add( sphereTrack );
simulation->add( capsuleTrack );
```

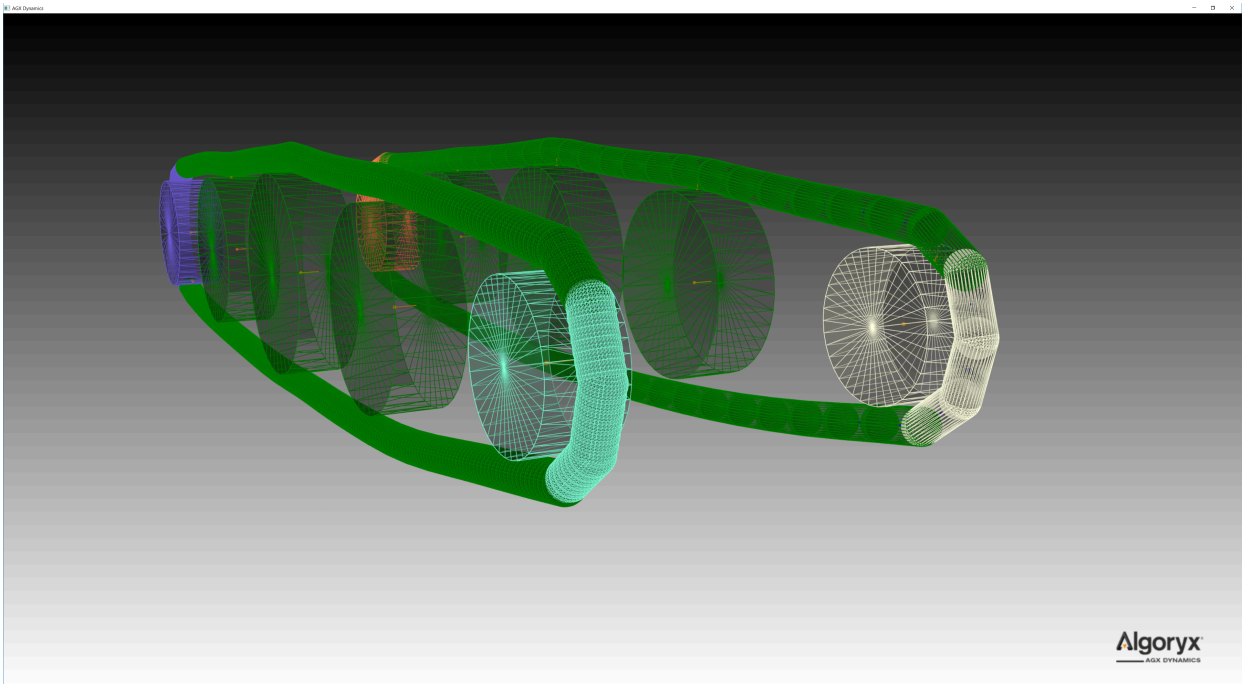


Fig. 22.7 Two tracks initialized with different geometries. Several spheres per node (left) and one capsule per node (right).

Each node doesn't have to have identical shape/geometry. This example is from tutorial_trackedVehicle.cpp:buildTutorial2 where every fourth node is thicker than the three before:

```
// Two, different thickness values along the track. Every forth node is thicker than the three before.
const agx::Real defaultThickness = 0.05;
```

```
const agx::Real thickerThickness = 0.09;

agxVehicle::TrackRef track = new agxVehicle::Track( 120, 0.4, defaultThickness );
track->add( createTrackWheel( agxVehicle::TrackWheel::Model::SPROCKET, 0.5, 0.3, agx::Vec3( 0, 0, 0 ) ) );
track->add( createTrackWheel( agxVehicle::TrackWheel::Model::ROLLER, 0.6, 0.3, agx::Vec3( 2, 0, 0 ) ) );
track->add( createTrackWheel( agxVehicle::TrackWheel::Model::ROLLER, 0.7, 0.3, agx::Vec3( 4, 0, 0 ) ) );
track->add( createTrackWheel( agxVehicle::TrackWheel::Model::ROLLER, 0.6, 0.3, agx::Vec3( 6, 0, 0 ) ) );
track->add( createTrackWheel( agxVehicle::TrackWheel::Model::IDLER, 0.4, 0.3, agx::Vec3( 8, 0, 0 ) ) );

// Initialization callback when the segmentation of the track has been made.
// When a callback is given to 'initialize' it's up to this callback to assign
// geometries/shapes to the nodes. Note that the track has calculated a length
// of the nodes that has to be taken into account.
agx::UInt counter = 0;
track->initialize( [&]( const agxVehicle::TrackNode& node )
{
  agx::Real heightOffset = 0.0;
  agx::Real thickness = defaultThickness;
  // For every fourth node we add a thicker box.
  if ( ( counter++ % 4 ) == 0 ) {
    thickness = thickerThickness;
    heightOffset = -0.5 * ( thickerThickness - defaultThickness );
  }
  node.getRigidBody()->add( new agxCollide::Geometry( new agxCollide::Box( 0.5 * thickness,
                                                                           0.25,
                                                                           0.5 * node.getLength() ) ),
                            agx::AffineMatrix4x4::translate( heightOffset, 0, node.getHalfExtents().z() ) );
} );
```

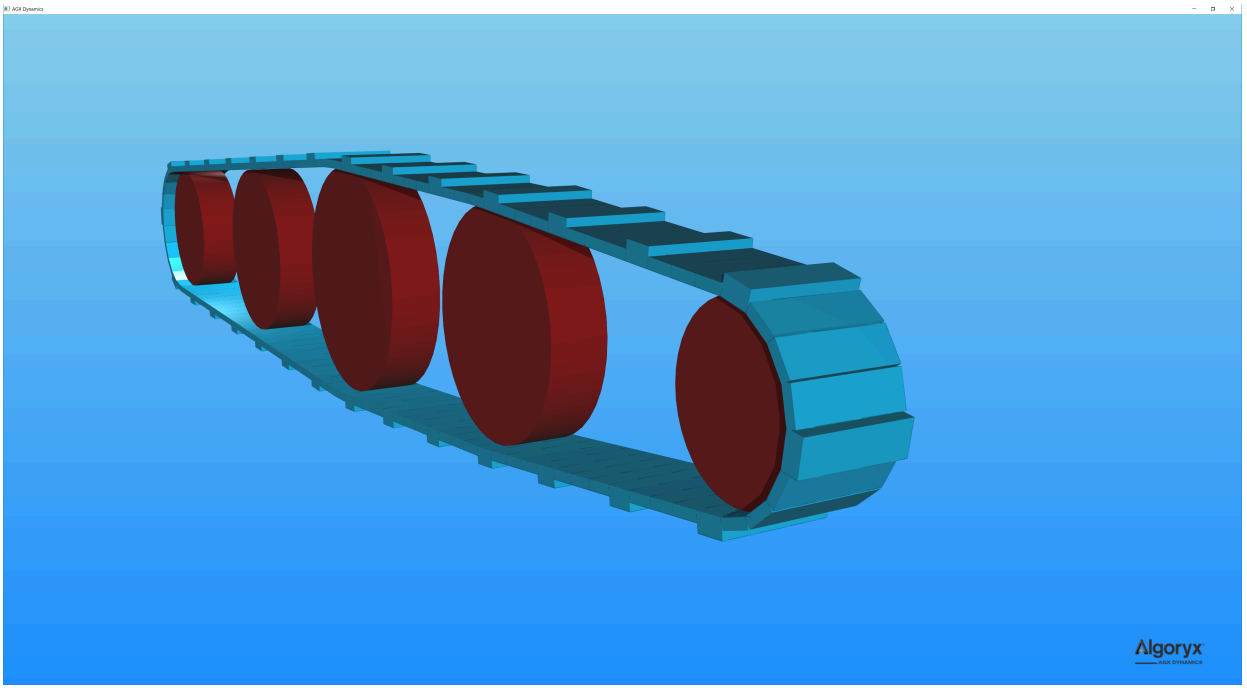


Fig. 22.8 Track initialized with different shapes/geometries along the track. Every fourth geometry is larger than the three before.

22.1.5.1.4.1. Accessing the nodes in a track

When the track has been initialized it's possible to iterate the nodes in the track in a few different ways:

```
for ( agx::UInt i = 0; i < track->getNumNodes(); ++i ) {
  agxVehicle::TrackNode* node = track->getNode( i );
  ...
}

for ( agxVehicle::TrackNode* node : track->nodes() ) {
  ...
}
```

Since the track is continuous, i.e., no actual start or end, the use of ranges is more convenient:

```
// Iterating all nodes.
for ( agxVehicle::TrackNode* node : track->nodes( { 0u, track->getNumNodes() } ) ) {
  ...
}

// Iterating segments.
const auto numNodesPerSegment = 5u;
for ( agx::UInt nodeIndex = 0; nodeIndex < track->getNumNodes(); nodeIndex += numNodesPerSegment ) {
  agxVehicle::Track::NodeRange range{ track->getIterator( nodeIndex ),
                                      track->getIterator( nodeIndex + numNodesPerSegment ) };
  for ( agxVehicle::TrackNode* node : range ) {
    // Do stuff with the node in range.
  }
}

// Since continuous, this is valid to iterate six nodes in a track with 100 nodes.
auto range = agxVehicle::Track::NodeRange{ track->getIterator( 1234u ),
                                           track->getIterator( 1234u + 6u ) };
for ( agxVehicle::TrackNode* node : range ) {
  std::cout << node->getIndex() << std::endl;
}

// Prints:
//   34
//   35
//   36
//   37
//   38
//   39
```

22.1.5.1.4.2. Simulation parameters

Given an initialized track, there are some special handling regarding the interaction between the track and the wheels:

- **Sprockets:** Instead of frictional contacts, nodes will merge with the sprocket - mimicking infinite friction or geared wheel versus track.
- **Idlers:** Default behavior is identical to the sprockets.
- **Rollers:** Frictional contacts. If computational performance is important, make sure the contact material between the rollers and the track nodes has been properly tweaked. E.g, for direct friction models - high surface viscosity.

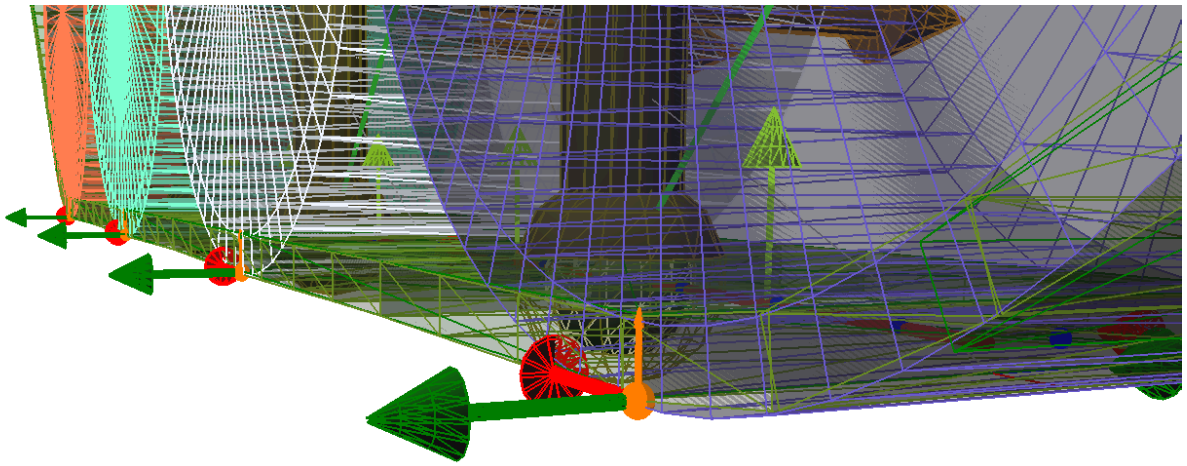
22.1.5.1.5. Contact materials and friction models

The default friction model and contact material normally isn't suitable for simulating track versus ground interactions. Mainly because the default friction box is oriented with the world frame and we're interested in separating the contact parameters *along* the track (traction) and *orthogonal to* the track (transverse slip/turning behavior).

For tracked systems, use one of the *track friction models*:

- `agxVehicle::TrackBoxFrictionModel`
- `agxVehicle::TrackScaleBoxFrictionModel`
- `agxVehicle::TrackIterativeProjectedConeFrictionModel`

These models automatically determine the friction directions from the interacting track bodies/geometries, i.e., they do **not** require an external reference frame (unlike the legacy oriented friction models). This also means a single friction model instance can be shared across an arbitrary number of tracked vehicles/tracks without per-chassis setup.



Algoryx AGX DYNAMICS

Fig. 22.9 Primary (red) and secondary (green) directions determined by the *track friction model*. The primary direction is tangent to the track (the track's positive direction). The track's positive direction is defined by the rotation axis of the main wheel/sprocket. The sign of the directions is not important for the simulation, what matters is alignment: The primary friction coefficient controls traction along the track, while the secondary friction coefficient controls transverse slip and turning.

Track friction model overview

`agxVehicle::TrackBoxFrictionModel`

Box friction model with track-aligned friction directions. Recommended default for performance/stability when simulating track-ground traction using *primary/secondary* parameters.

Feature-wise it supports the same key parameters previously used with `agx::ConstantNormalForceOrientedBoxFrictionModel` (Section 11.16.3.5):

- *constant normal force magnitude* (per-contact friction bound based on a supplied normal force estimate)
- optional `scaleWithDepth` to scale that bound with contact depth

`agxVehicle::TrackScaleBoxFrictionModel`

Scale box friction model with track-aligned friction directions. The nonlinear response of the friction force given the normal force is computationally more expensive but results in accurate friction. See ScaleBoxFrictionModel for further information about this type of friction model.

`agxVehicle::TrackIterativeProjectedConeFrictionModel`

Iterative projected cone friction model with track-aligned friction directions. Most accurate, and most computationally expensive, friction model. See IterativeProjectedConeFriction for more details regarding this type of friction model.

General recommendations

Normally a significant amount of nodes are in contact with the ground/floor/other objects, resulting in many contact points. For best numerical and behavioral performance:

- Use a direct friction model solve type (`agx::FrictionModel::DIRECT` or `agx::FrictionModel::DIRECT_AND_ITERATIVE`) for the track-ground contact material when possible.
- Prefer `agxVehicle::TrackBoxFrictionModel` for track-ground interactions, optionally with an estimate of the normal force for each contact point. Enable `scaleWithDepth` if the system is within bounds, performance wise.
- Set the *surface viscosity* as high as possible but low enough for the object to not slide with constant speed.
- Use different friction coefficients and surface viscosity *along* and *orthogonal to* the tracks – usually higher traction (and potentially lower viscosity) along the tracks, and slightly lower transverse traction with higher viscosity orthogonal to the tracks for smooth turning.
- Set *restitution* to zero for better stability and continuous contacts.

Example from *tutorial_trackedVehicle.cpp* (vehicle shown in Fig. 22.1):

```
// Setting up materials.
const agx::Real trackDensity                   = 2000.0;
const agx::Vec2 trackGroundFrictionCoefficients = agx::Vec2( 1000.0, 10 );
const agx::Vec2 trackGroundSurfaceViscosity     = agx::Vec2( 1.0E-7, 8.0E-6 );
const agx::Real trackWheelFrictionCoefficient   = 1.0;
const agx::Real trackWheelSurfaceViscosity      = 1.0E-5;
```

Note the differences in friction coefficients and viscosity. These has to be chosen to match the performance criterion, stability and behavior. Further:

```
agx::MaterialRef groundMaterial = new agx::Material( "groundMaterial" );
agx::MaterialRef trackMaterial  = new agx::Material( "trackMaterial" );
agx::MaterialRef wheelMaterial  = new agx::Material( "wheelMaterial" );

trackMaterial->getBulkMaterial()->setDensity( trackDensity );

ground->setMaterial( groundMaterial );

trackedVehicle->getRightTrack()->setMaterial( trackMaterial );
for ( const auto wheel : trackedVehicle->getRightTrack()->getWheels() )
  agxUtil::setBodyMaterial( wheel->getRigidBody(), wheelMaterial );

trackedVehicle->getLeftTrack()->setMaterial( trackMaterial );
for ( const auto wheel : trackedVehicle->getLeftTrack()->getWheels() )
  agxUtil::setBodyMaterial( wheel->getRigidBody(), wheelMaterial );
```

Note that we're separating ground, wheel and track materials. Normally it's enough to use the default friction model for the contacts between the track nodes and the wheels, but if the wheels doesn't respond enough to the friction, a simplified direct track friction model can be used:

```
#include <agxVehicle/TrackFrictionModels.h>

// Contact material between track nodes and the wheels.
auto trackWheelContactMaterial =
  simulation->getMaterialManager()->getOrCreateContactMaterial( trackMaterial, wheelMaterial );

// Slightly simplified friction model between the wheels and the track
// where we estimate the normal force constant @ ten times the mass of
// the chassis. Solved direct.
auto trackFmConstantNormalForce = 10.0 * trackedVehicle->getChassis()->getMassProperties()->getMass();

auto trackWheelFrictionModel =
  new agxVehicle::TrackBoxFrictionModel( agx::FrictionModel::DIRECT,
                                         trackFmConstantNormalForce );
trackWheelContactMaterial->setFrictionModel( trackWheelFrictionModel );

// Let the friction coefficient be high and "ease up" the friction constraints
// by using high surface viscosity. Friction properties doesn't affect the
// sprockets or idlers since the nodes will merge to them, simulating infinite
// stick friction against those geared wheel types.
trackWheelContactMaterial->setFrictionCoefficient( trackWheelFrictionCoefficient );
trackWheelContactMaterial->setSurfaceViscosity( trackWheelSurfaceViscosity );
trackWheelContactMaterial->setRestitution( 0.0 );
```

The contact material between the track nodes and ground is similar but here we include primary and secondary parameters:

```
// Contact material between track nodes and the ground.
auto trackGroundContactMaterial =
  simulation->getMaterialManager()->getOrCreateContactMaterial( trackMaterial, groundMaterial );

// Track-ground friction: use primary (along track) and secondary (transverse) parameters.
// This friction model can optionally scale the maximum friction force with contact depth.
auto trackGroundFrictionModel =
  new agxVehicle::TrackBoxFrictionModel( agx::FrictionModel::DIRECT,
                                         trackFmConstantNormalForce,
                                         true /* scaleWithDepth */ );
trackGroundContactMaterial->setFrictionModel( trackGroundFrictionModel );
```

```
trackGroundContactMaterial->setFrictionCoefficient( trackGroundFrictionCoefficients.x(),
                                                    agx::ContactMaterial::PRIMARY_DIRECTION );
trackGroundContactMaterial->setFrictionCoefficient( trackGroundFrictionCoefficients.y(),
                                                    agx::ContactMaterial::SECONDARY_DIRECTION );

trackGroundContactMaterial->setSurfaceViscosity( trackGroundSurfaceViscosity.x(),
                                                 agx::ContactMaterial::PRIMARY_DIRECTION );
trackGroundContactMaterial->setSurfaceViscosity( trackGroundSurfaceViscosity.y(),
                                                 agx::ContactMaterial::SECONDARY_DIRECTION );

trackGroundContactMaterial->setRestitution( 0.0 );
```

22.1.5.1.6. Merging segments

For performance reasons, it's possible to enable merging of nodes to segments along the track. The result can be seen in Fig. 22.2 where each segment has a different color. When the nodes are merged together several bodies and constraints aren't added to the solver (reducing the size of the matrix) and it's possible to perform contact reduction over the whole segment.

To enable merge (disabled by default):

```
auto mergeProperties = track->getInternalMergeProperties();
// Default: false
mergeProperties->setEnableMerge( true );
```

By default (when enabled), three nodes will merge together to a segment. To change the number of nodes to merge simply:

```
// Setting the number of nodes per segment to 6 (default 3).
mergeProperties->setNumNodesPerMergeSegment( 6u );
```

Contact reduction is by default *minimal* which means some contacts will be removed. The other options are *none* for no contact reduction, *moderate* for more contacts to be neglected and *aggressive* for the most intrusive reduction:

```
// Default: agxVehicle::TrackInternalMergeProperties::MINIMAL (bin resolution 3)
mergeProperties->setContactReduction( agxVehicle::TrackInternalMergeProperties::MODERATE );
```

The nodes must be aligned to be merged to avoid curvatures to build along the track. The nodes are constrained together with hinges and by default (when merge is enabled) the lock in the hinge is used to force the nodes to their straight/resting position. It's possible to toggle this feature and set compliance and damping of the lock:

```
// Default: true
mergeProperties->setEnableLockToReachMergeCondition( true );
// Default: 1.0E-11 (very strong)
mergeProperties->setLockToReachMergeConditionCompliance( 1.0E-6 );
// Default: 3.0 / 60.0
mergeProperties->setLockToReachMergeConditionDamping( 6.0 * timeStep );
// Maximum angle > 0 (in radians) when the nodes may merge.
// Default: 1.0E-5
mergeProperties->setMaxAngleMergeCondition( 6.0E-5 );
```

22.1.5.1.7. Track properties

All parameters and features implemented in the tracks model are exposed in the `agxVehicle::TrackProperties` object. All track instances have an instance of this object, and it is possible for several track instances to share a single instance of this object, e.g.:

```
auto globalProperties = track1->getProperties();
track2->setProperties( globalProperties );
track3->setProperties( globalProperties );

agxVehicle::TrackPropertiesRef otherProperties = new agxVehicle::TrackProperties();
track4->setProperties( otherProperties );
track5->setProperties( otherProperties );
```

It is also possible to load predefined property sets from the Material Library:

```
if ( properties->loadLibraryProperties( "steel" ) )
  std::cout << "Loaded track properties from library!" << std::endl;
```

You can query available library property sets using:

```
agx::StringVector available = agxVehicle::TrackProperties::getAvailableLibraryProperties();
```

There are many parameters in the track properties, some more important than others. This table, under “Property”, lists the accessor and mutator names – just add *get* or *set* and remove any spaces. E.g., **Bending Stiffness** is accessed as `properties->setBendingStiffness( stiffness, axis, attenuation )` and `auto k = properties->getBendingStiffness( axis )`:

Property	Description	Default value
Bending Stiffness	Resistance to bending deformation per unit length. Typical value is EI_a where E is Young’s modulus and I_a is the second moment of area. Requires specifying the axis (`Axis::LATERAL` or `Axis::VERTICAL`).	50 around lateral axis and 1.0e10 around vertical axis
Tensile Stiffness	Resistance to stretching/compression in the longitudinal direction per unit length. Higher values give a stiffer track. Default is 1e10 N/m.	1.0E10
Torsional Stiffness	Resistance to twisting around the track’s longitudinal axis. Typical value $(GJ) / L$.	1.0e10
Shear Stiffness	Resistance to shearing deformation per unit length, defined as GA / L. Requires specifying the axis (`Axis::LATERAL` or `Axis::VERTICAL`).	1.0e10 along both axis
Bending Attenuation	Attenuation factor applied in bending stiffness calculations, per axis.	2.0
Shear Attenuation	Attenuation factor applied in shear stiffness calculations, per axis.	2.0
Tensile Attenuation	Attenuation factor applied in tensile stiffness calculations.	2.0
Torsional Attenuation	Attenuation factor applied in torsional stiffness calculations.	2.0
Bending Friction Attenuation	Attenuation factor used for bending friction calculations.	2.0
Min Stabilizing Hinge Normal Force	Minimum value of the normal force (the hinge force along the track) used in “internal” friction calculations. I.e., when the track is compressed, this value is used with the friction coefficient as a minimum stabilizing compliance. If this value is negative there will be stabilization when the track is compressed.	100
Stabilizing Hinge Friction Parameter	Friction parameter of the internal friction in the node hinges. This parameter scales the normal force in the hinge.	1
Enable Hinge Range	Enable/disable hinge range between track nodes to define how the track may bend.	true
Hinge Range Range	The range (in radians) defining allowed relative angle between track nodes.	$-270 \leq \alpha \leq 20$ degrees.
Enable On Initialize Merge Nodes To Wheels	When the track has been initialized some nodes are in contact with the wheels. If this flag is true the interacting nodes will be merged to the wheel directly after initialize, if false the nodes will be merged during the first (or later) time step.	false
Enable On Initialize Transform Nodes To Wheels	True to position/transform the track nodes to the surface of the wheels after the track has been initialized. When false, the routing algorithm positions are used.	true
Transform Nodes To Wheels Overlap	When the nodes are transformed to the wheels, this is the final target overlap.	1.0E-3
Nodes To Wheels Merge Threshold	Threshold when to merge a node to a wheel. Given a reference direction in the track, this value is the projection of the deviation (from the reference direction) of the node direction onto the wheel radial direction vector. I.e., when the projection is negative the node can be considered “wrapped” on the wheel.	-0.1
Nodes To Wheels Split Threshold	Threshold when to split a node from a wheel. Given a reference direction in the track, this value is the projection of the deviation (from the reference direction) of the node direction onto the wheel radial direction vector. I.e., when the projection is negative the node can be considered “wrapped” on the wheel.	-0.05
Num Nodes Included In Average Direction	Average direction of non-merged nodes entering or exiting a wheel is used as reference direction to split of a merged node. This is the number of nodes to include into this average direction.	3

Note

The stabilization-related parameters (e.g., **Min Stabilizing Hinge Normal Force**, **Stabilizing Hinge Friction Parameter**) and node/wheel merge and hinge range settings are only relevant when using the **Full-Degree Tracks Model**. These parameters do not apply to the default **Low-Degree Tracks Model**.

The attenuation constants determines how much to attenuate an oscillation above the Nyquist frequency, default 2. This may be calculated from a (viscous) damping coefficient, see agxUtil::convertDampingCoefficientToSpookDamping and set attenuation to *spook damping time* / *simulation time step*.

22.1.5.1.7.1. Internal friction in the hinges

Internal friction in the hinges is used to stabilize the nodes in the track during high tension. The default friction coefficient (Stabilizing Hinge Friction Parameter) is relatively high and could work by default if the track is relatively large (sitting on a large tracked vehicle for example).

When the friction parameter is too low, the tracks will become unstable.

When the friction parameter is too high, the tracks will seem un-bendable, penetrating the wheels and/or the wheels aren't able to drive the track.

For example the tracked vehicle in *tutorial_trackedVehicle.cpp* (Fig. 22.1):

```
track->getProperties()->setStabilizingHingeFrictionParameter( 1.5 );
```

And to simulate a much smaller (thinner where the node mass is significantly smaller) track, in the same tutorial file, simulating a conveyor belt:

```
conveyor->getProperties()->setStabilizingHingeFrictionParameter( 4.0E-3 );
```

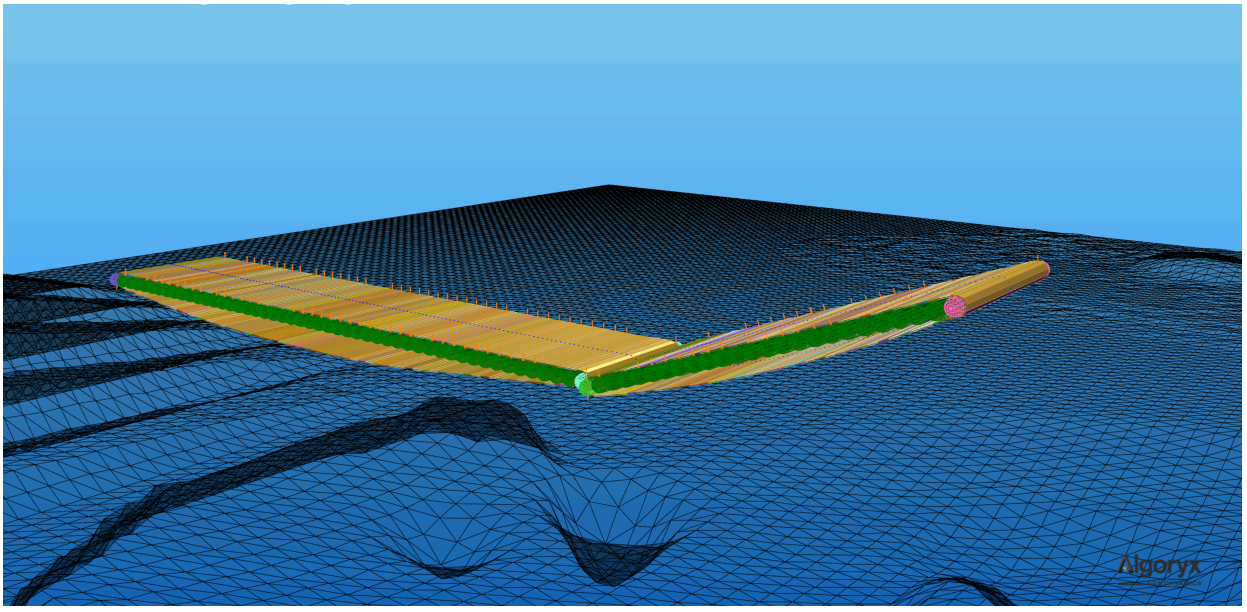


Fig. 22.10 Simulating conveyor belt using `agxVehicle::Track` where each roller is a wheel.

22.2. WheelJoint

A WheelJoint is a constraint to attach a wheel to a chassis or the world. The WheelJoint can be created in a few different ways. What they all have in common is that the wheel is the first body and the optional chassis is the second.

When using agx::Frames to construct a WheelJoint, some axes are more important than others. The Y-axis in the first body's frame specifies the wheel axle. The Z-axis in the second body's frame is used as suspension axis. When the steering is engaged, the rotation will be around this axis so it can also be referred to as steering axis.

The setup can also be done by using a WheelJointFrame which specifies in world coordinates, the wheel axis, the suspension/steering axis and the intersection point between those two. The two axes that are defined must be perpendicular.

```
agx::RigidBody* wheel = ... ;

// Position wheel at (1,0,0)
wheel->setPosition( agx::Vec3( 1.0, 0, 0 ) );

// We have the attachment between wheel axle and steering axle at (0.5, 0, 0)
agx::Vec3 worldAttach       = agx::Vec3( 0.5, 0, 0 );

// Wheel attachment point and wheel position are on the wheel axle
agx::Vec3 worldWheelAxle    = agx::Vec3::Y_AXIS();
```

```
agx::Vec3 worldSteeringAxle = agx::Vec3::Z_AXIS();

// Specify a frame
auto wheelJointFrame = WheelJointFrame( worldAttach, worldWheelAxle, worldSteeringAxle );

// Create the constraint. In this case, the wheel is attached to the world
agxVehicle::WheelJoint* wheeljoint = new agxVehicle::WheelJoint( wheelJointFrame, wheel, nullptr );

simulation->add( wheeljoint );

// The suspension is locked by default, release...
auto su_lock = wheeljoint->getLock1D( agxVehicle::WheelJoint::SUSPENSION );
su_lock->setEnable( false );

// .. and enable the range instead to show free movement.
auto su_range = wheeljoint->getRange1D( agxVehicle::WheelJoint::SUSPENSION );
su_range->setEnable( true );
su_range->setRange( -0.1, 0.1 );

// If we don't want rotation around the steering axis, we can lock it
// at its current angle
auto st_lock = wheeljoint->getLock1D( agxVehicle::WheelJoint::STEERING );
st_lock->setPosition( wheeljoint->getAngle( agxVehicle::WheelJoint::STEERING) );
st_lock->setEnable( true );
```

22.2.1. Connecting to a power-line

A WheelJoint can be powered by a power-line by attaching it to an Actuator. A RotationalActuator cannot be used by itself since a WheelJoint has multiple degrees of freedom, which means that we need to tell the Actuator which of them it should operate on. This is done by passing a WheelJointConstraintGeometry to the RotationalActuator. The WheelJointConstraintGeometry identifies one of the WheelJoint's free degree of freedom and we chose which one using the same technique as when getting a Lock1D or a Range1D from it.

Continuing the example above.

```
agxPowerLine::ConstraintGeometryRef wheelAttachAxis =
  new agxVehicle::WheelJointConstraintGeometry(wheelJoint, agxVehicle::WheelJoint::WHEEL);
agxPowerLine::RotationalActuatorRef actuator =
  new agxPowerLine::RotationalActuator(wheelJoint, wheelAttachAxis);
```

To control the suspension with a TranslationalActuator we would use `agxVehicle::WheelJoint::SUSPENSION` instead.

```
agxPowerLine::ConstraintGeometryRef suspensionAttachAxis =
  new agxVehicle::WheelJointConstraintGeometry(wheelJoint, agxVehicle::WheelJoint::SUSPENSION);
agxPowerLine::TranslationalActuatorRef actuator =
  new agxPowerLine::TranslationalActuator(wheelJoint, suspensionAttachAxis);
```

For more information on power-line components and how to connect them see AGX PowerLine. For a complete example see the `tutorial_wheel_joint.agxPy` Python tutorial.

22.3. Steering

When a car actually turns as shown in Fig. 22.11 a), the steering mechanism is designed to align the wheels simultaneously. The inner wheel will follow a circle with radius R, while the outer wheel follows a concentric circle with radius $R + W_t$. Note that W_t refers to the wheel track, W_b denotes the wheel base.

To minimize the slip and ensure the better turning performance, the front steerable wheels should hold the different steering angles of δ_{in} and δ_{out}. According to Ackermann principle, it yields

$$\delta_{out} = \arctan(\frac{1}{\cot(\delta_{in}) + W_t / W_b})$$

where δ_{in} refers to the steering angle of the inner wheel, δ_{out} is the ideal steering angle of the outer wheel corresponding to the prescribed δ_{in}.

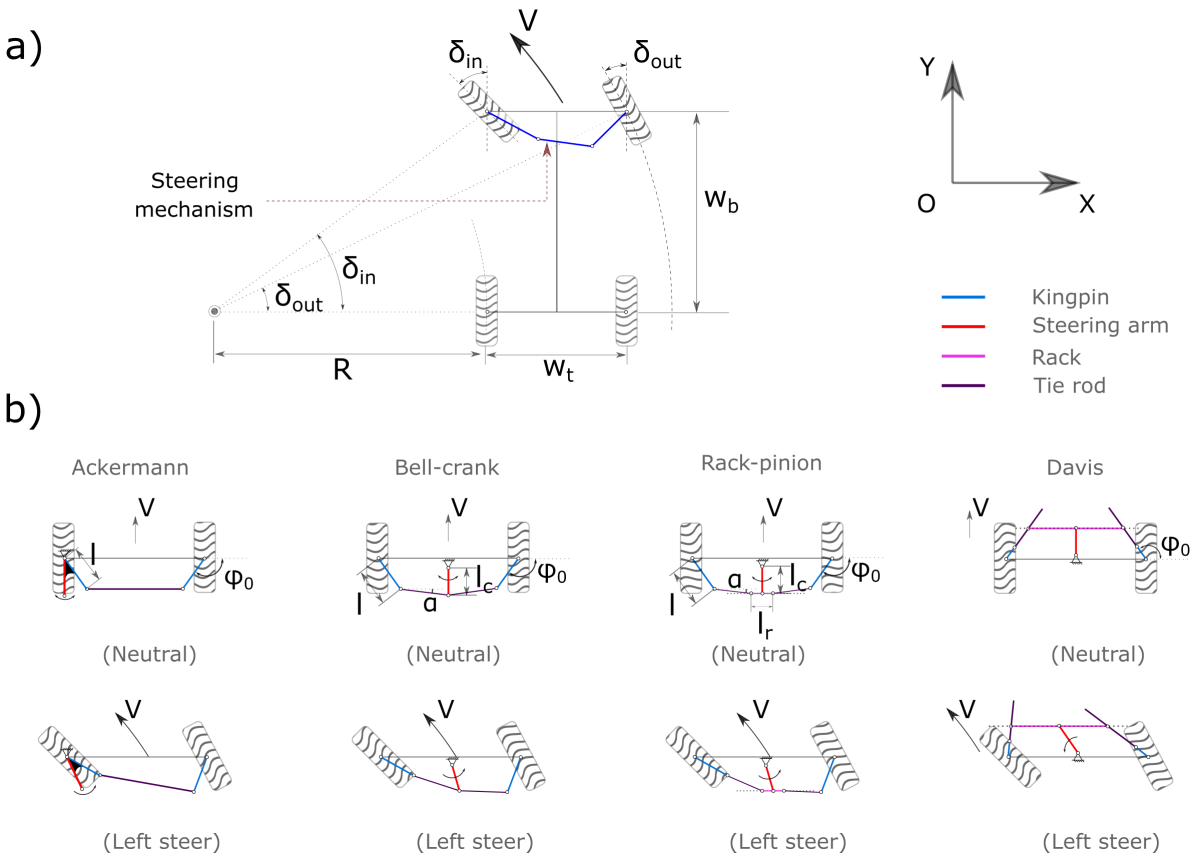


Fig. 22.11 Car steering schematic. a) Turning geometry; b) Four different steering mechanisms.

The generally used Ackermman steering mechanism (See Fig. 22.11 b) is a symmetric planar four-bar linkage. The steering arm, which rigidly connects but not limited to the left kingpin, will transmit the direct steering input of steering wheel to the two front steerable wheels via the trapezoid four-bar linkage. The Ackermann steering geometry is known to be simple and easy for calculation. But it is not always practically possible to implement. Alternatively, there are other steering mechanisms such as bell-crank (also called central-lever [7]), rack-pinion, and davis proposed.

For instance, in a bell-crank steering system, the steering arm rotates, pushing the shared ends of the tie rods to move left or right. This in turn will cause the wheel to turn due to the steering linkages. Bell-crank steering has been reported [7] that is common to wheeled tractors, ATVs, etc.

Regarding the Rack-pinion steering mechanism, the pinion is used to transmit the angular motion of the steering arm into the translational or linear motion of the rack. Rack-pinion steering mechanism is most commonly used to passenger cars [9].

Davis steering mechanism [10] is an exact steering mechanism but is seldom used in practice. It has two sliding pairs in-between the tie rod and the bar. The tie rods are only allowed to move in the direction of the its length. The main disadvantage of the Davis steering is the wear problem of the sliding pairs.

22.3.1. Geometrical parameters

As can be seen from Fig. 22.11 b), all the steering mechanisms share the common parameters of ϕ_0 and $gear$. The meaning of these parameters have been described in Table 22.1 as shown below. It can be noticed that the number of independent geometrical parameters varies in terms of the steering mechanisms. For example, there are only one independent parameter of ϕ_0 needed for Davis steering, and two parameters of ϕ_0 and l for Ackermann steering to define the steering geometry. The steering parameters as listed in Table 22.1 are default and optimal. They were selected from [7], [8], [11] and [12].

Table 22.1 Default steering parameters in AGX Dynamics

Parameter	Unit	Description	Ackermann	BellCrank	RackPinion	Davis
ϕ_0	rad	Angle between kingpin direction and X axis	$\frac{-115}{180}\pi$	$\frac{-108}{180}\pi$	$\frac{-108}{180}\pi$	$\frac{104}{180}\pi$
l	$\frac{1}{W_t}$	kingpin length	0.16	0.14	0.14	–
α	rad	Angle between tie rod and X axis	–	0.0	0.0	–
l_c	$\frac{1}{W_t}$	Steering arm length	–	1.0	1.0	–
l_r	$\frac{1}{W_t}$	Rack or bar length	–	–	0.25	–
gear	–	Steering wheel gear ratio	1.0	1.0	1.0	1.0
side	–	Side of steering arm positioned	0	–	–	–

In Table 22.1, "-" indicates that is not applicable.

The users can build their own desired steering geometry using the customized steering geometry parameters. For instance,

```
auto steerParams = new agxVehicle::SteeringParameters();
// The different values of parameters from the default are prescribed.
steerParams->phi0 = agx::degreesToRadians(-115);
steerParams->l = 0.16;

auto ackermann = new agxVehicle::Ackermann( leftWheel, rightWheel, steerParams);
sim->add( ackermann );
```

Once the steering mechanism is active, the range of the maximum allowed turning angle according to the geometrical configuration of the steering mechanism will be enabled automatically. This range should not be modified or disabled by the user. However, the user can access the maximum allowed steering angle using the getMaximumSteeringAngle() method.

22.3.2. Building steering

All the steering mechanisms implemented are enforced as three body constraints between each wheel body and the chassis.

The steering mechanisms can be created by simply replacing the “BellCrank” as shown below with the others such as “RackPinion”, “Davis” and “Ackermann”.

```
SteeringRef steeringMechanism = new agxVehicle::BellCrank( leftFrontWheel, rightFrontWheel );
sim->add( steeringMechanism );
```

If no SteeringParameters are assigned as an input after the “rightWheel”, the default steering parameters as listed in Table 22.1 will be used.

22.3.3. Steering manipulation

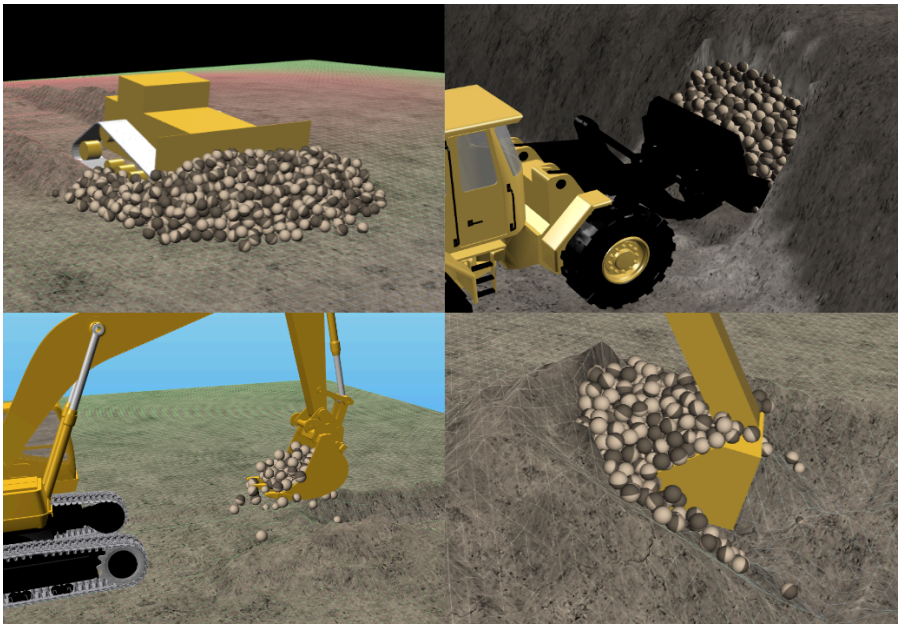
The steering can be manipulated using the setSteeringAngle() method.

```
class steeringListener : public agxSDK::StepEventListener
{
  public:
    steeringListener(agxVehicle::Ackerman* ackermann) :
    {
    }
    void pre(const agx::TimeStamp& t)
    {
      if (t < 1.0) {
        ackermann->setSteeringAngle(t);
      }
      else if (t <= 3.0) {
        ackermann->setSteeringAngle(1 - (t-1) / 2);
      }
      else
      {
        ackermann->setSteeringAngle(0.0);
      }
    }
```

Detailed C++ and python examples with the aforementioned steering mechanisms are available in `tutorials/agxVehicle/tutorial_carSteering.cpp` and `python/tutorials/tutorial_carSteering.agxPy`.

23. AGX Terrain

`agxTerrain` is a library of classes that is used to model a wide variety of earthmoving scenarios such as bulldozing, wheel loading, excavation, trenching and clamshell bucket excavation. It also supports moving the excavated soil on dump vessels. The deformable `agxTerrain::Terrain` model is based on a regular **3D grid** data structure and an overlapping surface height field. The terrain can be deformed by interacting `agxTerrain::Shovel` objects to perform different operations, such as **digging**, **push/pull** and **grading**.



The underlying terrain data model consists of a **3D grid** of cells, containing terrain data such as mass, compaction and soil type information. The terrain surface is represented as a 2D height field that is updated when the underlying grid structure is changed, mainly through excavation. The terrain model z-coordinate has a lower bound given by the specified *maximum depth*. Usually, the maximum depth is uniformly enforced on the entire terrain model but it is possible to configure a terrain with variable minimum heights.

Mass is moved in the terrain via interacting `agxTerrain::Shovels` that converts solid cells to **dynamic mass** in the form of soil particles with 6-degrees of freedom. The dynamic soil is pushed via the shovel through kinematic coupling. A **mass aggregate** body is constructed from the dynamics mass in **failure zones** predicted from **soil mechanics theory** using terrain material properties. These aggregates create a **resistance** to the shovel due to their inertia, acting through contacts in the soil failure plane. The dynamic soil is merged back into the terrain when approaching a steady state on the terrain surface.

Python tutorials and examples can be found in the `data/tutorials/agxTerrain` folder of the AGX Dynamics installation. Please consult the FAQ section to get answers to commonly asked questions.

This User Manual section focus on the practical application of the terrain module. We refer to the academic paper [1] for a more in-depth explanation of the underlying mathematical and physical models.

`agxTerrain::Terrain` height changed data can also be recorded to a journal.

23.1. Workflow

The main classes that the user interacts with when working with agxTerrain is as follows:

`agxTerrain::Terrain`	The deformable terrain object.
`agxTerrain::Shovel`	Shovel object that converts solid mass in an `agxTerrain::Terrain` object to Dynamic mass via an active zone.
`agxTerrain::TerrainMaterial`	Object that holds material properties used to characterize soil behavior such as compaction, soil particle dynamics and bulk properties.
`agxTerrain::TerrainProperties`	Object that holds generic terrain properties such as disable/enable for different features and locking terrain borders.
`agxTerrain::SoilSimulationInterface`	Interface for controlling the soil simulation governing dynamic soil behavior.
`agxTerrain::TerrainGridControl`	Controller for accessing and controlling the underlying terrain data on grid element level.

The overall work flow for setting up a terrain model is as follows:

1. Create a agxTerrain::Terrain object from resolution data or an existing height field.

2. Create and configure agxTerrain::Shovel objects by specifying a shovel body, cutting edge, top edge and cutting direction.
3. Configure the Terrain Material Configuration. For example, change the default `agxTerrain::TerrainMaterial` and specify other soil type domains.
4. Setup contact materials between the terrain and external objects, especially the shovel-terrain contact materials.
5. Calibrate excavation resistance by configuring shovel-terrain contact material and soil properties.
6. (Optional) Modify the terrain properties via the `agxTerrain::TerrainProperties` class.
7. (Optional) Modify the terrain grid structure using the `agxTerrain::TerrainGridControl` class.
8. (Optional) Configure the dynamic soil.

23.2. Terrain

`agxTerrain::Terrain` is the main class of the agxTerrain library and represents the actual deformable terrain object. It supports the following features:

- Soil excavation using `agxTerrain::Shovel` objects.
- Soil compaction from either shovels or other external objects, such as for example wheels, tracks or other rigid bodies.
- Angle of repose effects.
- Conversion between solid and dynamic mass during excavation.

The underlying terrain data model consists of a regular 3D grid of cells. Terrain data such as mass, compaction and soil type information is stored discretely in each cell. The terrain surface is represented as a 2D height field that is updated when the underlying grid structure is changed, mainly through excavation. Excavation is allowed in the terrain down unto the maximum depth of the model, specified upon creation. The specified maximum depth effectively sets a uniform lowest allowable height of the surface in the terrain model. The model can also be configured to use variable minimum heights at each 2D-coordinate.

Some terrain data, such as terrain height, is parametrized by specifying only the 2D x and y coordinate, called *terrain index*, which references a single discrete surface grid point on the terrain. The origin `(x:0, y:0)` is in the lower leftmost gridpoint and the maximum coordinate in the upper rightmost gridpoint `( x: resolutionX-1, y: resolutionY-1 )`.

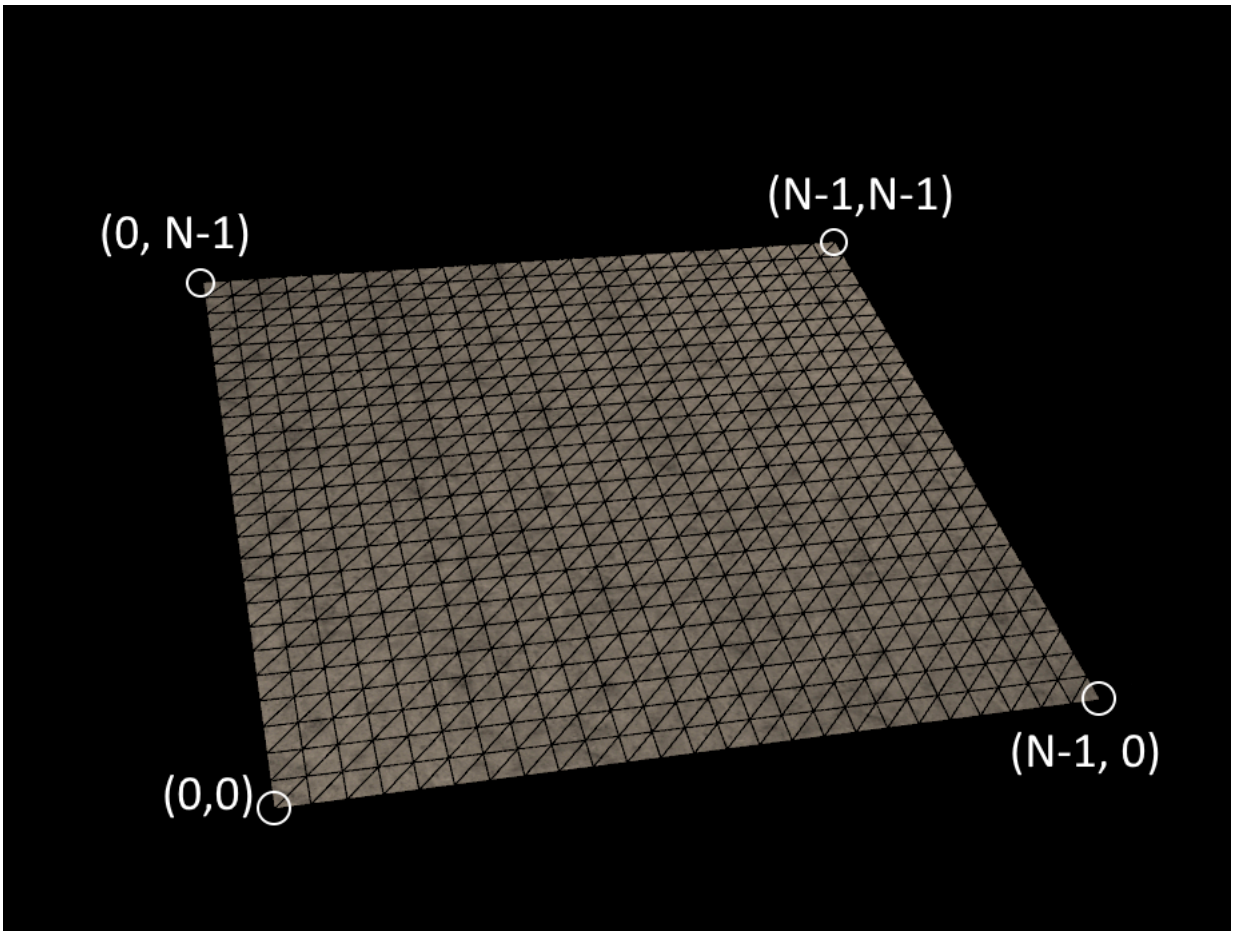


Fig. 23.1 Figure showing the index mapping of the `agxTerrain::Terrain` gridpoints. The origin `(x:0, y:0)` is in the lower leftmost gridpoint and the maximum coordinate in the upper rightmost gridpoint `( x: resolutionX-1, y: resolutionY-1 )`

Other terrain data, such as compaction, might need the full 3D x-, y-, and z-coordinate, called *voxel index*, to be accessed.

Note

Terrain indices and *voxel indices* use the same x- and y-coordinates. However, x- and y-coordinates are greater than or equal to 0, whereas the *voxel index* z-coordinate can be negative.

23.2.1. Terrain Setup

An `agxTerrain::Terrain` object can either be created by specifying the 2D size and resolution data or by using an existing height field to define the surface level of the model. The data cells in the terrain is symmetric by design which means that the user is constrained to setting resolution and cell size to ensure that we have a valid data structure.

Note

It is also possible to create a terrain using an arbitrary geometry via the Terrain Pager.

The following code can be used to create a terrain by specifying the 2D size and resolution data:

```
// Construct a Terrain with 51-by-51 elements with element size 0.2 m
// and a maximum depth of 2.5 m and it to the simulation
size_t resolutionX = 51;
size_t resolutionY = 51;
agx::Real elementSize = 0.2;
agx::Real maximumDepth = 2.5;
agxTerrainRef terrain = new agxTerrain::Terrain(resolutionX, resolutionY, elementSize, maximumDepth);
sim->add(terrain);
```

The `resolution` parameters defines the number of discretized grid cells in the X and Y axis. The `elementSize` sets the uniform size of the 3D grid cells in the terrain structure. The total length of the terrain in the x- and y-axis will thus become `(resolutionX-1) * elementSize` and `(resolutionY-1) * elementSize` respectively.

The `maximumDepth` parameter, defines the maximum depth below the `z = 0`-height that the terrain can have in its local z-direction. This is equivalent to the `lowestAllowedHeight` in the terrain. Below this depth the terrain contains no solid mass which can be converted to dynamic mass.

Note

Sometimes the user may not want a uniform depth in the terrain, such as when trying to adjust the terrain to fit on top of a vehicle's dump body which may not be entirely flat. To accommodate for this, it is possible to set a minimum height in each grid cell. For the purpose of dump bodies, there is a utility function `Terrain::createTerrainBedFromGeometries` which fits a terrain object on top of a collection of geometries. See the tutorial `data\python\tutorials\agxTerrain\tutorial_dump_vessel.agxPy` for a demonstration. It is also possible to manually set the heights, see the tutorial `data\python\tutorials\agxTerrain\tutorial_minimum_heights.py` for how to modify the minimum heights in a larger terrain.

Creating a terrain object from an existing height field is done by using the following method:

```
agxCollide::HeightFieldRef heightField = ... // Create the height field
// Supply height field object and a maximum depth in meters for the terrain
agxTerrain::TerrainRef terrain = agxTerrain::Terrain::createFromHeightField(heightField, 0.5);
```

Warning

The specified height field must have symmetric scale. I.e. the distance between vertices in x-direction must be the same as in y-direction. An agx::LOGGER_WARNING will tell you if you have an asymmetrical height field scale and a nullptr will be return from the function.

Below follows some common operations on the terrain to access data:

```
size_t res = 51;
agxTerrainRef terrain = new agxTerrain::Terrain(res, res, 0.2, 2.5);
sim->add(terrain);

// Returns the local height of the terrain at the lower left corner
terrain->getHeight(0,0);

// Returns the surface world position of the terrain at upper right corner index 50,50
terrain->getSurfacePositionWorld(res-1,res-1);
```

23.3. Shovel

Note

Only agxTerrain::Shovels can perform excavation and digging operations in the Terrain. Note that terrain compaction can be performed by all bodies.

`agxTerrain::Shovel` objects are used in order to deform the terrain. A shovel is an object that can represent different types of tools used in earthmoving operations: excavator bucket, wheel loader bucket and a bulldozer blade. The shovel deforms the terrain during digging and deformation operations that creates failure/active zones that converts solid mass in the terrain to dynamic mass. The shovel experiences feedback forces from penetration resistance and soil aggregates that are generated from the masses in the active zones. The shovel can also compress the soil when no excavation is in progress.

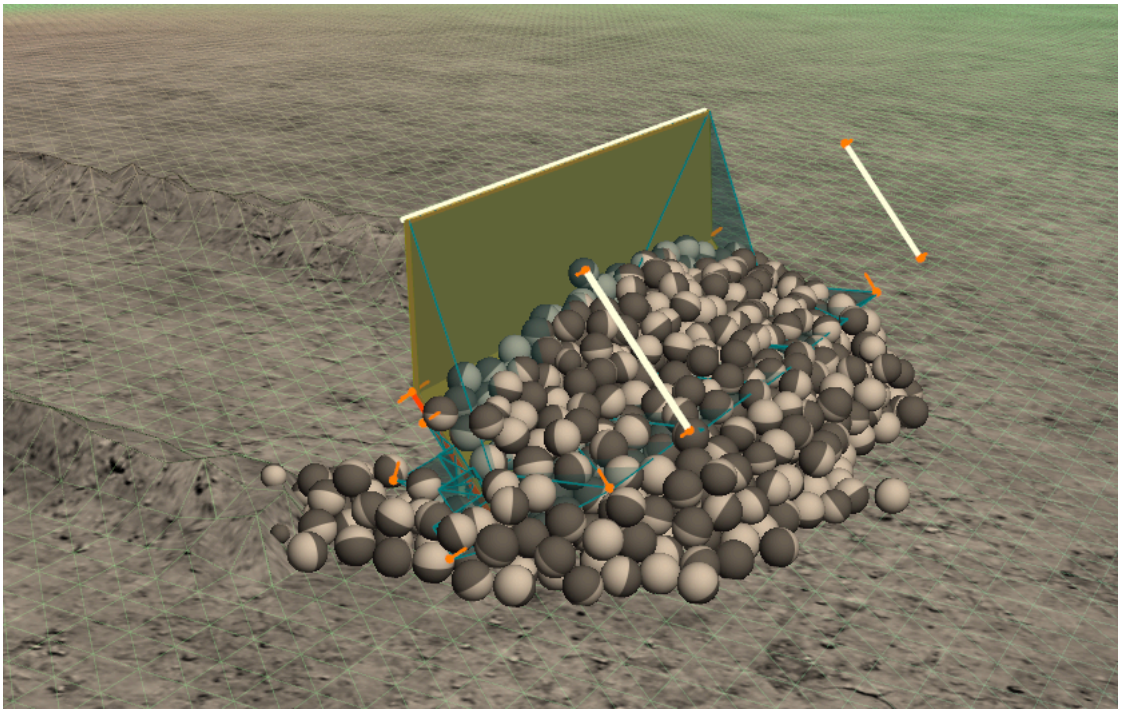


Fig. 23.2 Figure of a simple two-body *agxTerrain::Shovel* digging in a *agxTerrain::Terrain* object, converting solid mass to dynamic soil particle mass in the active zone (light blue wedges in front, to the side and in the back of the shovel).

A shovel can deform the terrain via the following applications:

Digging	This is done by driving the shovel in the cutting direction into the terrain, converting solid mass to dynamic mass. See Fig. 23.3 left.
Push/Pull	This is done via doing *push/pull* operations with the sides or backside of the shovel. See Fig. 23.3 middle.
Grading	This is moving the soil using the backside of the shovel. See Fig. 23.3 right.

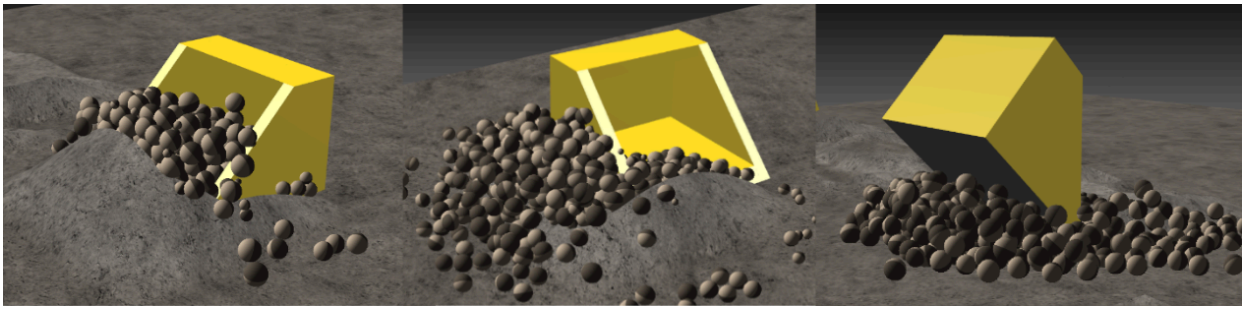


Fig. 23.3 Figure showing the different excavation operations of a shovel.

agxTerrain also support special composite shovels such as Clamshell buckets.

23.3.1. Shovel Setup

A shovel is created from a rigid body model of the shovel using a different set of vectors. A `top edge`, `cutting edge` and optional `tooth direction` and `tooth length` which **needs to be specified by the user in local body coordinates** and should coincide with the geometry. Both simple box geometries and more complex mesh shapes can be used.

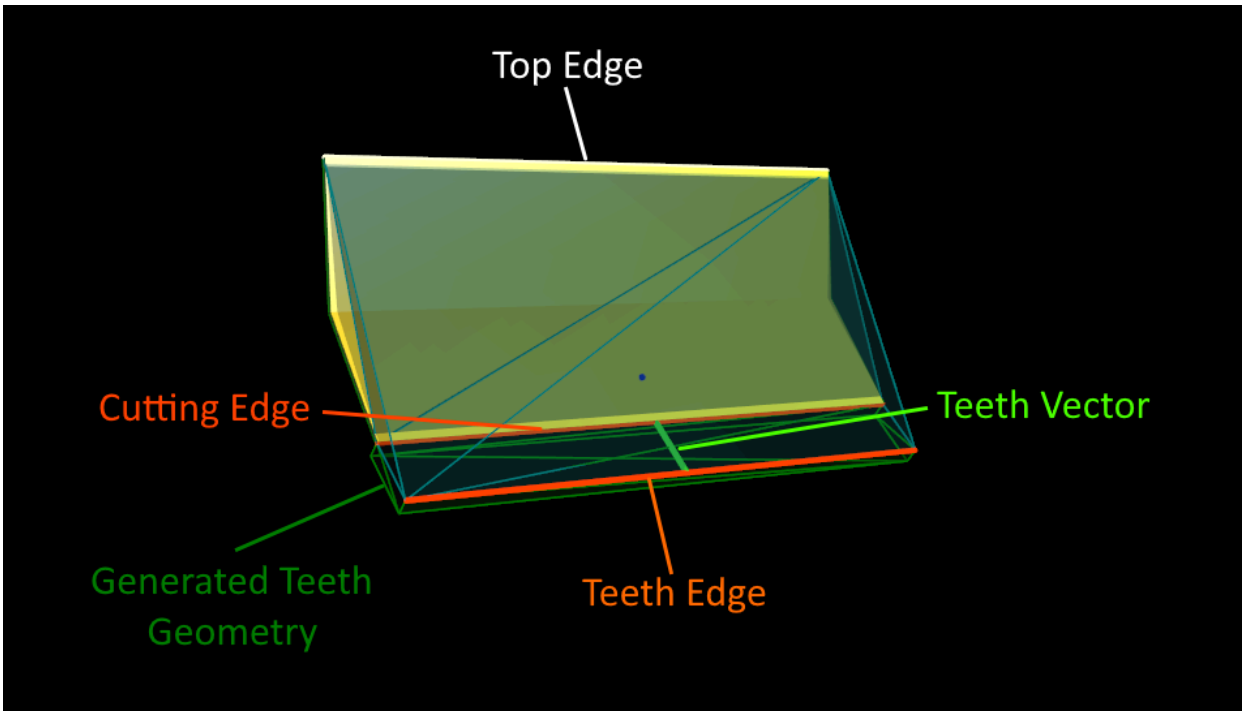


Fig. 23.4 Figure of the basic components needed in order to create a shovel. A Rigid body representing the shovel object. A `top edge` is placed on the upper confines of the expected digging area, a `cutting edge` is placed on the lowest confines of the shovel geometry and a `tooth direction` is placed along the expected tooth direction of the shovel. A `teeth edge` is generated from the specified tooth direction and length where the penetration resistance will start acting during digging. By the default, soil excavation will start at the cutting edge. It is also possible to enable excavation to start at the teeth edge. If excavation at the teeth edge is enabled, a `tooth geometry` is by default generated between the cutting edge and the teeth edge to represent the collision geometry of the teeth which will interact with generated dynamic mass during excavation.

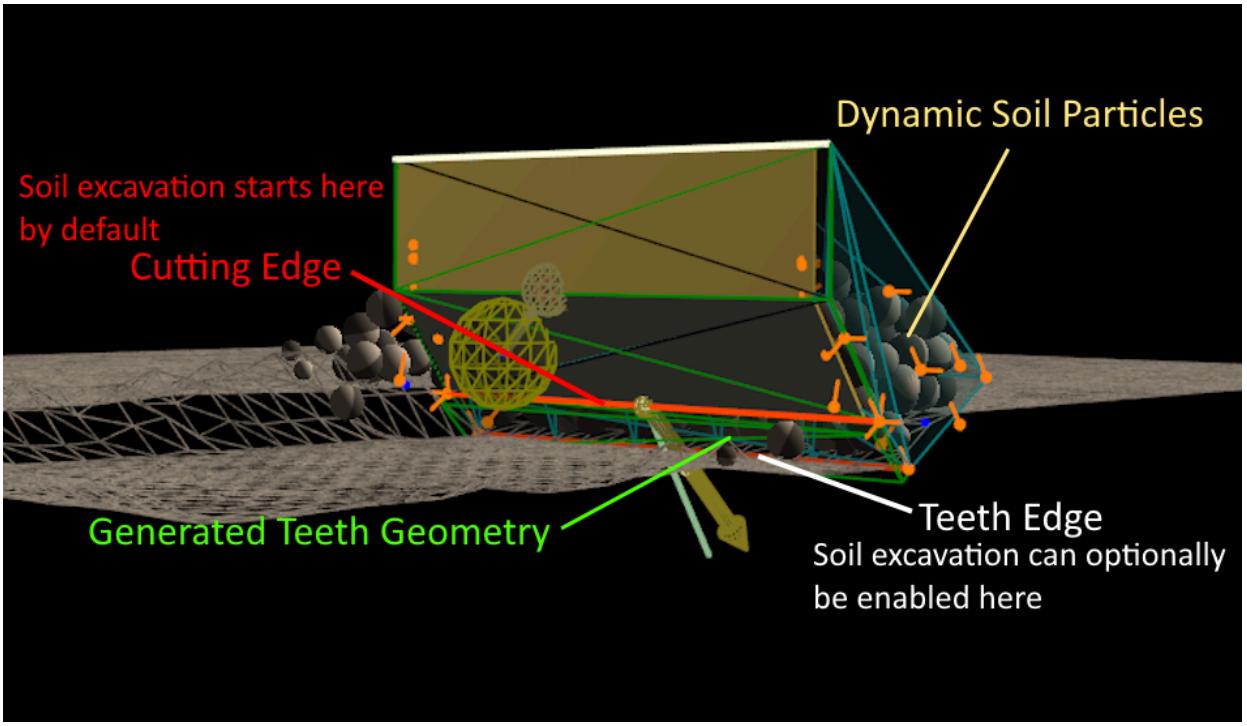


Fig. 23.5 Figure showing soil excavation of a bulldozer blade shovel in a terrain. By default, soil excavation starts at the `cutting edge` with failure zones being created in the terrain from the shovel according to soil mechanics theory. It is also possible to enable soil excavation starting at the teeth edge. During soil failure the solid terrain inside the failure volume is converted to dynamic soil. The dynamic soil is converted into soil particles which collides with the generated teeth volume.

The placement of the vectors should be as described below and visualized in Fig. 23.6:

- The `top edge` should ideally be placed at the upper confines of the cross-sectional digging area.
- The `cutting edge` should align with the lowest edge of the shovel that is expected to cut the soil, typically at the base of the shovel teeth if present. By default, the transition from solid to dynamic soil begins at this point.

- The `tooth direction` should be placed along the direction of the shovel tooth, if any exists.
- The `teeth edge` is *generated* from the `tooth direction` and `tooth length`. It is possible to enable the conversion of solid to dynamic soil during excavation to start here.

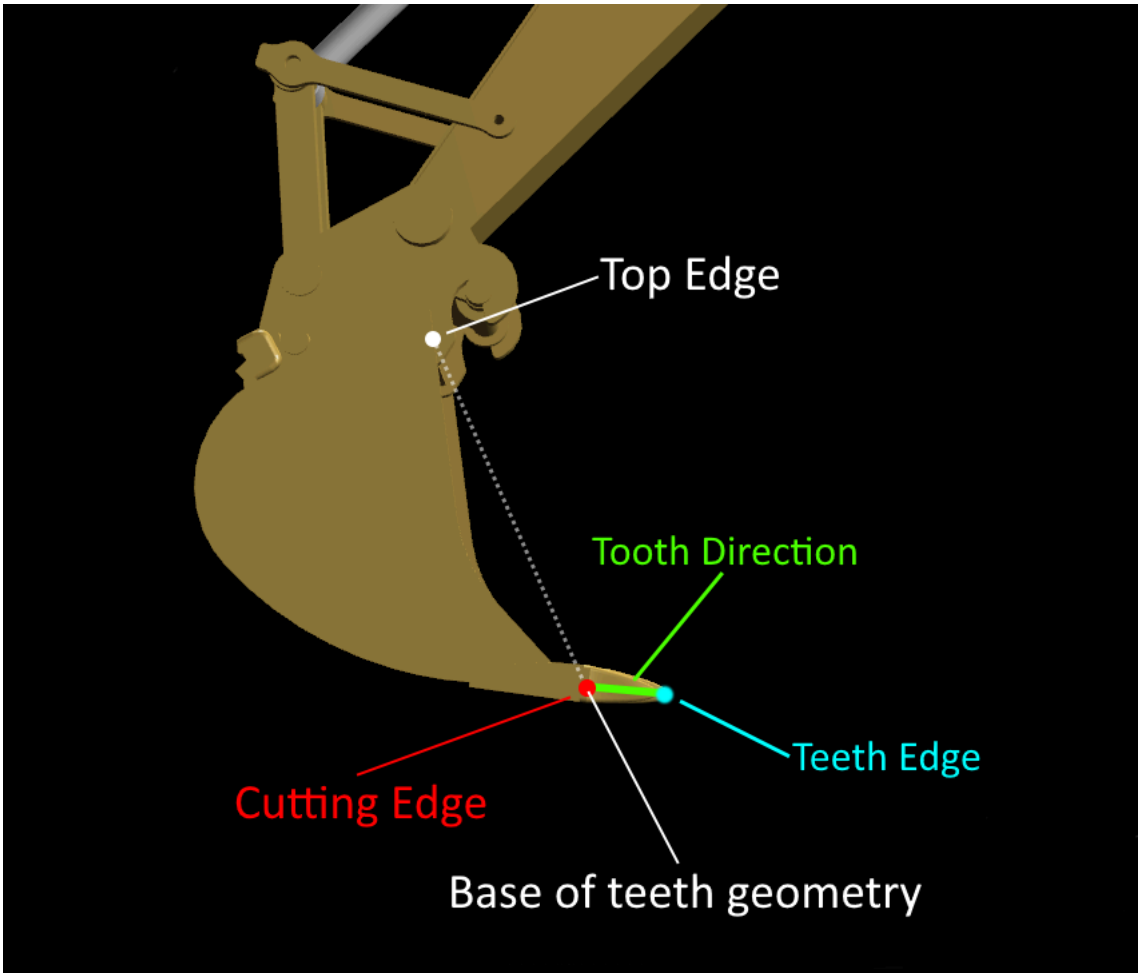


Fig. 23.6 Figure showing the shovel vector placement for an excavator bucket.

The `teeth direction` should be placed along the direction of the teeth geometry of shovel. `The cutting edge` should be placed at the base of the shovel teeth geometry.

A shovel is created via the following constructor and then added to the simulation:

```
agxTerrain::ShovelRef shovel = agxTerrain::Shovel( agx::RigidBody* shovelBody,
                                                   const agx::Line& topEdge,
                                                   const agx::Line& cuttingEdge,
                                                   const agx::Vec3& toothDirection,
                                                   agx::Real toothLength );
simulation->add(shovel)
```

See shovel settings for various options that is used to modify shovel behaviour.

To enable soil excavation to start at the teeth edge:

```
// Enable excavation to start at teeth edge
shovel->getSettings()->setEnableExcavationAtTeethEdge(true);
```

After a shovel has been added to a simulation, it will interact with any other terrains added to the simulation. This also includes any terrains used as movable terrain. It is possible to disable shovel<->terrain interactions between specific pairs of shovels or terrains. Example:

```
// Disable terrain interactions - excavation, bulldozing, etc - with a specific shovel
bool enable = false;
shovel->setEnableForTerrain(terrain, enable);
```

```
// Also disable geometry contacts, ie collisions between the shovel and the terrain
shovel->setEnableShovelTerrainGeometryContacts(terrain, enable);
```

You can get the current soil mass and bulk volume inside the shovel as well as the the total amount moved by the shovel in a terrain:

```
// Get the soil mass and bulk volume inside the shovel
agx::Real massInside = shovel->getInnerSoilMass();
agx::Real innerBulkVolume = shovel->getInnerSoilBulkVolume();

// Get the total soil mass and bulk volume moved by the shovel
// This includes soil inside the shovel as well as the volume
// and mass moved outside in the terrain.
agx::Real totalSoilMass = shovel->getTotalSoilMass();
agx::Real totalSoilBulkVolume = shovel->getTotalSoilBulkVolume();
```

23.4. Movable Terrain

It is possible to use a terrain as part of a dynamic, movable rigid body. This is especially useful if the user wishes to model dump trucks or other vessels used for moving excavated soil between locations. To attach a terrain to a rigid body, use the following code snippet:

```
void attachTerrainToRigidBody(agxTerrain::Terrain* terrain, agx::RigidBody* rb)
{
  rb->add(terrain->getGeometry());
}
```

To make it easier for the user to fit terrain objects to rigid bodies, there is a utility function `Terrain::createTerrainBedFromGeometries` which works on a list of geometries. If your rigid body simply is a platform, the following code snippet may suffice:

```
// Create a terrain bed from the geometries in a rigid body with resolution 16 along the x-axis. The resolution along the
y-axis will be approximated from the dimension of the given geometries.
auto bedTerrain = createTerrainBedFromGeometries(16, rb->getGeometries());
```

It is also possible to create a dump vessel which during tipping turns the offloaded soil on the bed into dynamic mass, which then can be redeposited into the ground. This is also showcased in `data\python\tutorials\agxTerrain\tutorial_dump_vessel.agxPy`, where a `StepEventListener` is used to turn solid mass into dynamic mass when the dump bed has been tipped over a certain angle. A visualization of a terrain used as a dump bed and the tipping can be seen below in Fig. 23.7:

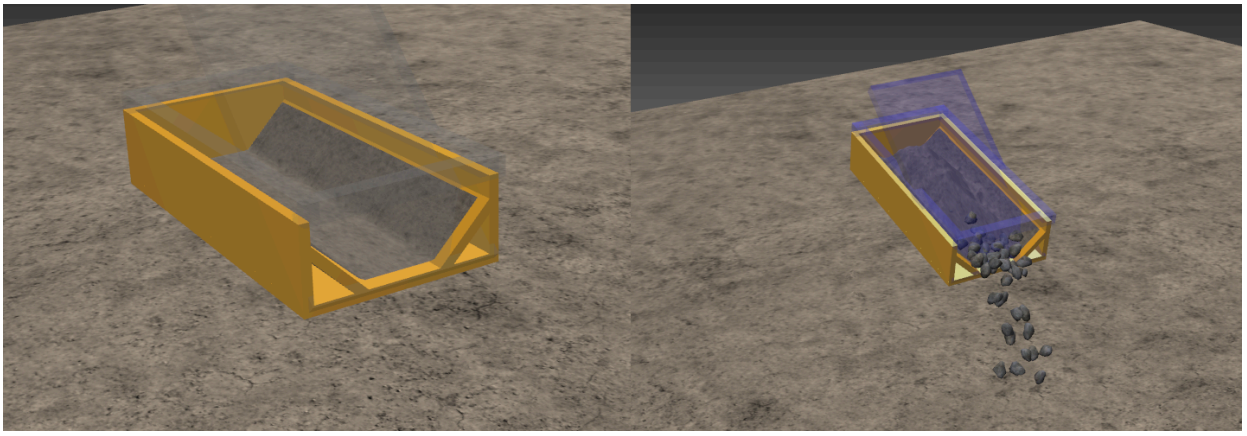


Fig. 23.7 Figure showing the example dump bed modeled in `tutorial_dump_vessel.agxPy`. To the left, an empty offload terrain attached to the rigid body. To the right, a partially filled dump bed during tipping.

Note

Modeling tipping, where the solid mass in the dump bed is turned into dynamic mass, has some caveats. For example, the user must make sure the same terrain material is used both in the bed and in the ground, otherwise mass conservation is not assured. Likewise, soil repose effects are not properly represented in a tilted *agxTerrain::Terrain* during tipping, as the objects local z-axis and the gravity direction does not align during that time.

23.5. Terrain Material Configuration

The agxTerrain::TerrainMaterial objects govern internal terrain behaviour and contains dynamic soil interaction parameters, internal soil contact parameters, angle of repose, compaction, etc, which may vary depending on soil type.

The `agxTerrain::Terrain` holds a default `agxTerrain::TerrainMaterial`, which applies everywhere unless other soil types are added. There are `get/set` methods for interacting with the default `agxTerrain::TerrainMaterial`:

```
// Get the currently assigned default terrain material
agxTerrain::TerrainMaterialRef defaultTerrainMaterial = terrain->getTerrainMaterial();

// Create a new TerrainMaterial with default parameters, and set it as the default terrain material
agxTerrain::TerrainMaterialRef terrainMaterial = new agxTerrain::TerrainMaterial()
terrain.setTerrainMaterial(terrainMaterial)
```

It is also possible to add several other `agxTerrain::TerrainMaterial` to the terrain, representing different soil types in different regions. To do this, use the `addTerrainMaterial` method:

```
agxTerrain::Terrain::addTerrainMaterial(agxTerrain::TerrainMaterial* terrainMaterial);
agxTerrain::Terrain::addTerrainMaterial(agxTerrain::TerrainMaterial* terrainMaterial, agxCollide::Geometry* geometry);
```

The first method only adds the terrain material to the terrain, without assigning it to any area. The second method adds the terrain material to the overlap between the terrain and the geometry. This is the part of the terrain below the height field and the geometry that intersects.

If a terrain material has been added, it can be removed entirely, together with any domain assignment, with the `removeTerrainMaterial` method:

```
agxTerrain::Terrain::removeTerrainMaterial(agxTerrain::TerrainMaterial* terrainMaterial);
```

If the user wants to keep the assigned domain but simply change the material, there is a method for that as well:

```
bool agxTerrain::Terrain::exchangeTerrainMaterial(TerrainMaterial* oldTerrainMaterial, TerrainMaterial*
newTerrainMaterial);
```

Note

The default terrain material can not be removed, but it can be exchanged.

agxTerrain::TerrainMaterial properties can be configured according to the desired effects. There is also pre-calibrated material presets for the `agxTerrain::TerrainMaterial` objects to choose from, if you do not wish to specify their properties manually. The user can also specify their own presets. All preset materials can be found under the `data/MaterialLibrary/TerrainMaterials` in the AGX Dynamics installation folder.

Note

We recommended that new users start with a material preset as a starting point before manually configuring their own TerrainMaterial.

Refer to the agxTerrain::TerrainMaterial section to get an overview of all available soil parameters.

23.6. Setup Contact Materials

There are 5 types of interactions that the terrain can have:

`Shovel-Terrain`	This is the interaction between the shovel and the terrain that happens via aggregate formation during excavation and deformation. The contact parameters here are derived from the contact materials specified between the shovel and the terrain. See calibration of excavation resistance.
`Object-Terrain`	This is the interaction between the terrain and external objects such as wheels and tracks.
`Particle-Object`	The interaction between the dynamic soil and the *external* object, such as shovels and tracks.
`Particle-Particle`	The interactions between the dynamic soil particles.
`Particle-Terrain`	The interactions between the dynamic soil particles and the terrain surface.

Each of these interactions depend on the defined contact materials between the relevant materials.

23.6.1. Internal Materials

In the case of interactions with the terrain, every `agxTerrain::TerrainMaterial` in the terrain has a link to an internal associated `agx::Material`. To access the associated materials with any terrain material, use the following `get/set` methods:

```
agx::Material* Terrain::getAssociatedMaterial(TerrainMaterial* terrainMaterial);
bool Terrain::setAssociatedMaterial(TerrainMaterial* terrainMaterial, agx::Material* material);
```

If no additional terrain materials have been added, the `getMaterial` method can be used to get all the relevant internal materials:

```
agx::Material* getMaterial(agxTerrain::Terrain::MaterialType type) const;
```

Where the `type` variable is either:

`MaterialType::TERRAIN`	This is the default material set on the terrain surface. This material can be extracted and used to construct contact material parameters between the terrain ground and vehicle.
`MaterialType::PARTICLE`	The internal material for the dynamic soil particles in the terrain internal simulation. This material is used for interactions with the dynamic soil.

The total `Shovel-Terrain` interaction feedback is a combination of `agxTerrain::TerrainMaterial` and `agx::Material` properties. This can be configured in the following ways:

1. Specifying the agxTerrain::TerrainMaterial objects in the Terrain. There are pre-calibrated material presets for the `agxTerrain::TerrainMaterial` objects to choose from, if you do not wish to specify their properties manually.
2. Extracting the associated `agx::Material` instances from the terrain in order to create shovel-terrain contact materials for objects interacting with the terrain. This will affect interaction force feedback with the tool.

Note

The coupling between the Shovel and the dynamics soil particles are kinematic. The force feedback on the shovel comes from contact with soil aggregates that are constructed from the particles. So the defined Shovel <-> Terrain contact materials is what is driving excavation and penetration resistance.

23.6.2. Shovel - Terrain Contact Materials

The contact materials specified between the shovel and the terrain are some of the primary determinants of excavation resistance. The contact properties are used in the shovel-aggregate contacts for all excavation modes. It is also used in regular geometry contacts between the shovel and the terrain when excavation mode is active. For each soil type added to the terrain instance there is a contact material between that soil types associated material and the shovels material.

To set up the contact material for the default part of the terrain, the following code can be used:

```
agx::MaterialRef shovelMaterial = new agx::Material("Shovel");
simulation->add(shovelMaterial);

// ... set the shovel material on the shovel bodies
agxUtil::setBodyMaterial(shovelBody, shovelMaterial);

agx::MaterialRef terrainMaterial = terrain->getMaterial( agxTerrain::Terrain::MaterialType::TERRAIN );
agx::ContactMaterialRef shovelTerrainCM = new agx::ContactMaterial( shovelMaterial, terrainMaterial );
shovelTerrainCM->setFrictionCoefficient( 0.4 );
shovelTerrainCM->setYoungsModulus( 1e8 );
shovelTerrainCM->setRestitution( 0.0 );
shovelTerrainCM->setAdhesion( 0, 0 );
simulation->add( shovelTerrainCM );
```

To set up the contact material with a part of the terrain where another soil type was added, the following code can be used:

```
agx::MaterialRef shovelMaterial = new agx::Material("Shovel");
simulation->add(shovelMaterial);
// Get and add a terrain material
agxTerrain::TerrainMaterial dirt = terrain->getLibraryMaterial("dirt_1");
terrain->addTerrainMaterial(dirt);
// Get the associated material
agx::Material dirtAgxMaterial = terrain->getAssociatedMaterial(dirt);
// Define the contact material between dirt<->shovel
agx::ContactMaterialRef shovelDirtCM = new agx::ContactMaterial( shovelMaterial, dirtAgxMaterial );
shovelDirtCM->setFrictionCoefficient( 0.4 );
shovelDirtCM->setYoungsModulus( 1e8 );
shovelDirtCM->setRestitution( 0.0 );
```

```
shovelDirtCM->setAdhesion( 0, 0 );
simulation->add( shovelDirtCM );
```

Refer to the section Terrain material configuration for a description of how to add different soil types to the terrain and the section on internal materials to configure associated materials with the terrain materials.

Note

When the shovel interacts with a part of the terrain that intersects two or more regions where there are different soil types, the contact material properties are averaged.

23.6.3. Object - Terrain Contact Materials

The `Object-Terrain` contact materials can be used to calibrate the interaction between the terrain and external objects such as wheels or tracks. The code to setup these contact materials are similar to how the Shovel-Terrain contact materials are defined. Just use the `agx::Material` of the object instead of the shovel!

```
agx::MaterialRef objectMaterial = new agx::Material("ExternalObject");
simulation->add(objectMaterial);

// ... set the object material on the external bodies
agxUtil::setBodyMaterial(externalObjectBody, objectMaterial);

agx::MaterialRef terrainMaterial = terrain->getMaterial( agxTerrain::Terrain::MaterialType::TERRAIN );
agx::ContactMaterialRef objectTerrainCM = new agx::ContactMaterial( objectMaterial, terrainMaterial );
objectTerrainCM->setFrictionCoefficient( 0.4 );
simulation->add( objectTerrainCM );
// Get and add another terrain material
agxTerrain::TerrainMaterial dirt = terrain->getLibraryMaterial("dirt_1");
terrain->addTerrainMaterial(dirt);
// Get the associated material
agx::Material dirtAgxMaterial = terrain->getAssociatedMaterial(dirt);
// Define the contact material between dirt<->shovel
agx::ContactMaterialRef objectDirtCM = new agx::ContactMaterial( objectMaterial, dirtAgxMaterial );
simulation->add( objectDirtCM );
```

Refer to the section Terrain material configuration for a description of how to add different soil types to the terrain and the section on internal materials to configure associated materials with the terrain materials.

Note

If the external object interacts with a part of the terrain that intersects two or more regions where there are different soil types, the contact material properties are averaged.

23.7. Calibration of Excavation Resistance

The terrain generates excavation resistance through the following effects:

Separation resistance	Generated by digging and pushing soil in the forward digging direction.
Penetration Resistance	Force generated via penetration of the cutting plate in the tooth direction.
Deformation Contact Forces	Similar to the separation forces but generated during *push/pull/grading* operations.
Generic Contact Forces	These are the regular contact forces when no excavation mode is active.

Note

It is recommended that the user always check their excavation forces during terrain setup. This is in order to properly asses the feasibility of the forces and phenomena that is generating feedback.

The force feedback that is generated on a shovel can be extracted by the following methods:

```
bool Shovel::getPenetrationForce(agx::Vec3& force, agx::Vec3& torque) const;

agx::Vec3 Shovel::getSeparationContactForce() const;

agx::Vec3 Shovel::getDeformationContactForce() const;
```

```
agx::Vec3 Shovel::getContactForce() const;
```

The main ways to adjust these are:

1. Contact properties between the shovel and the terrain aggregate. This is controlled by the contact material specified between the shovel geometries and the different soil types in the terrain instance. See contact materials. This affects both penetration resistance and separation/deformation resistance.
2. Terrain Material properties, such as soil density, stiffness and friction angle. This determines soil aggregate weight and also internal contact properties, i.e contacts between the aggregate and the terrain.
3. Adjust the penetration resistance by modifying the `PenetrationForceScaling` parameter.

23.7.1. Contact material calibration

The following properties are the most relevant when calibrating the contact materials between the shovel and terrain.

`YoungsModulus`	Determines the stiffness in the shovel <-> terrain contacts.
`FrictionCoefficient`	Adjusts the friction force between the shovel and terrain.

23.7.2. Terrain Material

The following parameters determines the internal contact and shape properties of the soil aggregates. Most of them are found the in the bulk properties but some are also found in excavation contact properties. To see how to add/remove terrain materials, please refer to the section on Terrain material configuration.

`YoungsModulus`	Determines the stiffness in the aggregate <-> terrain contacts.
`Cohesion`	Affects the force feedback from the terrain via setting a cohesion value in the aggregate <-> terrain contacts. Also affects the cohesion of the dynamic mass hat might lead to higher measured masses.
`FrictionAngle`	This affects the angle of the failure zones and also determines the friction coefficient in the aggregate <-> terrain contacts.
`Density`	Affects the total mass of the soil aggregates formed in the shovel active zone.
`ExcavationStiffnessMultiplier`	Scales the bulk Young's modulus applied in the shovel <-> aggregate contact in order to allow the user to fine-tune the excavation force.
`AggregateStiffnessMultiplier`	Scales the bulk Young's modulus applied in the aggregate <-> terrain contact in order to allow the user to fine-tune the excavation force.

23.7.3. Calibration of Penetration Force

The user can explicitly control the penetration force either by apply a scaling factor or a maximum cap on the penetration force:

```
shovel->setPenetrationForceScaling(5.0);
shovel->setMaxPenetrationForce(10000); // In Newton
```

Note

The need to apply a scalar scaling factor on the penetration model is due to the fact that shovels have different geometry and cutter formation that interact in a way that the current model is unable to capture. Therefore it might be necessary to calibrate the scaling factor to achieve the desired initial breach force.

23.7.4. Suggested calibration workflow

This section provides a sample workflow for calibrating agxTerrain. This list assumes that the user has access to benchmark data from either theory or a experimental test that they wish to reproduce in agxTerrain:

- First ensure that the TerrainMaterial->BulkProperties holds the same parameters as the real world soil used in the experiments. Also use values obtained from experiments and literature. *Density is especially important*.
- Ensure that the final mass in the shovel agrees with the end mass in the test specified. This involves modifying the following bulk properties:
    - Update `Bulk Properties - Swell Factor` This determines how the material grows or swells after excavation, emulates de-compaction of soil into a loosely packed state.
    - Update `Bulk Properties - Bulk density`.

    - Update `Bulk Properties - Cohesion`. High cohesion values will cause the material to stick together more, creating more material in the shovel.
- Calibrate the Penetration Resistance Force Scaling so that the force feedback during the initial breach agrees with measured data. Recommended start interval 10-16.
- Modify BulkProperties.
    - Young's Modulus.
    - Friction Angle.
    - Cohesion.
- Contact material between Shovel and terrain (Suggested parameters).
    - Friction.
    - Young's modulus.
- Excavation Contact Properties (Suggested parameters, Used as a 'last resort').
    - AggregateStiffnessMultiplier.
    - ExcavationStiffnessMultiplier.

Note

CompactionProperties does not affect force feedback during excavation.

23.8. Terrain Paging

Terrain paging exists to solve the following problems:

- One needs a large area to operate in, which results in excessively high direct memory usage.
- One needs a resolution and an area that together result in excessively high direct memory usage.
- There are multiple machines that need to operate on terrain simultaneously at too great a distance for a terrain to handle it (due to memory usage).
- The terrain is defined in a format that does not allow for direct creation via an API in agxTerrain, for example, a triangle mesh.

This is achieved via the `agxTerrain::TerrainPager` class. It has a virtual 2D grid with square tiles in which it can load in `agxTerrain::Terrain` instances as needed. The Terrain instances are unloaded again when they are not needed any longer.

```
/**
 Constructor. Specifies tile information and the 2D-plane used for dynamic terrain tiles.

 The TerrainPager do not mirror all the different parameters a terrain exposes,
 instead a template terrain is used for settings.

 To support excavation, the terrain tiles have an optional margin. This margin can
 be set so that the excavation is performed within one tile and correct soil wedges
 are formed.

 Example: tileResolution = 301, tileOverlap 10, tileElementSize = 0.25, ....

          Each tile then becomes (301-1)*0.25 = 75m x 75m
          The overlap on each side will be 10*0.25 = 2.5m

 \param tileResolution    Specifies the number of height values for each dimension for (tile+margin).
 \param tileOverlap       Specifies the number of height values that should overlap between two tiles.
 \param tileElementSize   Size in meters for one element. Same as distance between two height values.
 \param maximumDepth      How deep in meters the terrain can be excavated
 \param referencePoint    A world reference point for where the terrain is located
 \param referenceRotation A rotation that will transform the Z-axis to the up direction at the referencePoint
 \param templateTerrain   A template terrain whose settings will be used for the terrain tiles.
 */

 TerrainPager(size_t tileResolution, size_t tileOverlap, agx::Real tileElementSize, agx::Real maximumDepth,
              agx::Vec3 refPoint, agx::Quat refRotation,
              agxTerrain::Terrain* templateTerrain );
```

Note

It is important that the edges of two adjacent tiles match. To handle this and support excavation, adjacent tiles can have an overlap. When excavation is performed, the soil wedges are kept within a tile and to have non-truncated soil wedges. The overlap should be larger than both the length of the cutting edge interacting with the terrain aswell as twice the expected maximum length of a soil-wedge (which depends on how deep excavation in the terrain will be performed).

Warning

Adding a Terrain instance to a Simulation will also set some Solver parameters that the terrain needs and cause the Simulation update task to be rebuilt. It is not supported to rebuild the update task when it is executing, that is, during Simulation::stepForward. It can cause Simulation::stepForward to not return.

The TerrainPager must be able to add and remove terrains during stepForward. Hence, Simulation::add(TerrainPager) prepares the Simulation by setting the same solver parameters that Simulation::add(Terrain) would do, preventing a later task rebuild.

After a TerrainPager has been added to a Simulation, it is not supported to change any of the following solver parameters: - setUseParallelPgs - setUse32bitGranularBodySolver - setUseGranularWarmStarting Also, adding a TerrainPager to the Simulation will change the BroadPhase Algorithm used by Space to HIERARCHICAL_GRID. It is not supported to change it to SWEEP_AND_PRUNE.

23.8.1. Tracking bodies / shovels

To know which tiles that should have Terrain instances, bodies and shovels can be added to the `agxTerrain::TerrainPager` for tracking. Each tracked item have two radiuses associated with them:

- The required radius: All tiles within this radius must be in the Simulation for `agxSDK::Simulation::stepForward` to be able to proceed. Can cause `stepForward` to wait while the tile processing occurs in the background.
- The preload radius: Tiles within this radius will be preloaded in the background and inserted into the Simulation when ready.

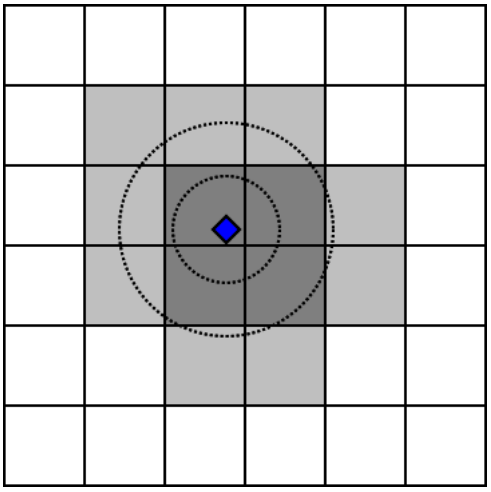


Fig. 23.8 Figure that shows a tracked item in blue, required tiles in dark gray and tiles within the preload radius in light gray. Normally the tiles are much larger than in this example image and fewer tiles are active.

The value for the required radius should be selected so that all geometries attached to the body that may interact with the terrain are enclosed within the radius. The value for the preload radius should be chosen so that tiles may be loaded into memory well before they will be needed in the simulation to avoid hiccups. This means the higher the expected maximum velocity the higher the preload radius relative to the required radius.

```
/**
Adds a RigidBody to the terrain pager. The pager will make sure that terrain tiles
within the required radius are present in the Simulation before stepping and tiles
within the preload radius will be loaded and inserted when available.
*/
bool add( agx::RigidBody* body, agx::Real requiredTileRadius, agx::Real preloadTileRadius );


/**
Remove body so that the terrain pager no longer uses the body when determining
which terrain tiles that are needed.
*/
bool remove( agx::RigidBody* body );
```

If a shovel is added to the pager, the shovels rigid body will be tracked as outlined above. The pager needs to be added to a simulation in order to register shovels.

23.8.2. TerrainDataSource

The `TerrainPager` must also have a `TerrainDataSource` from which it can request height data for the needed tiles. The first time a tile is needed, heights are requested. If the tile is paged out then the height values are stored to disk. If it is needed again the possibly modified values are read from file and not requested from the data source.

There is one class that implements the `TerrainDataSource` interface: `agxTerrain::TerrainRasterizer`. The `TerrainRasterizer` should be given source geometry (e.g. large heightfield(s)) which it will then sample with raycasting to get height values at the needed positions.

Note

When implementing a custom data source in languages other than C++ using the AGX bindings, the `ExternalTerrainDataSource` class should be inherited instead of the `TerrainDataSource` class and serialization of the data source is disabled

23.8.3. Paged Terrain settings

The TerrainPager do not expose the material settings for the terrain tiles. Instead, one of the arguments to the TerrainPager constructor is a template terrain. Each time a terrain tile is inserted into the simulation, the template settings are applied to the new tile.

```
/**
Fetches the template terrain.
The settings for the template terrain are used on new terrain tiles
when they are inserted into the simulation.

\see applyChangesToTemplateTerrain
*/
agxTerrain::Terrain* getTemplateTerrain();

/**
Signals to the TerrainPager that the template terrain has been updated
and that any changes to the template terrain should be applied to the
active terrain tiles in the pager.
*/
void applyChangesToTemplateTerrain();
```

Each time a new Terrain instance is inserted into the Simulation, the TerrainPager triggers an event, `tileLoadEvent`, and in the same way when removing a Terrain `tileUnloadEvent` is triggered. With these events it it also possible to customize tiles settings per tile.

Compaction data for paged out tiles is not stored by default. If this is needed the method `TerrainPager::setShouldStoreCompaction( bool, size_t )` can control this and specify how many voxels below the surface that should be stored.

Note

The amount of stored data per paged tile will increase greatly if `maximumCompactionDepth` is set too high.

23.8.4. Serialization

The terrain pager has two ways of serialization. There is a full state serialization using InputArchive and OutputArchive and there is a serialization of modified terrain height data.

The full state serialization use the functions `void TerrainPager::store(agxStream::OutputArchive& out) const` and `void TerrainPager::restore(agxStream::InputArchive& in)`.

Part of the state held by the terrain pager can be paged out to disk (see `TerrainPager::setFileCacheDirectory`). The files in that directory are needed along with the serialized archive. If needed, the paged out data can be embedded in the archive. This is controlled via `TerrainPager::setEmbedCacheFilesInArchive( bool )`.

Furthermore, compaction data is not stored by default. If this is needed the method `TerrainPager::setShouldStoreCompaction( bool, size_t )` can control this and specify how many voxels below the surface that should be stored.

Note

The amount of stored data per paged tile will increase greatly if `maximumCompactionDepth` is set too high.

The serialization for the changed heights uses the methods `void TerrainPager::store( agxTerrain::TileModificationVector& storeData );` `void TerrainPager::restore( const agxTerrain::TileModificationVector& restoreData );` this is described in more detail below.

23.8.5. Store / restore of terrain height data

There are custom store/restore functions which can be used for serialization of paged terrains heights if needed.

If `TerrainPager::setShouldStoreDeltas` is enabled, then terrain pager will start collecting height changes internally. Unless these values are fetched periodically, the internal structure will keep growing. These delta changes can be fetched via `void TerrainPager::store( agxTerrain::TileModificationVector& storeData );`. The output is written to `storeData` and will contain the changes since last time the function was called. After `store` has been called, the TerrainPager will clear its internal buffer with changes.

To restore the system, `void TerrainPager::restore( const agxTerrain::TileModificationVector& restoreData );` should be used. The restoreData should be the aggregated state from all the store deltas. This will give the terrain pager a set of changes which should be applied to a tile after the height data has been requested from the terrain data source.

23.8.6. Terrain paging tutorials

A full example of how the TerrainPager and TerrainRasterizer can be used along with dynamic loading/unloading of Terrain tiles can be seen in `data/python/agxTerrain/basic_paging_example.agxPy`.

There is also an example which uses a kinematic body to control which tiles that are preloaded and performs efficient raycast measurements of the terrain height `data/python/agxTerrain/paging_depth_measurements.agxPy`.

There is also a tutorial showing bulldozing with a terrain pager `data/python/agxTerrain/tutorials/agxTerrain/tutorial_bulldozing_terrain_paging.agxPy`.

23.9. Additional Terrain Tools

This section covers the usage of special tools for interacting with the terrain that are not covered by the standard shovel model. These types of tools are often are represented with two or more interacting shovels that requires synchronization.

23.9.1. Clamshell Bucket

`agxTerrain` support the simulation of Clamshell buckets which are a two-sided, hinged buckets used in earthmoving and material handling operations such as dredging, construction and bulk material handling. The bucket often operates using hydraulic cylinders or mechanical cables that open and close the shells. It can be attached to cranes, excavators, or other lifting machines.

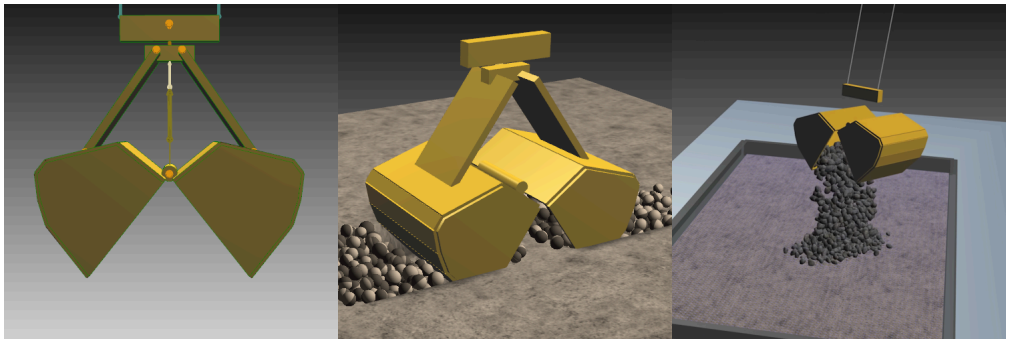


Fig. 23.9 Figure that demonstrates the construction and operation of a ClamShellBucket. Left image shows a basic layout of the construction of two hinged buckets that opens and closes via a central prismatic joint. Middle image shows a excavation operation where the clam shell is in the closing stage before extracting soil material. The right image shows an unloading operation of a crane steered clam bucket where bulk material is unloaded on top of a dump terrain after excavation.

In order for the internal terrain calculations to work properly for a clam shell construction an `agxTerrain::ClamShellBucket` object needed used to connect the soil aggregates of both shovels when they are in a closed state:

```
// Create the clam shell object that will handle the internal terrain logic
// of connecting the soil aggregates when the buckets are closed.
agxTerrain::ClamShellBucketRef clamShellBucket = new agxTerrain::ClamShellBucket( bucket1, bucket2 );

// The threshold distance between the two shovel cutting edges under
// which the clam shell is considered in closed state.
clamShellBucket->setClosedThreshold(0.1) // in meters

// Add the bucket to the simulation
simulation->add(clamShellBucket)
```

Note

Note that the `agxTerrain::ClamShellBucket` class does not handle the mechanical construction of the bucket, that needs to be done via modelling via constraints and rigid bodies.

For an example of a crane steered Clamshell Bucket in an excavation scenario, see `data/python/tutorials/agxTerrain/tutorial_clamshell_bucket.agxPy`.

23.9.2. External Deformers

External deformers can be attached to any rigid body object to make it able to push and deform the terrain, similar to the deformers on the sides and bottom of a shovel. Example usecases can be attaching external deformers on the sides of tracks to be able to push soil when turning.

The external deformer is initiated by attaching it to a rigid body and by adding a deformation plane, defined by three lines: a cuttingEdge, topEdge and cuttingDirection.

```
# Set up a rigid body and an external deformer object.
box_half_vec = agx.Vec3(0.5, 0.5, 0.5)
body = agx.RigidBody(agxCollide.Geometry(agxCollide.Box(box_half_vec)))
simulation().add(body)
ed = agxTerrain.ExternalDeformer(body)

# Add the deformer plane.
cuttingEdge = agx.Line(agx.Vec3(0.5, -0.5, -0.5), agx.Vec3(0.5, 0.5, -0.5) )
topEdge = agx.Line(agx.Vec3(0.5, -0.5, 0.5), agx.Vec3(0.5, 0.5, 0.5) )
cuttingDirection = agx.Vec3(0, 0, -1)
ed.addDeformer(cuttingEdge, topEdge, cuttingDirection)

# Add the deformer to the simulation.
sim.add(ed)
```

23.10. Forbidden bounds

It is possible to add bounding volumes based on the `agxTerrain::IForbiddenBound` interface in either positions or local terrain index coordinates to `agxTerrain::Terrain` objects that prevents avalanching and merging of particles into the terrain indices that overlap the bounds. The following snippet shows how to add a bound based on local terrain index:

```
agxTerrain::TerrainRef terrain = new agxTerrain::Terrain( 51, 51, 0.1, 2.5 );
simulation->add( terrain );

// Add a 3D index bound to the terrain that covers the left half of the terrain.
agxTerrain::IForbiddenBoundRef ib = new agxTerrain::ForbiddenIndexBound( agx::Bound3i( agx::Vec3i( 0, 0, -5 ),
                                                                                       agx::Vec3i( 25, 51, 5 ) ) );
terrain->addForbiddenBound( ib );
```

Spatial 3D bounds can also be created and attached to the frames of bodies and geometries so that they can follow the object's transform:

```
// Add a moving rigid body
agxCollide::GeometryRef box_geom = new agxCollide::Geometry( new agxCollide::Box( agx::Vec3( 0.25, 0.25, 0.25 ) ) );
agx::RigidBodyRef rb = new agx::RigidBody( box_geom );
simulation->add( rb );

// Add a 3D forbidden bound attached to the frame of the rigid body.
// Add a local transform offset for rotation and position to the bound
agx::AffineMatrix4x4 localTransform( agx::EulerAngles( 0, 0, agx::PI/4.0 ), agx::Vec3( 0, 0, -0.5 ) );
// Create the bound and use the agx::Frame of the RigidBody as the parent transform via rb->getFrame
agx::Vec3 halfVec( 0.5, 2.5, 0.5 );
agxTerrain::IForbiddenBoundRef fb = new agxTerrain::ForbiddenBound( agxCollide::BoundingAABB( -halfVec, halfVec ),
                                                                    localTransform,
                                                                    rb->getFrame());
terrain->addForbiddenBound( fb );
```

Bounds can also be removed from the terrain:

```
agxTerrain::IForbiddenBoundRef fb = new agxTerrain::ForbiddenBound ...
terrain->addForbiddenBound(fb)
...
terrain->removeForbiddenBound(fb)
```

23.11. Terrain agxJournal support

`agxTerrain::Terrain` height data changes during a simulation can be stored to an agxJournal file and be replayed during journal playback.

The `agxTerrain::TerrainJournalRecorder` class can be used to record the terrain height data to a journal file during journal recording and automatically restore it during playback:

```
agxTerrain::TerrainJournalRecorderRef recorder = new agxTerrain::TerrainJournalRecorder(terrain);
simulation->add(recorder);
```

Note that an argument can be supplied when running a terrain script via ExampleApplication that automatically adds this class for each terrain object in the simulation.

```
python myTerrainScript.py --journalRecord --terrainJournalRecord
agxViewer myTerrainScript.agxPy --journalRecord --terrainJournalRecord
```

23.12. Terrain Model

This section contains model descriptions of the different parts in the *agxTerrain::Terrain* object.

23.12.1. Terrain Grid Structure

The terrain data is stored in an underlying 3D grid structure. Each gridpoint contains the following information:

- Solid Mass occupancy
- Soil Compaction
- Terrain Material information

Surface terrain data, such as terrain height, is accessed by specifying a *terrain surface index*, which references a single discrete surface grid point on the terrain. The terrain surface index consists of 2D x and y coordinates with origin `(x:0, y:0)` in the lower leftmost gridpoint and the maximum coordinate in the upper rightmost gridpoint `( x: resolutionX-1, y: resolutionY-1 )` in the terrain surface structure.

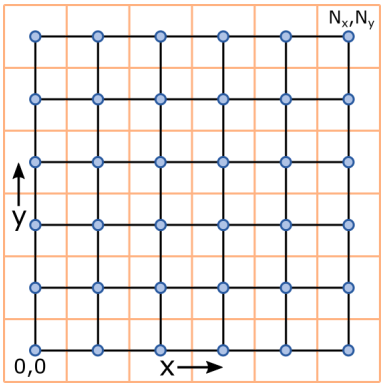


Fig. 23.10 Figure that shows the indexing and grid layout of the `agxTerrain` in the x/y surface plane. The blue points denote grid center points that coincide with the surface heightfield vertices. The orange grid denotes the cell boundaries. The indexing starts in the lower left corner with `x:0,y:0` and goes to the upper right corner `x:resolutionX-1, y:resolutionY-1`.

To access full spatial data, a 3D grid coordinate `(x,y,z)` is specified using the same 2D surface index as above but also with an added depth index. The **origin of the depth index is defined where the surface height of the terrain is zero**, and the **lowest allowable z index corresponds to the maximum depth of the terrain**.

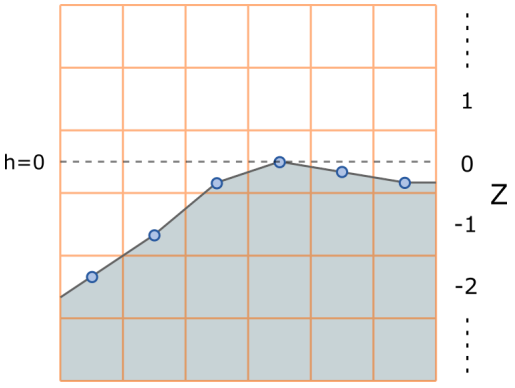


Fig. 23.11 Figure that illustrates the layout of the depth index in the terrain cell structure via cross-section view in the x/z plane. The orange grid shows the cell boundraries in the terrain structure and the blue points shows the height field vertex height positions. The local terrain center z position of the grid cell with depth index 0 is defined where the height of the coupled heightfield is zero.

A uniform maximum depth is default configuration for the terrain model, but the terrain can be configured to use specific minimum heights in each *terrain index* with the following functionality:

```
/*
\return bool indicating whether the terrain should use minimum heights instead of uniform maximum depth.
*/
bool getEnableMinimumHeights() const;

/*
Enable (true) or disable (false) the usage of minimum heights in the grid. It is initialized at the maximum depth
if not enabled beforehand.
*/
void setEnableMinimumHeights(bool enable);

/*
Set the minimum height allowed at the specified terrain index.
*/
void setMinimumHeight(const agx::Vec2i& terrainIndex, agx::Real minimumHeight);

/*
Gets the minimum height allowed at the specific terrain index.
*/
agx::Real getMinimumHeight(const agx::Vec2i& terrainIndex) const;

/*
Set the minimum height values in the terrain grid. If minimum heights was not enabled before, it will be enabled with
this method.
\param minimumHeights - must be a row major array with dimensions (resolutionX * resolutionY).
*/
bool setMinimumHeights(const agx::RealVector& minimumHeights);


/**
Return row major array with dimension (resolutionX * resolutionY) with the minimum height allowed in each grid point.
If minimum heights are disabled, this is equivalent to the value of getLowestAllowableHeight in all grid points.
If minimum heights are enabled, this can be a varying field.
*/
agx::RealVector getMinimumHeights() const;
```

3D grid data can be accessed via the TerrainGridControl interface. If the user wants to extract the depth index for the terrain surface given a surface index the following code can be used:

```
/**
Find terrain surface depth index given the terrain index x,y coordinates.
\param terrainIndex - The specified 2D terrain surface index.
\return The terrain depth index (z) at the surface.
*/
int TerrainGridControl::findSurfaceIndex(const agx::Vec2i& terrainIndex) const;
```

23.12.2. Mass Data

Each grid cell contains information about how much solid terrain mass it contains. It is represented as cell `occupancy`. Assuming nominal compaction (1.0), each cell can have an occupancy value that can range from 0.0 (empty) to 1.0 (full). If the cell has a compaction value other than 1.0, the amount of solid `occupancy` that can be stored in a grid cell changes according to:

$$\phi_{max_i} = \phi_{max} * c_i$$

where ϕ_{max_i} is the new maximum occupancy that can be stored in cell i, ϕ_{max} is the nominal maximum occupancy that can be stored in a cell (1.0) and c_i is the compaction in cell i. This effectively means that more mass can be contained in dense packed soil and less in looser soil.

23.12.3. Accessing Grid Data

Grid data can be modified or inspected at the grid level through the `agxTerrain::TerrainGridControl` which can be accessed from the `Terrain` object:

```
auto gridControl = terrain->getTerrainGridControl();
// Get the mass occupancy of the terrain 3D cell at (0, 0, 0)
```

```
compaction gridControl->getOccupancy(agx::Vec3i(0, 0, 0));
```

Terrain solid mass is modified in the `occupancy` format top-down on the terrain surface at a 2D index. Either is it added via specifying the amount `occupancy` or by filling/removing up to a certain height or layer.

Note

The terrain does not support discontinuous terrain mass in the z-direction due to the nature of the surface heightfield. That is why mass can only be added/removed columnwise top-down on the terrain surface.

The following snippet shows some functions that enables the user to do this:

```
float addSolidOccupancyInColumn(const agx::Vec2i& terrainIndex, float occupancy, float compaction, bool shouldAvalanche =
true);
float removeSolidOccupancyInColumn(const agx::Vec2i& terrainIndex, float occupancy, bool shouldAvalanche = true);
float addSolidOccupancyLayerInColum(const agx::Vec2i& terrainIndex, agx::Real layerHeight, float compaction, bool
shouldAvalanche = true);
float addSolidOccupancyInColumnToHeight(const agx::Vec2i& terrainIndex, agx::Real height, float compaction, bool
shouldAvalanche = true);
```

The following snippet shows how to add a layer of loose soil on top of the terrain surface via the `agxTerrain::TerrainGridControl` object:

```
auto gridControl     = terrain->getTerrainGridControl();
auto terrainMaterial = terrain->getTerrainMaterial()->getBulkProperties();

// Add a loose layer of 0.5 m on top of the terrain that has compaction according to the swell factor
agx::Real compaction = 1.0 / terrainMaterial->getSwellFactor();
agx::Real height     = 0.5; // height in meters
for ( int x = 0; x < terrain->getResolutionX(); x++ )
  for ( int y = 0; y < terrain->getResolutionY(); y++ )
  {
    gridControl->addSolidOccupancyLayerInColum( { x, y },
                                                height,
                                                compaction,
                                                true );
  }
```

Often one wants to extract the depth index for the terrain surface given a surface index:

```
/**
Find terrain surface depth index given the terrain index x,y coordinates.
\param terrainIndex - The specified 2D terrain surface index.
\return The terrain depth index (z) at the surface.
*/
int TerrainGridControl::findSurfaceIndex(const agx::Vec2i& terrainIndex) const;
```

23.13. Terrain Properties

The general properties of the terrain is stored in a `agxTerrain::TerrainProperties` object which can be accessed:

```
agxTerrain::TerrainProperties* properties =  terrain->getProperties()
```

These properties contains useful options for locking terrain border height, enable/disable for various terrain functions.

The following function locks the terrain borders from changing due to avalanching or merging.

```
properties->setEnableLockedBorders(true)
```

The following function removes particles that are outside the bounds of the terrain:

```
properties->setDeleteSoilParticlesOutsideBounds(true)
```

The following function set the lowest speed limit for soil particles before they start to merge back into the terrain:

```
properties->setSoilMergeSpeedThreshold(0.06)
```

The following function set the maximum volume that can be activated in the terrain due to shovel excavation operations:

```
properties->setMaximumParticleActivationVolume(0.5)
```

The following function scales the penetration resistance with the speed of the shovel in the terrain:

```
properties->setPenetrationForceVelocityScaling(0.05)
```

Particle size can be scaled in the simulation to either increase or decrease the amount of soil particle created during excavation. The scaling parameters multiplies with the target nominal radius of the particles:

properties->setSoilParticleSizeScaling(1.2)

23.14. Terrain Material

`agxTerrain::TerrainMaterial` is used to describe the material properties of the soil in agxTerrain. The properties are grouped internally in objects according to the phenomena they affect:

`BulkProperties`	These properties govern general bulk behaviour such as soil density, failure zone shape and excavation forces.
`CompactionProperties`	These properties govern behaviour of the soil compaction.
`ParticleProperties`	These properties govern the soil particle dynamics. Mostly contains parameters for the contact material in-between particles and particle <-> terrain contacts.
`ExcavationContactProperties`	These properties govern the contact model of the internal terrain <-> aggregate and the shovel <-> aggregate contacts.

23.14.1. BulkProperties

The bulk properties govern the general bulk behaviour of the terrain, such as failure zone shape, soil density and feedback forces. It contains the following parameters:

`YoungsModulus`	The bulk stiffness of the material in Pa. This affects the stiffness in the internal aggregate<->terrain contacts.
`Cohesion`	Describes the internal cohesion of the material in Newton. This affects the force feedback from the terrain. It also determines the cohesion in aggregate<->terrain contacts.
`FrictionAngle`	Describes the internal friction angle of the material. This affects the angle of the failure zones and angle of repose in a 1-to 1 ratio for the avalanching algorithm. It also determines the friction coefficient in aggregate<->terrain contacts.
`Density`	Describes the density of the soil in kg/m3. This translates to the specific density of the solid soil and the bulk density of the dynamic soil (i.e soil particles).
`Maximum Density`	Describes the maximum density of the soil in kg/m3. This implicitly determines the maximum compaction of the material.
`Swell Factor`	Specifies the fractional increase in bulk volume that occurs when solid soil is converted to dynamic soil.
`Dilatancy Angle`	Specifies the dilatancy angle of the material which affects the internal friction of the material. It is affected by compaction. Used in Penetration resistance and Separation Resistance calculations.
`Poisson's Ratio`	Specifies the Poisson's ratio of the material. Used in Penetration resistance calculations.
`Adhesion Overlap Factor`	Adhesion Overlap Factor is the fraction of the soil particle radius that is allowed to overlap. This is used to simulate adhesion. Default value is 0.05
`Delta Repose Angle`	The delta repose angle increases the base angle of repose of the material on top of the internal angle of friction. Default value is 0.0

Note

The `Swell Factor` is inversely correlated to the compaction level of the dynamic soil when it is merged back to the terrain.

The bulk properties can be accessed and modified via the TerrainMaterial object:

```
auto terrainMaterial = terrain->getTerrainMaterial()
auto bulkProperties = terrainMaterial->getBulkProperties()
```

```
bulkProperties->setDensity( 1.4e3 )
bulkProperties->setFrictionAngle( 23.0 * ( 180.0 / agx::PI ) )
bulkProperties->setSwellFactor( 1.05 )
bulkProperties->setAdhesionOverlapFactor( 0.07 )
```

23.14.2. Excavation Contact Properties

The excavation contact properties govern the some of the contact dynamics between the shovel, soil aggregates and the terrain during excavation and deformation. Bulk Material properties are used to set parameters such as stiffness, friction and cohesion in the terrain-aggregate and terrain-shovel contacts, but the excavation contact properties allows to user to access and manipulate the underlying contact model. The contact model between the soil aggregate and terrain aims to model elasto-plastic behaviour with a maximum depth limit and decay factors. There is also parameters for controlling the stiffness of the shovel-aggregate contact interface.

`ExcavationStiffnessMultiplier`	This adjust the stiffness of the shovel-aggregate contacts by multiplying the Young's Modulus of the material that is set to the contact. See aggregate<->terrain contacts.
`Depth Increase Factor`	Specifies the rate of depth increase in the aggregate-terrain contact when excavating and deforming.
`Depth Decay Factor`	Specifies the rate of depth decay in the aggregate-terrain during the separation stage of the excavation and deformation contact.
`Maximum Depth`	Specifies the maximum depth of the aggregate-terrain contact.
`Maximum Aggregate Force`	Sets the maximum normal force that can be applied in the aggregate-terrain contact.

The excavation properties can be accessed and modified via the `TerrainMaterial` object:

```
auto terrainMaterial = terrain->getTerrainMaterial();
auto excavationProperties = terrainMaterial->getExcavationContactProperties();
excavationProperties->setExcavationStiffnessMultiplier(0.05);
```

23.14.3. Compaction Properties

The compaction properties govern how the soil compacts under pressure generated from terrain contacts. See the soil compaction section for model details.

`Hardening Ke`	Hardening constant that determines how Young's Modulus changes with compaction.
`Hardening Ne`	Hardening constant that determines how Young's Modulus changes with compaction.
`preconsolidationStress`	The initial consolidation stress that the soil in the bank state (compaction = 1.0). Only stresses above this may increase compaction levels.
`compactionTimeRelaxationConstant`	The time relaxation constant that limits the rate of density changes due to compaction.
`stressCutOffFraction`	Determines the fraction of the surface stress that will determine when to stop propagating stress down into the soil.
`angleOfReposeCompactionRate`	Determines how angle of repose should change with varying compaction.

The compaction properties can be accessed and modified via the `TerrainMaterial` object:

```
auto terrainMaterial = terrain->getTerrainMaterial();
auto compactionProperties = terrainMaterial->getCompactionProperties();
compactionProperties->setCompressionIndex(0.05);
```

23.14.4. Particle Properties

The particle properties determines the properties for the 6-DOF particles representing dynamic soil and also the contact parameters in particle <-> particle contact and particle <-> terrain interactions. Data about soil particles can be accessed via the soil simulation interface.

`Density`	Determines the *specific* particle density. This is implicitly set via the density parameter in bulk properties.
`Particle Young's Modulus`	Sets the Young's Modulus (Pa) for the particle `agx::Material` and the internal particle <-> particle contacts.
`Particle Restitution`	Sets the restitution of the particle <-> particle contacts.
`Particle Surface Friction`	Sets the friction of the particle <-> particle contacts.

`Particle Rolling Resistance`	Sets the rolling resistance of the particle <-> particle contacts.
`Particle Cohesion`	Sets the cohesion (Pa) of the particle <-> particle contacts.
`Particle-Terrain Young's Modulus`	Sets the the Young's Modulus (Pa) of the particle <-> terrain contacts.
`Particle Terrain Restitution`	Sets the restitution of the particle <-> terrain contacts.
`Particle Terrain Surface Friction`	Sets the friction of the particle <-> terrain contacts.
`Particle Terrain Rolling Resistance`	Sets the rolling resistance of the particle <-> terrain contacts.
`Particle-Terrain Cohesion`	Sets the cohesion (Pa) of the particle <-> terrain contacts.

The particle properties can be extracted via the `TerrainMaterial` object:

```
auto terrainMaterial = terrain->getTerrainMaterial()
auto particleProperties = terrainMaterial->getParticleProperties()
particleProperties->setParticleRestitution(0.05)
```

23.15. Material Library

`agxTerrain` comes with a pre-existing library of `agxTerrain::TerrainMaterial` presets for different common soils that can be used. The available material presets are stored as external files in JSON format and are from parsed the default material library folder in AGX Dynamics: `data/MaterialLibrary/TerrainMaterials`. Users can also create their own material presets files representing other soil types specific to their use cases and add those to the material library folder or load them via the TerrainMaterialLibrary and TerrainMaterialReadWriter classes.

23.15.1. Loading a TerrainMaterial Preset

Loading a material preset into an `agxTerrain::Terrain` object is done like so:

```
if( terrain->loadLibraryMaterial( agx::String& filename ) );
  std::cout << "Material preset " << filename << "loaded successfully."
```

This loads the terrain material as the default terrain material. See Terrain Material Configuration for more information.

If the user wishes to load a terrain material and add it as an additional soil type to the terrain instance, the `getLibraryMaterial` can be used:

```
agxTerrain::TerrainMaterialRef terrainMaterial = terrain->getLibraryMaterial( agx::String& filename );
if( terrainMaterial != nullptr )
  std::cout << "Material preset " << filename << "loaded successfully."
```

The profiles that can be set correspond to the default set of .json files in the AGX Dynamics installation: `data/MaterialLibrary/TerrainMaterials`. The available default material presets can be extracted by using the following method:

```
terrain->getAvailableLibraryMaterials();
```

The default set of presets are as follows:

gravel_1	Represents typical material properties for coarse gravel.
sand_1	Represents typical material properties for fine sand.
dirt_1	Represents typical material properties for cohesive dirt.

Warning

Changing the bulk material properties or internal contact material parameters will invalidate the previous settings applied from the material library. The profile *must* be applied again in order to get the properties back.

23.15.2. agxTerrain::TerrainMaterialLibrary

`agxTerrain::TerrainMaterialLibrary` is a utility class that helps handling folders of `JSON` files specifying `agxTerrain::TerrainMaterial` presets. It parses a specified folder of files and stores verified material presets in tables according to material name. This class is used by the `agxTerrain::Terrain::loadLibraryMaterial` when loading material presets. By default, this library folder contains the default TerrainMaterial presets in AGX Dynamics. The user can specify their own folder with `JSON` files.

The following method loads a material profile according to material name in the specified library folder:

```
/**
Create an agxTerrain::TerrainMaterial object from one of the existing library
presets, derived from the .json files in the specified library folder. The default location
of the preset folder contains the default material preset of AGX Dynamics.
Note - Existing library materials can be extracted via the getAvailableLibraryMaterials() method:
\param materialName - the name of the material preset to load.
\param libraryFolder - The path to the folder containing all the terrain material preset files.
\return a TerrainMaterial object created from the specified material preset or nullptr if the creation failed.
*/
static agxTerrain::TerrainMaterial* loadMaterialProfile( const agx::String& materialName,
                                                         const agx::String& libraryFolder =
DEFAULT_TERRAIN_MATERIAL_LIBRARY_FOLDER );
```

The following method prints all available valid material presets in the specified library folder:

```
/**
Get the available TerrainMaterial presets from the existing.json files in the specified
material library folder. The default location of the preset folder contains the default
material preset of AGX Dynamics.
\param libraryFolder - The path to the folder containing all the terrain material preset files.
\return a vector containing the available TerrainMaterial presets from the existing
        .json files in the specified material library folder.
*/
static agx::StringVector getAvailableLibraryMaterials( const agx::String& libraryFolder =
DEFAULT_TERRAIN_MATERIAL_LIBRARY_FOLDER );
```

23.15.3. agxTerrain::TerrainMaterialReadWriter

The `agxTerrain::TerrainMaterialReadWriter` class is used to store and read material profiles in .json format and is used by the `agxTerrain::TerrainMaterialLibrary`. The following methods are used to read and store files:

```
/**
Read TerrainMaterial configuration from a file in JSON format into a TerrainMaterial object.
\param filename - the filename of the file containing the terrain material configuration in JSON format.
\param terrainMaterial - the TerrainMaterial object that will be initialized with the data from the JSON file.
\return true if the file was successfully read, false otherwise.
*/
static bool readFile( const agx::String& filename, TerrainMaterial* terrainMaterial );
```

```
/**
Write a TerrainMaterial configuration to a file in JSON format.
\param terrainMaterial - the TerrainMaterial object that will be written to the JSON file.
\param filename - the filename of the file where the the terrain material configuration will be written in JSON format.
\return true if the file was successfully written, false otherwise.
*/
static bool writeFile( const TerrainMaterial* terrainMaterial, const agx::String& filename );
```

23.15.4. TerrainMaterial .json files

The basic preset file structure is in `JSON` format that the reflects the underlying class layout of the TerrainMaterial class. Below follows a sample file for the preset `dirt_1` where comments describing the parameter units have been added:

```
{
  "BulkProperties" : {
      "adhesionOverlapFactor" : 0.050,       // Dimensionless
      "cohesion" : 12000.0,                  // Pa
      "density" : 1300.0,                    // kg/m3
      "dilatancyAngle" : 0.2268928027592629, // radians
      "frictionAngle" : 0.7504915783575616,  // radians
      "maximumDensity" : 2000.0,             // kg/m3
      "poissonsRatio" : 0.150,               // Dimensionless
      "swellFactor" : 1.280,                 // Dimensionless
```

```
        "youngsModulus" : 5000000.0,           // Pa
        "deltaReposeAngle": 0.0                // radians
    },
    "CompactionProperties" : {
        "angleOfReposeCompactionRate" : 24.0,        // Dimensionless
        "bankStatePhi" : 0.6666666666666666,         // Dimensionless
        "compactionTimeRelaxationConstant" : 0.050,  // Dimensionless
        "compressionIndex" : 0.110,                  // Dimensionless
        "hardeningConstantKE" : 1.0,                 // Dimensionless
        "hardeningConstantNE" : 0.08333333333333333, // Dimensionless
        "preconsolidationStress" : 98000.0,          // Pa
        "stressCutOffFraction" : 0.010               // Dimensionless
    },
    "ExcavationContactProperties" : {
        "aggregateStiffnessMultiplier" : 0.050,       // Dimensionless
        "depthDecayFactor" : 2.0,                     // Dimensionless
        "depthIncreaseFactor" : 1.0,                  // Dimensionless
        "excavationStiffnessMultiplier" : 1.0,        // Dimensionless
        "maximumAggregateNormalForce" : "inf",        // N
        "maximumContactDepth" : 1.0                   // m
    },
    "ParticleProperties" : {
        "particleCohesion" : 200.0,                   // Pa
        "particleFriction" : 0.40,                    // Dimensionless
        "particleRestitution" : 0.0,                  // Dimensionless
        "particleRollingResistance" : 0.10,           // Dimensionless
        "particleTerrainCohesion" : 200.0,            // Pa
        "particleTerrainFriction" : 0.70,             // Dimensionless
        "particleTerrainRestitution" : 0.0,           // Dimensionless
        "particleTerrainRollingResistance" : 0.70,    // Dimensionless
        "particleTerrainYoungsModulus" : 100000000.0, // Pa
        "particleYoungsModulus" : 10000000.0          // Pa
    },
    "name" : "DIRT_1"
}
```

23.16. Shovel Settings

The settings for a “Shovel” object are divided into two categories: those commonly used during initial setup and calibration, and those intended for more specific scenarios. Additionally, there are settings dedicated to the shovel’s various excavation planes.

23.16.1. Common Settings

The following are the more useful and generic settings for configuring the Shovel object:

Property	Description	Type | Default value
EnableParticleFreeDeformers	Enables/disables the deformers of the shovel to interact with the terrain without creating particles.	Boolean | False.
PenetrationDepthThreshold	Vertical penetration depth threshold for full effectiveness of penetration resistance.	Decimal | 0.5 m.
PenetrationForceScaling	Linear scaling coefficient for the penetration force exerted by the terrain.	Decimal | 1.0.
MaxPenetrationForce	Maximum limit on penetration force (N) that the terrain will generate on the shovel.	Decimal | Infinity.
ToothLength	Length of the shovel’s teeth.	Decimal | 0.15 m.
ToothMinimumRadius	Minimum radius of the shovel’s teeth.	Decimal | 0.015 m.
ToothMaximumRadius	Maximum radius of the shovel’s teeth.	Decimal | 0.075 m.
NumberOfTeeth	Integer signifying the number of teeth evenly distributed along the cutting edge of the shovel.	Integer | Calculated as ceil((cuttingEdge.p2 - cuttingEdge.p1).length() / 0.5).
VerticalBladeSoilMergeDistance	Vertical distance under the blade cutting edge for merging soil instantly.	Decimal | 0.0 m.
EnableGenerateTeethGeometry	Set whenever a teeth geometry should be generated between the cutting edge and the teeth edge from the teeth radius and length.	Boolean | True.

Property	Description	Type | Default value
EnableCustomTeethGeometry	Set whenever custom tooth geometry dimensions should be used instead of automatically generating a teeth geometry from the shovel settings	Boolean | False.
CustomTeethGeometryHalfVec	Set the custom tooth geometry box half vector dimensions to be used when generating a custom teeth geometry, if it is enabled.	Vec3 | (0, 0, 0).
EnableExcavationAtTeethEdge	Set whenever a excavation at the teeth edge should be enabled so that digging starts at the teeth edge instead of at the cutting edge.	Boolean | False.

The common settings are stored in a `ShovelSettings` class, and can be obtained from the `Shovel` object via the following code:

```
const auto& commonSettings = shovel->getSettings();
```

23.16.2. Advanced Settings

The following are some of the more advanced settings for the Shovel object:

Property	Description	Type | Default value
AlwaysRemoveShovelContacts	Determines whether shovel-terrain contacts are always removed.	Boolean | False.
EnableInnerShapeCreateDynamicMass	Indicates if the inner shape alone always creates dynamic mass.	Boolean | True.
EnableParticleForceFeedback	Indicates whether excavation force feedback is generated from particle contacts.	Boolean | False.
ZeroAggregateVelocity	Indicates if the innerbody aggregate should have zero velocity before the solve step.	Boolean | False.
UseCustomFrictionModel	Determines whether custom friction models are used for shovel-aggregate contacts.	Boolean | True.
NoMergeExtensionDistance	Specifies the extension distance outside the shovel bounding box where soil particle merging is forbidden.	Decimal | 0.5 m.
MinimumSubmergedContactLengthFraction	Minimum submerged cutting edge length fraction [0-1] for generating submerged cutting.	Decimal | 0.5.
SecondarySeparationDeadloadLimit	Dead-load limit where secondary separation activates.	Decimal | 0.8.
ContactRegionThreshold	Starting distance threshold from the shovel planes for regular geometry contacts.	Decimal | cuttingEdgeLength / 10.0.
ContactRegionVerticalLimit	Maximum vertical contact distance threshold from the shovel bottom plane for geometry contacts.	Decimal | cuttingEdgeLength / 10.0.
ParticleInclusionMultiplier	Radius multiplier for extending the inclusion bound with particle radius during post-excavation.	Decimal | 1.0.
CustomExcavationStiffnessMultiplier	Multiplier for the Young's modulus of user-defined contact materials during excavation.	Decimal | 1.0.

The advanced settings are stored in a `AdvancedShovelSettings` class, and can be obtained from the `Shovel` object via the following code:

```
const auto& advancedSettings = shovel->getAdvancedSettings();
```

23.16.3. Excavation Mode Settings

Each excavation plane have individual settings which allows the user to control aspects of the digging depending on the type of operation. Excavation planes can be disabled in full or only partially, for example to disallow it to create dynamic soil but still receive force feedback and perform compaction.

Property	Description	Type | Default value
Enable	Determines whether the excavation mode is enabled, generating dynamic mass and force feedback.	Boolean | True.
EnableCreateDynamicMass	Indicates whether the excavation mode should create dynamic mass.	Boolean | True.

Property	Description	Type | Default value
EnableForceFeedback	Indicates whether the excavation mode should generate force feedback from created aggregates.	Boolean | True.

The following snippet in c++ shows how to disable the left and right planes fully and the dynamic soil creation for the back plane. As can be seen, they are obtained with the `getExcavationSettings` method on the `Shovel` object and needs specification for which excavation plane you wish to configure.

```
shovel->getExcavationSettings( agxTerrain::Shovel::ExcavationMode::DEFORM_LEFT )->setEnable( false )
shovel->getExcavationSettings( agxTerrain::Shovel::ExcavationMode::DEFORM_RIGHT )->setEnable( false )
shovel->getExcavationSettings( agxTerrain::Shovel::ExcavationMode::DEFORM_BACK )->setEnableCreateDynamicMass( false )
```

23.16.3.1. Overriding Contact Material in Excavation Modes

The user can override the contact material between the shovel and the terrain for certain excavation modes. This can be desirable if different excavation modes should experience different resistance. This is done on a shovel basis and uses the following function:

```
/**
Explicitly set contact material properties in a shovel-aggregate contact for a specific
excavation mode for the shovel. This overrides the shovel-terrain contact
material properties that are used in the default case.
\param contactMaterial - The contact material to be set in the aggregate contact.
\param mode - The specified excavation mode that corresponds to the aggregate.
\return true if the contact material was successfully set, false otherwise.
*/
bool setShovelAggregateContactMaterial( agx::ContactMaterial* contactMaterial,
                                        Shovel::ExcavationMode mode = Shovel::ExcavationMode::PRIMARY );
```

The following example illustrates overriding the default contact material used in the back deformer with a custom contact material.

```
terrainMaterial = terrain->getMaterial( agxTerrain::Terrain::MaterialType::TERRAIN )
shovelTerrainContactMaterial = agx::ContactMaterial( shovelMaterial,
                                                     terrainMaterial )
shovelTerrainContactMaterial->setYoungsModulus( 1e5 )
shovelTerrainContactMaterial->setFrictionCoefficient( 0.2 )
shovelTerrainContactMaterial->setRestitution( 0.0 )
shovelTerrainContactMaterial->setAdhesion( 0.0, 0.0 )
simulation->add( shovelTerrainContactMaterial )
shovel->setShovelAggregateContactMaterial( shovelTerrainContactMaterial,
                                           Shovel::ExcavationMode::DEFORM_BACK )
```

Note

In order to reset the contact material to the default shovel-terrain contact material, simply use the method above with nullptr as the contactMaterial argument.

23.17. Angle of Repose

agxTerrain can model soil repose effects when changes in the terrain surface causes the soil angle between two adjacent elements to exceed the specified angle of repose of the material. The base angle of repose of a material, in the bank state unmodified by compaction, is given by adding the internal angle of friction and the delta repose angle:

$$\psi_r = \psi + \delta\psi_r$$

Where ψ_r is the effective repose angle of the material, ψ is the internal friction angle and $\delta\psi_r$ is the delta repose angle of the material.

Local soil compaction modifies the angle of repose for specified grid elements at a rate specified by the angleOfReposeCompactionRate.

23.18. Soil Compaction

Soil compaction determines how much soil mass that can exist inside a terrain 3D grid element. The default compaction value for a grid element is 1.0 which is defined as the bank state. As the compaction level changes, so does the local material properties in that grid element such as Young's Modulus and angle of repose. Every material has a swell factor which determines the fractional increase in soil volume during excavation which also decreases the compaction.



Fig. 23.12 Figure showing a wheel loader scene during excavation that illustrates terrain soils of different compaction levels through lightness rendering. Dark surface patches in the soil shows wheel tracks from compressed soil with a compaction level higher than the nominal compaction. The lighter areas show soils of looser compaction resulting from excavation or avalanching where it grows in volume according to the `swell factor` of the material. Soil of loose compaction offers less resistance and is easier to compress.

Default compaction for an initialized terrain is 1.0 in all terrain grid elements. Compaction can be set for all grid cells via the following function:

```
/**
Set a compaction for all current active solid grid elements in the Terrain object.
\note - the terrain grid elements all start with a the nominal compaction value of 1.0, of which all
user specified bulk parameters correspond to.
\param compaction - The compaction value to set to every voxel in the terrain (Default: 1.0)
\param replaceCompactedSoil - True if all voxels which already have a compaction value (compaction != 1.0)
                              should get overwritten.
*/
void setCompaction(agx::Real compaction, bool replaceCompactedSoil=true) const;
```

The TerrainGridControl class can be used to access and set the compaction value for single grid cells at the surface or at a 3D grid index.

```
float TerrainGridControl::getCompaction(const agx::Vec2i& terrainIndex, int z) const;
float TerrainGridControl::getSurfaceCompaction(const agx::Vec2i& terrainIndex) const;
void TerrainGridControl::setCompaction(const agx::Vec3i& terrainIndex, float compaction, bool compressOccupancy = false);
void TerrainGridControl::setSurfaceCompaction(const agx::Vec2i& terrainIndex, float compaction, bool compressOccupancy =
false, bool compressToBottom = false);
```

Compaction can also be set from the surface to a specified depth:

```
void TerrainGridControl::setSurfaceCompactionToDepth(const agx::Vec2i& terrainIndex, float compaction, float depth);
```

The soil is normally compacted via contacts from external geometries that generates surface terrain contacts where the contact area is calculated from the voxels that the geometry is overlapping. A uniform surface stress in the voxel contact patch is estimated from the contact force and the estimated area. The active terrain contacts that generate soil pressure can be extracted from the terrain and it's properties extracted:

```
auto terrainContacts = terrain->getTerrainContacts();
for( const auto& c : terrainContacts )
{
  auto geometry           = c.getGeometry();
  agx::Real surfaceStress = c.getSurfaceStress();
  agx::Real area          = c.getArea();
}
```

Pressure propagates down into the soil by approximating the surface in each voxel as a uniform plate. The pressure thus decays according to the formula:

$$\sigma_z^v(z) = \sigma_s[1 - (\frac{z}{\sqrt{A_s + z^2}})^3]$$

where $\sigma_z^v(z)$ is the local stress in depth z from the surface, A_s is the plate area and σ_s is the uniform surface stress.

The soil is compacted in a grid cell if it exceeds the local preconsolidation stress from earlier compactions or the starting preconsolidation stress. The level of compaction is determined by the specified `compression index` and the `preconsolidation stress` in the `Compaction Properties`. The compression index is defined as a logarithmic dependence between the change in compaction/void ratio from stress:

$$C_c = \frac{e_0 - e_1}{\ln\sigma / \sigma_0}$$

The new density in a grid cell is thus calculated via the following expression:

$$\rho = \rho_0[1 + \phi_0 C_c \ln\frac{\sigma}{\sigma_0}]$$

where the ϕ_0 is the volume fraction of the material, σ_0 the preconsolidation stress of the material, C_c the compression index of the material, σ the local stress and ρ_0 the density if the material at compaction 1.0 (i.e the bank state).

This is then used to calculate the compaction level at grid cell i:

$$c_i = \rho / \rho_0$$

One can apply a time relaxation to the rate of compaction changes in the terrain. It is governed by the following equation:

$$f_t = 1 - \exp[-t_c / \tau]$$

where τ is the time relaxation constant and t_c is the contact duration time. The timefactor is multiplied with the estimated density changes and approaches 1.0 as the contact time increases.

The parameters for the soil compaction algorithms can be found and adjusted in the `CompactionProperties` object.

23.18.1. Material changes from compaction

Some material properties change locally from compaction level.

23.18.1.1. Young's Modulus

The local Young's Modulus in a grid element changes according to the following relation:

$$E_{bulk} = E^0_{bulk}[1.0 \pm k_E[\frac{\rho}{\rho_0} - 1.0]^{n_e}]$$

where E_0 is the bulk elasticity in the bank state and k_e and n_e are hardening parameters that have default values of 1.0 and 0.5 respectively.

23.18.1.2. Dilatancy Angle

The level of compaction of the soil affects its dilatancy angle which in turn affects the internal friction of the material. The effective internal friction angle of the material thus becomes:

$$\phi_{eff} = \phi + \psi_{dilatancy}$$

Where ϕ_{eff} is the effective friction angle, ϕ the base internal friction angle of the material and $\psi_{dilatancy}$ is the dilatancy angle of the material. The effective internal friction angle of the soil is used to determine the soil wedge failure shapes and the resulting friction coefficient of the terrain <-> aggregate contacts. The dilatancy angle is given by:

$$\psi_{dilatancy} = c_{psi}(\varphi - \varphi_c)$$

Where φ is the local compaction and φ_c is the critical compaction, at which the soil switches between expanding or shrinking during shearing and c_{psi} is a constant that depends on particle shape.

> Note
>
> Note that the dilatancy angle of the material does not affect angle of repose. This has a separate compaction effect and is also affected by the delta repose of the material.

23.18.1.3. Swelling

When solid soil undergoes transformation to dynamic soil during excavation, the soil particle bulk volume expands according to the specified `Swell Factor` of the material. When the dynamic soil is transformed back to solid during merge it will have a compaction level according to:

$$c_i = 1.0 / S_f$$

where c_i is the local compaction at grid element i and S_f is the `Swell Factor`.

23.18.1.4. Angle of Repose

The local angle of repose changes with varying compaction levels by adjusting the allowed relative heights between two grid elements by using the following formulation:

$$
\begin{aligned}
h_r &= \tan\psi_r L \\
h_r &= h_r m \\
m &= 2^{M(max(0.0, c_i - (1.0)))}
\end{aligned}
$$

where M is a constant, ψ_r is the effective repose angle of the material, L is the length of a grid cell, c_i is the local compaction level at cell i. This effectively means that compaction levels above 1.0 increases the angle of repose for the material.

23.19. Dynamic Soil

Dynamic mass is created in the active zone intersection between the Shovel and the Terrain. It consists of both **fluid mass** and **soil particles**.

Soil particles are expressed in the form of 6-degrees of freedom (6-DOF) soil particles with hertzian contact dynamics, Coulomb friction and rolling resistance. The user can access and create soil particles via the soil simulation interface. The soil particles created from the shovel active zone intersection is simulated in the main simulation and can interact with other objects. By default, only the *particles see the shovels as kinematic objects*. This means that the particles themselves do not give force feedback to the shovels. The mass in the active zone, from both dynamic and solid mass, is used in the construction of the active zone aggregate that generates force feedback on the shovel through created contacts.

Note

The number of active soil particles depends on the amount of grid points touched by the dig volume which is a function of both blade size, dig depth, terrain resolution and dig length. A larger volume of influence means increased numbers of touched grid points and created particles which leads to an increase in computational time.

Soil particles are created when the active zone of the shovel touches new grid points. **Fluid mass** is generated if the shovel continues to remove mass from the same grid point and is then spread onto existing particles in the area, increasing their size. The fluid mass movement is integrated using a velocity field in the terrain grid generated from the shovels and the soil particles. If there are no particles that the fluid can merge to, it will merge back to the terrain.

Particles are also resized locally to cluster around a `nominal radius`. This is calculated from the resulting element size upon construction of the terrain object. It is also modified by the material swell factor and estimated packing ratio of the particles so that the total bulk volume of the particles created should correspond to the volume excavated, modified by the expected volume swelling. This radius can be extracted by using the following function:

```
agx::Real getParticleNominalRadius()
```

The radius that the particles cluster around can be altered using the `soil particle scaling factor`. What this does is that it alters the average radius of the soil particles, this can be used either to reduce the number of particles in the simulation to gain a potentially significant performance increase, or by increasing the number of soil particles to increase fidelity or better dynamic soil visuals. This scaling factor can be set and extracted using the following functions respectively:

```
void getProperties()->setSoilParticleSizeScaling( 1.5 )
agx::Real getProperties()->getSoilParticleSizeScaling()
```

The contact network between the soil particles is solved using an iterative Parallel-Projected-Gauss-Seidel Solver (PPGS) together with the rest of the system. It is enabled in the simulation as soon as a terrain instance is added. The user can also set the fidelity of the solver by adjusting the `setNumPPGSRestingIterations` which sets the number of iterations used in the PPGS solver. Increasing the iteration number typically results in the better solution and may be necessary when the particle counts increases in the simulation in order to get correct bulk behavior with stable particle piles that merge properly with the terrain. The default number is 25 iterations:

```
simulation->getSolver()->setNumPPGSRestingIterations( 25 );
simulation->getSolver()->setUseParallelPgs( false );
```

Warning

The PPGS solver currently disables island partitioning which means that performance will be lost if the scene contains multiple self-contained systems.

23.19.1. SoilSimulationInterface

Users can extract the `agxTerrain::SoilSimulationInterface` to access and create soil particle data in the terrain. The currently active dynamic soil particles in a simulation can be accessed via the following function:

```
auto soilSimulationInterface = terrain->getSoilSimulationInterface();
agx::Real totalMass{ 0.0 };
agx::Real averageRadius{ 0.0 };
auto soilParticles = soilSimulationInterface->getSoilParticles();
for ( auto& particle : soilParticles )
{
```

```
  totalMass += particle->getMass();
  averageRadius += particle->getRadius();
  agx::Vec3 position = particle->getPosition();
  agx::Vec3 velocity = particle->getVelocity();
}
averageRadius /= soilParticles.size();
```

Note

Do note that changing either *mass*, *material* or *radius* will change the mass of the particle and thus violate mass conservation.

Warning

It is not recommended to change particle properties such as *mass* and *material* as those are derived from the Terrain instance.

Individual soil particles can be created and removed manually:

```
auto soilSimulationInterface = terrain->getSoilSimulationInterface();
agx::Real radius = terrain->getParticleNominalRadius();
agx::Vec3 position{ 0.0, 0.0, 1.0 };
agx::Vec3 velocity{ 0.0, 0.0, 0.0 };
auto soilParticle = soilSimulationInterface->createSoilParticle(radius, position, velocity);
soilParticle->setPosition( ... );
soilParticle->setVelocity( ... );
...
soilSimulationInterface->removeSoilParticle( soilParticle );
```

Before doing any operations on the particles in the terrain, it can be useful to check if the particle is valid or not already removed:

```
auto soilParticles = soilSimulationInterface->getSoilParticles();
for ( auto& particle : soilParticles )
{
  if ( soilSimulationInterface.isValid( p )
  {
    ...
  }
}
```

Soil particle collisions can be controlled via the collision group system in `agxCollide::Space` and added to the created soil particles via the soil interface:

```
// Create a collision group and add it to all particles
agx::Name collisionGroup { "myCollisionGroup" };
soilSimulationInterface->addCollisionGroup( collisionGroup );

/**
Disable collisions between the objects that have the specified group. This will prevent particles
from colliding with each other.
*/
simulation->getSpace()->setEnablePair( collisionGroup, collisionGroup, false );
```

The current active bulk volume of soil particles can be extracted by using the following function:

```
/**
\return soil particle bulk volume (m3) active in the simulation.
\note - the bulk volume is the specific volume of all particles divided with the
       estimated particle packing fraction (0.67)
*/
agx::Real calculateSoilParticleBulkVolume() const;
```

The internal `GranularBodySystem` used for simulating the soil particles can be extracted and used with other objects, such as a `agx::ParticleEmittter` to create masses of soil particles.

```
agx::Physics::GranularBodySystem* getGranularBodySystem() const;
```

Warning

When using an emitter with the GranularBodySystem, the emitter particle material should be the same as the particle material of the terrain that the emitted particles should belong to! The particle material is used by different terrains in the simulation to determine which particle belongs to which terrain instance.

23.19.2. Custom soil failure volume

In certain cases it is desirable to be able to trigger soil conversion into dynamic soil particles manually without the terrain interacting with a shovel. This can be achieved by creating a soil failure volume and providing it to the terrain. The following code showcases how to trigger soil failure in a 1x1x1 box:

```
agxCollide::Box failureVolume = agxCollide::Box(0.5,0.5,0.5);
terrain->convertToDynamicMassInShape(failureVolume);
```

23.20. Excavation Model

We refer to the published academic paper [1] for a more in-depth explanation of the mathematical model for agxTerrain. This section will only provide a brief overview of the different components of the model.

During setup of a shovel, deformation planes are created from the geometry of the shovel. As the shovel interacts with the terrain, active zones are generated in the deformation planes. The active zones consists of discretized soil wedges that are created from soil mechanics theory.

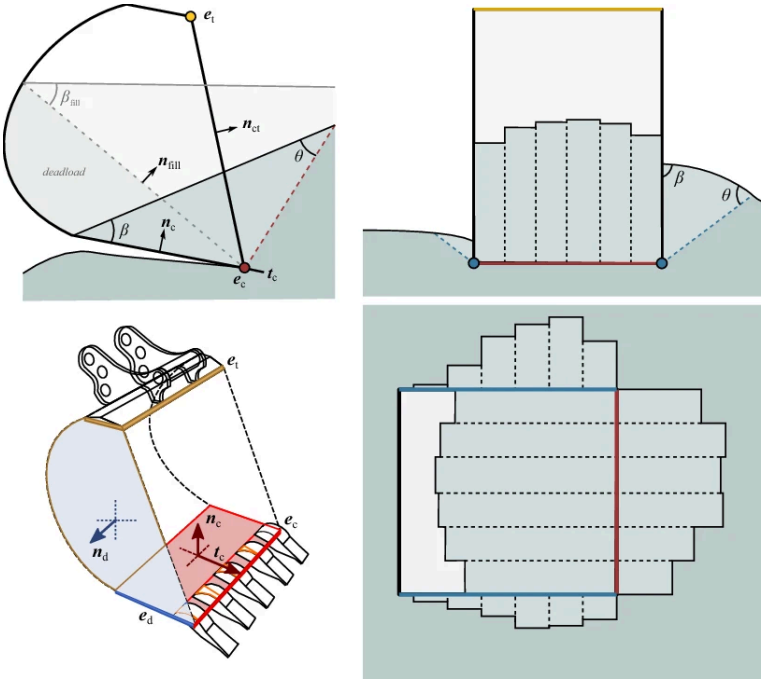


Fig. 23.13 The left images illustrate a digging tool with a cutting edge e $_c$ (**red line**), separating plate n $_c$ (**red face**), deformer edge e $_d$ (**blue line**), deformer face n $_d$ (**blue face**) and top edge e $_t$ (**yellow line**). The inner shape is indicated with the dashed lines. The normal n $_{fill}$ of the secondary separation plate range from n $_c$ to n $_{ct}$ depending on the fill ratio of the inner shape. The three active zones, discretized by vertical wedges, is illustrated from the rear (**upper right**) and from above (**lower right**).

Four different excavation planes are automatically constructed from the shovel edges, each corresponding to a mode of deformation:

Primary	This is the primary excavation mode of the shovel that is driven by the `separation plate`. Used during *excavation* operations. Defined by the `top edge`, `cutting edge` and `cutting direction`.
Deformer Back	This is the deformer plane defined by the underside of the shovel. Used in *grading* and *push* operations.
Deformer Left	This is the deformer plane to the left side of the shovel. Used in *grading* and *push* operations.
Deformer Right	This is the deformer plane to the right side of the shovel. Used in *grading* and *push* operations.

These individual planes can be configured.

23.20.1. Mass deformation model

As the shovel moves through the terrain it converts solid mass into dynamic mass in the active zone, consisting and internal fluid and particle mass:

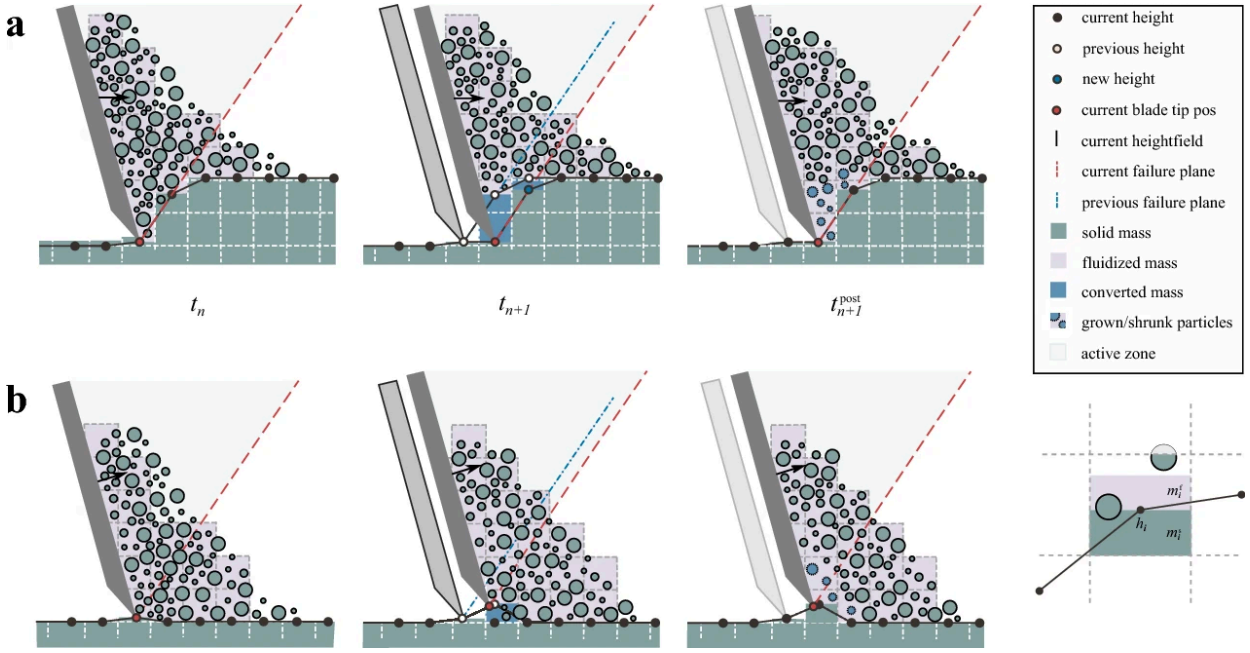


Fig. 23.14 As a blade and its active zone moves into the terrain **a**, new solid voxels are resolved into particles or fluidized mass that form the aggregate body. The voxel height value, corresponding to the solid occupancy, is found by projection to the failure plane of the active zone. When the blade is raised **b**, the gap is filled by solid mass converted from fluidized or particle mass in the vicinity of the edge.

23.20.2. Excavation Resistance Model

Force feedback in the separation and deformation mode is generated by aggregate rigid bodies created from dynamic and solid mass in the shovel active zones. Contacts are created both between the shovel and the aggregate and the aggregate and terrain. The resulting forces depend on the contact materials properties derived from the `agxTerrain::TerrainMaterial` bulk properties and also the contact material between the shovel and the terrain that is used for the shovel-aggregate contacts.

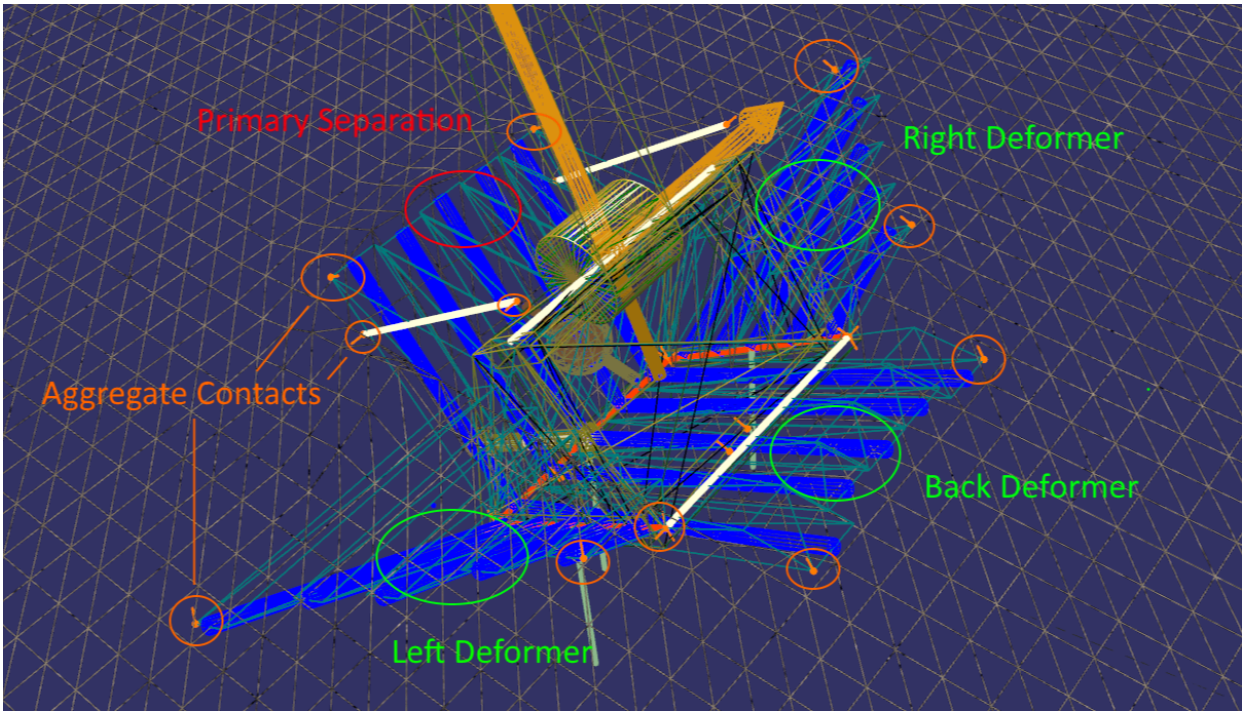


Fig. 23.15 Figure showing the aggregate formation during excavation. Primary separation resistance is denoted in red and the deformer zones are denoted in green. The resulting aggregate contacts for aggregate <-> terrain and aggregate <-> shovel are denoted in orange. The blue lines denote the direction of the soil wedges that are created in the terrain via calculation using the Fundamental Earth Moving model (FFE) and the material bulk properties. These soil wedge shapes are the basis for the failure zone that ultimately form the soil aggregates in the separation and deformer zones.

23.20.3. Penetration Resistance Model

Pure penetration occur when a thin tool moves straight into the terrain, displacing only the soil necessary for fitting the tool. Buckets are often penetrated deep into the soil before separation (cutting) starts. The penetration resistance experienced by a tool consists of two main parts: from the teeth (or edge)

and from the primary separating plate. The resistance from the separating plate contacting the soil is simply modeled by a frictional force proportional to the passive earth pressure in the soil. The pressure on the teeth during penetration can be modeled by using the cavity expansion model. The penetration force acts in the cutting direction of the Shovel.

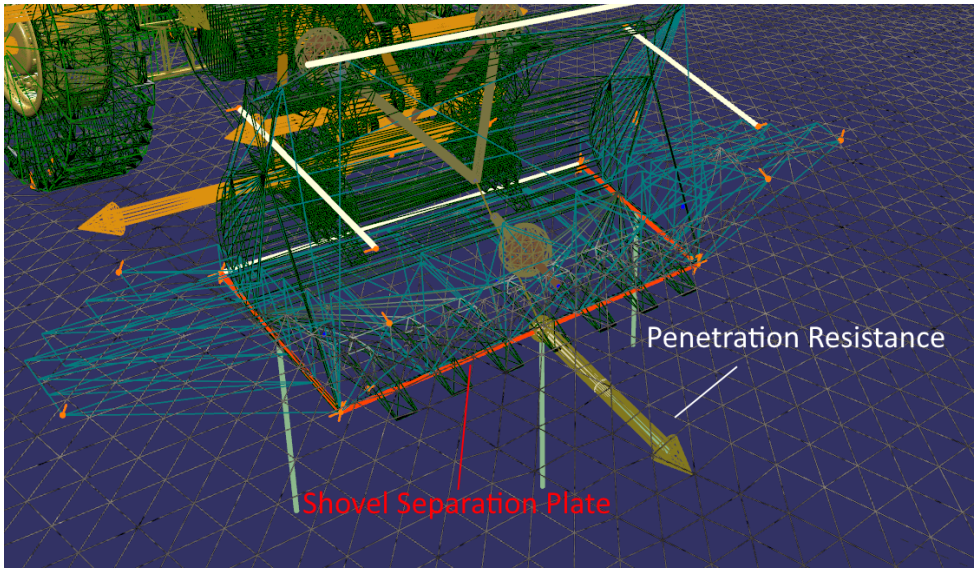


Fig. 23.16 Figure illustrating the shovel <-> terrain interaction during a wheel loader digging process. The penetration resistance works in the cutting direction parallel to the separation plate of the bucket and is expressed as a prismatic velocity constraint with force limits according to the penetration model theory.

23.21. Performance notes

The performance of agxTerrain depends on the number of dynamic soil particles in the simulation. The number of active soil particles depends on the amount of grid points touched by the dig volume, which in turn is a function of both blade size, dig depth, terrain resolution and dig length. A larger volume of influence means increased numbers of grid points and more particles, and thus an increase in computational time.

The number of solver iterations set via `setSolverIterations` via `SoilSimulationInterface` also determines the overall performance. Increasing the solver iterations is needed when particle count increases in order to get stable soil piles. A low iterations count relative to the amount of particles will results in an error that introduces numerical vibrational noise in the contact forces, causing the particle piles to “melt”. A typical cue of when to increase solver iterations is when the particles are unable to merge with the terrain due to this numerical noise.

Do note that the material parametrization of the terrain, with the exception of terrain resolutions, does not overtly affect the performance. This means that soil behavior should be achieved without compromising performance.

References

[1] (1,2)

Servin, M., Berglund, T. & Nystedt, S. A multiscale model of terrain dynamics for real-time earthmoving simulation. Adv. Model. and Simul. in Eng. Sci. 8, 11 (2021). link

23.22. FAQ

23.22.1. Questions

1. Why do I get an error of “bad soil wedges” during excavation?
2. Why is my shovel not digging?
3. How can I reduce the amount of soil particles in the simulation without lowering the terrain resolution?
4. How can I modify the digging resistance?
5. How do I measure the amount of material in a bucket?
6. How can I calculate the amount of transported/excavated material?
7. How can I use the terrain on the bed of a truck?
8. How can I simulate mud so that I sink down with the vehicle?
9. How can I “reset” the terrain quickly to its initial state?
10. How do I adjust the static angle of repose in my simulations?

11. How can I simulate dry or wet sand?

23.22.2. Answers

Why do I get an error of “bad soil wedges” during excavation?

This error occurs when the soil wedge geometries are improperly constructed, due to a faulty shovel setup. The problem is typically associated with misaligned vectors, specifically in the top edge or cutting edge directions. To resolve the error, consider adjusting the orientations of these vectors to ensure proper alignment.

Why is my shovel not digging?

There are several reasons why your shovel might not be digging. First, take a look at the Shovel setup section here in the User Manual. The vectors specified in the schematic can be seen in debug mode, which is accessed by pressing `g` in `agxViewer`. If that does not help, try to resolve the following questions:

- Is the cutting direction vector specified in the correct direction?
- Is the cutting edge vector actually aligned with the shovel geometry?
- Is the shovel added to the terrain you want it to dig in?
- Is the terrain added to the simulation?

There are also several tutorials where shovels are setup from simpler geometric shapes in the `data\python\tutorials\agxTerrain` folder.

How can I reduce the amount of soil particles in the simulation without lowering the terrain resolution?

The amount of particles in the simulation can be adjusted with the following method:

```
auto* properties = terrain->getProperties();
// Increase the particle radius by a factor of two.
properties->setSoilParticleSizeScaling(2.0);
```

This scaling factor modifies the nominal radius of the created soil particles with a scalar factor. Increasing the nominal radius will adjust the soil mass resizing algorithm to increase the particle size and correspondingly reduce the amount of particles in the simulation.

How can I modify the digging resistance?

The digging resistance encountered during excavation is a combination of several forces. The primary approach for modifying digging resistance includes fine-tuning the contact material properties between the shovel and terrain. This affects both penetration, separation and deformation resistance. Additionally, tweaking the terrain material properties, such as soil density and friction angle, can change contact forces between aggregate and terrain.

The effective terrain material properties used in the force calculations are also affected by the local compaction in the terrain, which also can be adjusted manually. There is also a PenetrationForceScaling parameter which can be used to adjust the penetration resistance.

The User Manual provides a dedicated chapter on the Calibration of excavation resistance, and there is also a chapter on soil compaction, which we urge you to read in case you are interested in modifying the digging resistance.

How do I measure the amount of material in a bucket?

One way to do this is to measure the total weight of the soil aggregates in the shovel.

There are two kinds of aggregates that are created:

- Inner body = the inner body inside the confines of the shovel.
- Wedge bodies = the soil wedge bodies that are created during excavation.

To get the amount of material inside the confines of the bucket, use the following method:

```
agx::Real innerBodyMass = shovel->getSoilParticleAggregate()->getMass();
```

To get the amount of material in the wedge bodies, use the following method:

```
agx::Real  wedgeBodyMass = shovel->getSoilParticleAggregate()->getTotalAggregateMass() - innerBodyMass;
```

Of course, you can use the following method to get the combined mass for all aggregates formed in and on the shovel.

```
shovel->getSoilParticleAggregate()->getTotalAggregateMass();
```

How can I calculate the amount of transported/excavated material?

This is quite complicated to achieve but possible. During each dig cycle, a moment where the shovel contains the amount that is excavated/transported needs to be identified.

Preferably this is while the shovel is raised above the ground. In that moment we measure the amount of material in the bucket, and then we dump the contents of the shovel somewhere. The amount measured during the repeated dig cycles then needs to be summed up in the end, to find out how much material has been transported/excavated during the simulation.

Note that there is a method on the terrain which can be called in every `POST`-step to get the last excavated volume in `[m^3]`:

```
agx::Real lastExcavatedVolume = terrain->getLastExcavatedVolume();
```

This is the volume which is transformed into dynamic mass in each timestep and not that which is captured by the shovel.

How can I use the terrain on the bed of a truck?

This is achieved by adding the terrain geometry to the truck's rigid body:

```
// Given that truck is an agx::RigidBody
truck->add(bedTerrain->getGeometry());
```

The `bedTerrain` must be constructed to fit onto the bed of the truck. To allow tipping, failure zones also have to be constructed which trigger the conversion of the solid terrain into dynamic mass when the bed is tipped to a certain angle. How this is done is shown in the tutorial `tutorial_soil_failure_volume.agxPy`.

How can I simulate mud so that I sink down with the vehicle?

You can simulate the sinking into the terrain by using soil compaction. How much the vehicle sinks is defined by the terrain materials compaction properties, which are accessed via the following method:

```
auto compactionProperties = terrainMaterial->getCompactionProperties()
```

Where the terrain material is in the desired area. The tutorial `tutorial_compressed_soil.agxPy` shows some basics of manipulating the compaction properties of a terrain material to allow sinkage of external objects.

How can I "reset" the terrain quickly to its initial state?

For a single terrain, this can be achieved by saving the terrain heights in the initial state and then later setting the heights again. Use the following methods:

```
agx::RealVector initialHeights = terrain->getResizedHeightField(1, 1);
//... later in the simulation, use the initialHeights to reset the height field:
terrain->setHeights(initialHeights);
```

The `setHeights` method also takes a `resetCompaction` parameter, which by default resets the compaction values in the terrain. This means if you have changed the compaction values manually with the `terrain->setCompaction(agx::Real compaction)` call, this needs to be repeated. Similarly, if you have added any additional terrain material configurations, the information of which terrain material existed in the excavated terrain may have been lost and need to be reapplied again. Also, if you have performed some sort of excavation operation you might also need to remove lingering particles. This can be achieved via the granular system:

```
auto soilParticles = terrain->getSoilSimulationInterface()->getSoilParticles();
for(auto sp : soilParticles)
  terrain->getSoilSimulationInterface()->removeSoilParticle(sp);
```

For a paged terrain, when using a `TerrainPager`, resetting can be achieved by calling the `terrainPager->reset()` method. This method will reset all terrain tiles to their initial states, as set by the `TerrainDataSource`, including the currently active tiles.

How do I adjust the static angle of repose in my simulations?

The angle of repose is controlled via the following terrain material parameters:

```
// The friction angle(radians) has 1:1 relationship with the angle of repose of the material
bulkProperties->setFrictionAngle(0.7);

// The delta repose angle (radians) that is applied on the friction angle when determining the angle of repose of a
material.
bulkProperties->setDeltaReposeAngle(0.15);
```

You can adjust these to get the angle of repose you want. It is also worth noting that soil compaction also modifies the local angle of repose.

How can I simulate dry or wet sand?

You can simulate both dry and wet sand by adjusting specific terrain material parameters. Two key parameters to focus on are bulk density and bulk cohesion, and the internal friction angle plays a role as well. Dry sand typically has a lower bulk density than wet sand, reflecting the absence of added moisture. On the other hand, simulating wet sand involves increasing the bulk density, as water contributes to the overall mass.

In granular materials with particles exceeding 0.1 mm in diameter, repulsive forces usually dominate. However, as moisture content rises, bulk cohesion may increase due to capillary forces and chemical reactions at particle contact points. It's important to note that excessive water content can lead to pore pressure, diminishing attractive contact forces and thereby reducing internal friction, a parameter influenced by the internal friction angle. Consequently, bulk cohesion may decrease as capillary forces diminish.

In summary, manipulating the terrain material properties of bulk density, bulk cohesion, and the internal friction angle allows for the accurate simulation of both dry and wet sand conditions.

24. Material calibration – bulk materials

In AGX Dynamics, AGX Terrain, and Momentum Granular, *bulk materials* consisting of particles with certain particle parameters can be created. (In agxTerrain, only the dynamic mass of the bulk material consists of particles.) Examples of bulk materials are soils, sands, and gravels, and examples of particle parametes are friction and restitution. A bulk material has certain bulk properties, such as the static angle of repose `theta`, bulk cohesion `c`, and friction angle `phi`, which determine the large scale dynamics of the material. Now, in AGX programs, the particle parameters are set explicitly and yield a particular behavior of the bulk material, i.e. a particular static angle of repose, bulk cohesion, and friction angle will result in one behavior of the bulk material. However, the bulk properties resulting from a certain set of particle parameters are unknown beforehand. Therefore, material calibration tests — the static angle of repose test and the triaxial test — are implemented in AGX Dynamics which determine the bulk properties resulting from a certain set of particle parameters. The static angle of repose test determines the static angle of repose `theta`, and the triaxial test determines the bulk cohesion `c`, and the friction angle `phi`. Thus, the tests offer a *mapping* between particle parameters and bulk properties.

The point of the tests is to identify the particle parameters that yield the bulk properties of a particular bulk material that the user wants to model (denoted the “desired material”). The process of seeking the particle parameters that yield desired bulk properties is denoted *material calibration*, and is carried out by using one or both of the tests. If the test with a particular set of particle parameters yield the same bulk property or properties as those of the desired material, the calibration is complete, since then, by using that set of particle parameters, desired bulk properties are yielded. If the result of a test is bulk properties that do not match those of the desired material, the test needs to be repeated with different particle parameters until desired bulk properties are obtained.

This section describes these bulk properties, how the material calibration tests work, and how to carry out the tests and interpret the results.

24.1. Bulk properties

As previously noted, bulk materials such as soils, sand, and gravels, have particular bulk properties, for example static angle of repose `theta`, bulk cohesion `c`, and friction angle `phi`. These properties govern the large scale dynamics of the bulk material and can be used to predict for example if a soil will fail under certain conditions, such as if a house is placed on a hill. These properties are independent of simulation implementation and are observed in nature. The static angle of repose is the greatest angle between the side of a static pile and the horizontal plane at which the pile does not slump, see see Fig. 24.1. Typically, a bulk material with high friction has a greater static angle of repose than a bulk material with low friction.



Fig. 24.1 The static angle of repose `theta`, illustrated for a pile of sand. The static angle of repose of sand is observed in the lower bulb of an hour glass when the top bulb is empty.

For completeness, we also note that there is also the property *dynamic angle of repose*. For this property, the angle itself is measured essentially as the static angle of repose, but when the material is in motion. This property is for example observed when sand tumbles in a cylindrical mixer. The dynamic angle of repose will not be treated further here.

The shear strength of a material is a function of the friction angle `phi` (also known and the angle of internal friction), and the bulk cohesion `c` of the material. These two properties are included in the straight line *failure envelope equation*, given by

```
tau = c + tan(phi) * sigma,
```

where `tau` is shear stress, and `sigma` is normal stress, see Fig. 24.2.

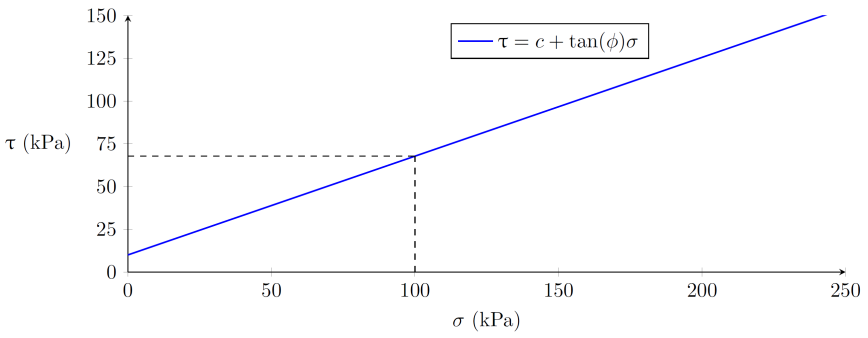


Fig. 24.2 The failure envelope equation, with bulk cohesion `c` being the intercept, and the tangent of the internal friction `phi` being the slope. In this particular case, the bulk material has `c=10` kPa and `phi=30` degrees.

The failure envelope equation describes, for a given normal stress, the amount of shear stress a material can withstand without shearing, i.e. failing. Both a higher `phi` and a higher `c` implies a greater shear strength. For example, in Fig. 24.2, we note that for this particular material, at a normal stress of 100 kPa, the material shears at a shear stress of 67.7 kPa. At a higher (lower) normal stress, the material shears at a higher (lower) shear stress. That an increased normal stress implies an increased ability to withstand shear stress is exemplified by considering an unopened coffee package under vacuum (coffee, like sand, is a bulk material). In the unopened package, the coffee is in a packed state, with relatively high normal stress, and it is difficult to twist the package, i.e. shear the coffee sample. However, by opening the package (or even by punching a small hole in it with a needle), the normal stress drops significantly, and the package is easily twisted. Shearing is depicted in Fig. 24.3.

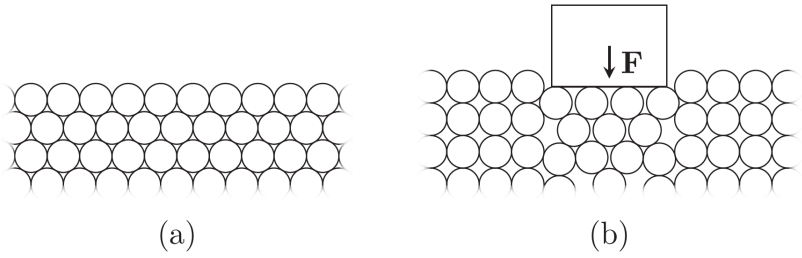


Fig. 24.3 (a) A bulk material, for example sand, represented by spherical particles. The material is in a packed state. (b) A load is applied to the surface of the material. The shear stress is greater than the shear strength so shearing occurs during which particle layers slide over each other: the material fails. Typically, when a bulk material fails, the volume is increased.

The friction angle `phi` is due to contacts and is responsible for an increased shear strength at an increased normal stress: at a higher normal stress, the particles are brought closer, increasing the contacting areas between them and so also the particle-particle frictional forces. The bulk cohesion `c` is due to attractive cohesive forces and is independent of the normal stress. We note in Fig. 24.2 that an increased `c` increases the tolerable shear stress with the same amount for all values of the normal stress. Usually, a wet material, for example wet sand, has a higher bulk cohesion than a dry material, as water molecules tend to attract one another—it is easier to build a sand castle with wet or damp sand than with dry sand!

24.2. Selection of particle parameters

There is no clear cut strategy in the selection of particle parameters for the material calibration tests. The process is a mix of educated guesses and trial and error. After a completed test, the results give an indication on how to proceed. For example, in the static angle of repose test, if the resulting static angle of repose is lower than the corresponding value of your desired bulk material, a suitable course of action is to repeat the test with a higher friction and/or higher rolling resistance. In the triaxial test, if the resulting *bulk* cohesion is lower than your desired bulk cohesion, a repeated test with a higher *particle* cohesion is advisable. One approach, applicable for all tests, is the following: create a (large) number of different particle materials with different particle parameters, place them in a dedicated directory, and then run the test of interest on all of them. This yields several results that you can inspect and compare which gives you a hint on how to proceed.

Now, the particle parameter to bulk properties mappings are not unique. While the bulk properties of a particular bulk material are constant, more than one set of particle parameters can yield one or several of these properties.

24.3. The static angle of repose test

The static angle of repose test determines the static angle of repose (described in Bulk properties) and comes in many variants, using a tilted box, fixed funnel, or a hollow cylinder. The test is carried out in real life or with a computer simulation. In this section we describe the test implemented in AGX Dynamics using a hollow cylinder that is lifted, so the full cumbersome name of the test is the “static angle of repose test using the hollow cylinder lifting method”. In Test description we describe how the test works, and in User guide (detailed) we describe in detail how to carry out the test with step by step instructions and how to interpret the results. In User guide (brief) we offer a very short user guide that can be used as a cheat sheet for readers familiar with the test. It is recommended to first read the Test description as this will make the User guide easier to follow and aid in the selection of test settings.

24.3.1. Test description

This section chiefly describes the theory and mechanics of the test, and only indirectly describes the workflow for the execution of the test. For a detailed description of the workflow, see User guide (detailed).

24.3.1.1. Setup

In the first part of the test setup particle parameters such as friction and rolling resistance are specified. These parameters specify particle-particle interactions and will be applied to the particles that represent the bulk material. Wall parameters — specifying particle-wall interactions — do not be set explicitly as they are automatically calculated from particle parameters. How these are set will be elaborated further. Particle parameters are specified in a particle material file.

In the second part of the test setup simulation settings such as timestep and particle radii are specified in a simulation settings file.

24.3.1.2. The test

With particle parameters and simulation settings specified, the test is ready to be executed.

In the 1st phase of the test, “particle emission”, see Fig. 24.4, an emitter (yellow cylinder) moves slowly upwards and emits particles. The particles land on a static base (violet cylinder). The particles are surrounded by an initially static tall hollow cylinder (light blue), preventing them from falling of the base. Below the base, there is a static particle sink (blue cylinder of large radius), .

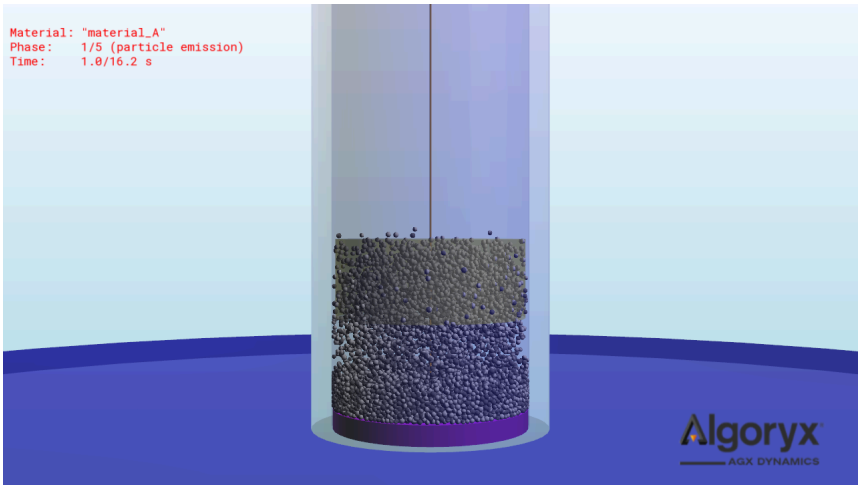


Fig. 24.4 The test in the 1st phase: “particle emission”. The selected particle material file is `material_A.json`.

In the 2nd phase, “particle sample relaxation”, see Fig. 24.5, the particle sample is allowed to rest for a short duration, allowing particle velocities to drop to essentially zero.

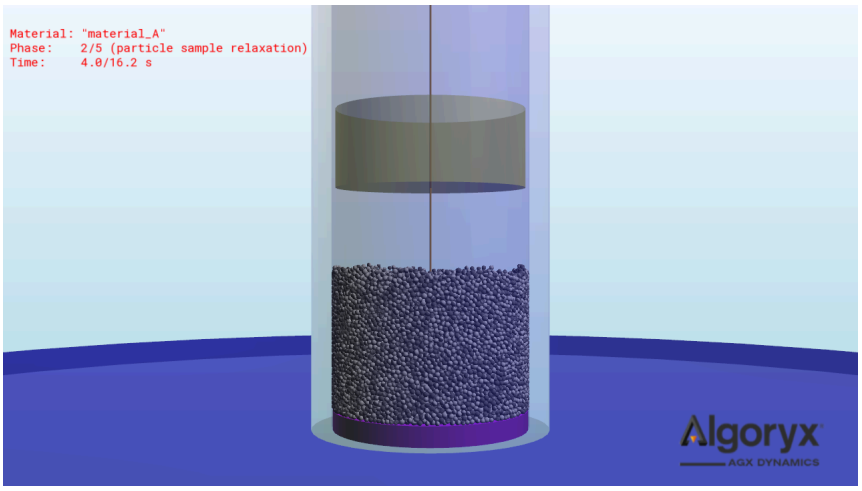


Fig. 24.5 The test in the 2st phase: “particle sample relaxation”.

In the 3rd phase, “cylinder lifting”, see Fig. 24.6, the hollow cylinder is slowly lifted, allowing particles to fall off the base and towards the sink where they are removed. By removing particles that fall of the base, the computational time is decreased. Now, the static angle of repose is independent of the height of the particle pile. Therefore, the top surface of the base can be considered to be a slice of the pile. Thus, the base-particle interactions are set to be the same as the particle-particle interactions. However, we would like the hollow cylinder to interact as little as possible with the particle sample, so we set zero friction in the hollow cylinder-particles interactions.

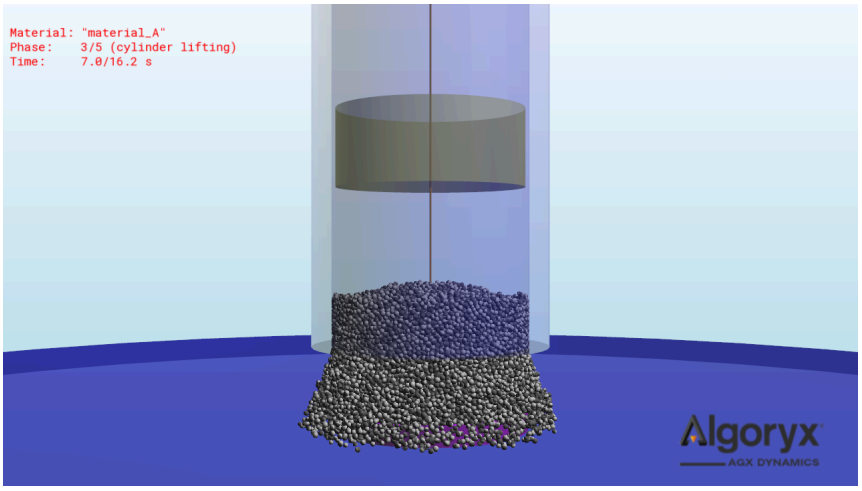


Fig. 24.6 The test in the 3rd phase: “cylinder lifting”.

In the 4th phase, “particle pile relaxation”, Fig. 24.7, the particle sample is allowed to rest for a short duration, again allowing particle velocities to drop to essentially zero. At the end of this phase, a static granular pile has formed with some static angle of repose.

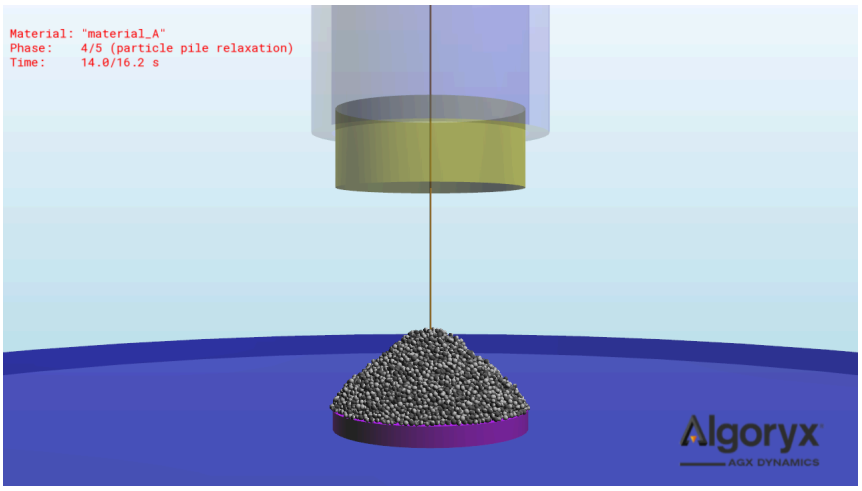


Fig. 24.7 The test in the 4th phase: “particle pile relaxation”.

24.3.1.3. The analysis

In the 5th phase, “particle pile analysis”, the particle pile is analyzed by an algorithm by which angle of repose results are obtained. The surface of the base is parallel to the xy-plane, z is the vertical axis, $r = \sqrt{x^2 + y^2}$ is the polar radial coordinate, and $\psi = \text{atan}(y / x), x > 0$ is the azimuthal angle.

For the analysis, n_{slices} slices are introduced such that slice $i = 1, 2, \ldots, n_{\text{slices}}$ is the plane $\psi_i = (i - 1)(360^\circ / n_{\text{slices}})$. Each slice has n_{cells} associated cylindrical cells of radius R_{cell} and infinite height with axes parallel to the z-axis and in the plane of the slice such that cell $j = 1, 2, \ldots, n_{\text{cells}}$ is centered at $r_j = jR_b / (n_{\text{slices}} + 1)$, where R_b is the radius of the base. R_{cell} is larger than the largest particle radius of the particle system. These elements are illustrated in Fig. 24.8.

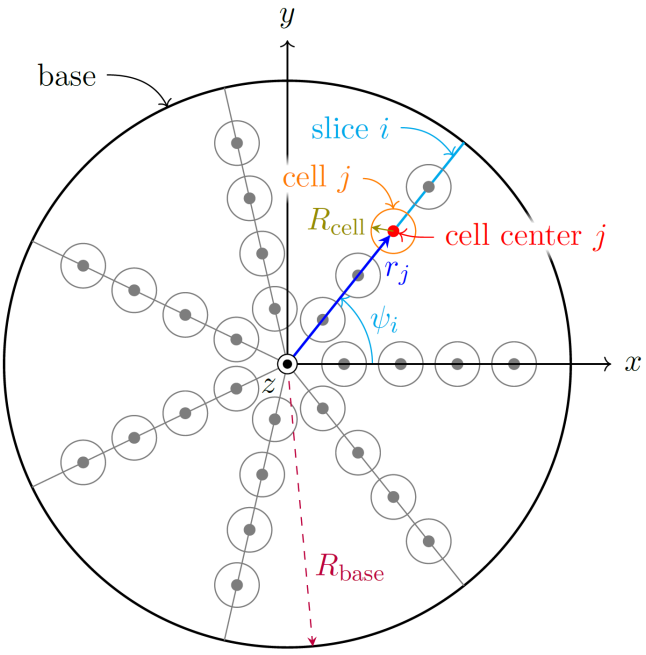


Fig. 24.8 A top view (looking down the z-axis) of the test, with emphasis on the analysis elements. We recall that the base is the cylinder of radius R_{base} upon which the particles rest. Slice i, making an angle ψ_i with the x-axis, is highlighted. Also highlighted is cell j of slice i and its associated quantities: r_j is the distance from the origin to the cell center, and R_{cell} is the radius of the cell. In this figure, there are 7 slices, and 4 cells, yielding a total of $7 \times 4 = 28$ cells.

For each cell, all particles within the cell are considered by the algorithm (for cell j of slice i, the particles considered are within the region indicated by the orange circle in Fig. 24.8), and the particle with the largest z-coordinate is selected for that cell and its r- and z-coordinates are stored. This particle is also colored, giving a three dimensional presentation of the analysis algorithm, see Fig. 24.9.

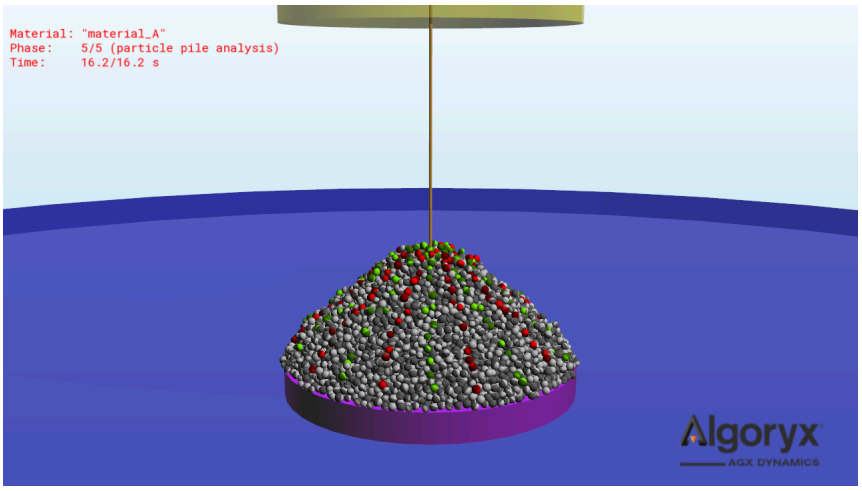


Fig. 24.9 The test in the 5th phase: “particle pile analysis”. For a particular slice, the associated particles with the largest z-coordinates are colored by a single color, red or green. In this case, the pile is divided into 20 slices, and each slice is divided into 15 cells. Thus, we have 20 strings of colored particles, with 15 particles in each string.

For each slice, a linear least-squares regression is applied to the associated sets of the z- and r-coordinates of the particles with the largest z-coordinates, yielding a straight line with a slope such that the arctangent of the slope approximates the static angle of repose for that slice. Thus, for n_{slices} slices, we have n_{slices} approximations of the static angle of repose. From these n_{slices} approximations, we obtain the average static angle of repose, the standard deviation of the average, the minimum angle, and the maximum angle. The test and analysis is complete.

24.3.2. User guide (brief)

The static angle of repose test is performed by executing the following steps:

1. Find the static angle of repose `theta` of your desired bulk material
2. Make sure that required python modules are installed (see `static_angle_of_repose_test/requirements.txt`)
3. Create your particle material(s) (see e.g. `particle_soil/json/sand_1.json`)
4. Set up the simulation settings file (see `static_angle_of_repose_test/SimConfDefault.py`)
5. Open an AGX terminal at `/static_angle_of_repose_test/` and run the static angle of repose test (`runStaticAngleOfReposeTest.py`)
6. Inspect the test result(s) (stored in `static_angle_of_repose_test/results/`). If the static angle of repose `theta` matches the value of the desired bulk material, the material calibration is complete. If not, rerun the static angle of repose test with different particle parameters until desired static angle of repose is yielded.

24.3.3. User guide (detailed)

24.3.3.1. Find the static angle of repose `theta` of your desired bulk material

The static angle of repose `theta` of a particular material can be found by consulting a table of soil or granular materials, if the material is not too uncommon. If tables are inaccessible or do not exist, physical experiments on the real material need to be carried out to measure the property. If such experiments are infeasible, the property needs to be judiciously estimated.

24.3.3.2. Locate the static angle of repose test

Your AGX Dynamics installation path should be on the form `C:/.../AGX-<version>`. Then, the static angle of repose test is located at `AGX-<version>/data/python/material_calibration/particle_soil/static_angle_of_repose_test`.

24.3.3.3. Install python modules

The test is written in python and the modules required for the static angle of repose test are listed in `static_angle_of_repose_test/requirements.txt`. Make sure that they are installed for the python version specified to be used with AGX Dynamics. To install the modules, use e.g. `python -m pip install -r requirements.txt`.

24.3.3.4. Create your particle material(s)

Browse to `/particle_soil/json/`. This is the directory in which particle materials are stored and should be created. The particle sample in the static angle of repose test will have particle parameters of a file from this directory or a subdirectory of it, say `/particle_soil/json/my_materials/`. More specifically, these parameters set particle-particle interactions. (Particle-wall interactions do not need to be specified.) For convenience, copy one of the existing files, for example `sand_1.json`, rename the file to, say, `material_A.json` and enter desired particle parameters and save the file. An example is shown in Code snippet 1.

```
{
    "soilName" : "MATERIAL_A",
    "ParticleProperties" : {
      "particleCohesiveOverlapRatio" : 0.025,
      "particleCohesion" : 0,
      "particleDamping" : 4.5,
      "particleFriction" : 0.45,
      "particleRestitution" : 0.1,
      "particleRollingResistance" : 0.25,
      "particleSpecificDensity" : 2700,
      "particleYoungsModulus" : 1e9
    }
}
```

Code Snippet 1 - `material_A.json`

Notes on how to select particle parameters are given in Selection of particle parameters.

The material in Code snippet 1 is used for the simulation demonstrated in Test description: the test.

24.3.3.5. Set up the simulation settings file

The simulation settings file with default settings is `static_angle_of_repose_test/SimConfDefault.py`. In Code snippet 2 an abbreviated version of this file is given.

```
class SimConf():
    def __init__(self, particleSpecificationPath=defaultSpecificationPath, particleSpecification=""):

        ### Simulation parameters ###
        self.timeStep = 1/1000
        self.numPPGSRestingIterations = 500
        self.agxOnly = False
        ...

        ### Particle material ###
        self.particleRadius = 0.015
        self.particleRadiusMin = 0.9 * self.particleRadius
        self.particleRadiusMax = 1.1 * self.particleRadius
        self.particleWeight = 0.4
        self.particleWeightMin = 0.3
        self.particleWeightMax = 0.3
        ...

        ### Static angle of repose test parameters ###
        self.baseRadius = 0.6
        self.numSlices = 20
        self.numCells = 15
        ...
```

Code Snippet 2 - An abbreviated version of `SimConfDefault.json`

In this file simulation settings such as time step, particle radii, etc. are specified. If you desire to alter these simulation settings, it is recommended to save them in a new file, say `static_angle_of_repose_test/SimConfMySettings.py`. Usually, the default values can be used for most settings. The settings of most interest are shown in Code snippet 2.

The time step (`self.timeStep`) and the number of iterations for the iterative solver (`self.numPPGSRestingIterations`) have an impact on the static angle of repose. Both a larger time step and a smaller number of iterations usually result in a smaller static angle of repose. It is recommended, if possible, to use the same time step and number of iterations in the test as in the simulation in which you desire to use your bulk material. Computational time is roughly linearly proportional to the simulation frequency (the inverse of the time step) and linearly proportional to the number of iterations. `self.agxOnly` sets graphics on/off. With `self.agxOnly = True` only the simulations are carried out, and with `self.agxOnly = False` simulations are carried out and windows are opened in which graphics are rendered. `self.agxOnly` is important to consider if you run the tests on a computer using a remote connection. With `self.agxOnly = False`, access to open windows may be denied which results in a crash. If you use a remote connection, `self.agxOnly = True` is recommended.

Up to three different particle radii can be specified (`self.particleRadius`, `self.particleRadiusMin`, and `self.particleRadiusMax`). `self.particleRadius` is the basis for the calculation of the cohesive overlap. The particles carry weights such that the particles of radius `self.particleRadius` will constitute `self.particleWeight / (self.particleWeight + self.particleWeightMin + self.particleWeightMax)` of the total mass of the particle sample, and similarly for the particles of radii `self.particleRadiusMin` and

`self.particleRadiusMax`. With the settings in Code snippet 2, the particles of radius `self.particleRadius` will constitute `0.4 / (0.4 + 0.3 + 0.3) = 0.4` of the total mass, i.e. 40%. The particle radii have an effect on the static angle of repose, so it is recommended to use the same particle radii in the test as in your simulation.

The number of particles to emit is calculated from the particle radii and the radius of the base (`self.baseRadius`). A larger `self.baseRadius` to `self.particleRadius` ratio will increase the number of particles emitted. More particles imply a more refined slope of the granular pile, and a greater precision in the calculation of the static angle of repose. With the assumption that the particle radii are approximately equal, and the particle weights are approximately equal, the following ratio is recommended: `self.baseRadius / self.particleRadius > 30`. Computational time is roughly linearly proportional to the number of particles, so increasing the mentioned ratio can drastically increase the computational time. For example, by doubling the ratio, the number of particles is increase by a factor of roughly eight. `self.numSlices` specifies the number of slices to divide the granular slice into (i.e. the number of individual angles of repose to calculate), and `self.numCells` specifies the number of cells associated with each slice. Both an larger number of slices and a larger number of cells increase the number of points for the calculation, increasing accuracy in the angles of repose calculations. However, too many slices and cells is unwanted since it runs the risk of overlapping cells such that a particular particle can be the particle with the highest z-coordinate in two or more cells.

The settings in Code snippet 2 are used for the simulation demonstrated in Test description: the test.

24.3.3.6. Run the static angle of repose test

The static angle of repose test should be started with `/static_angle_of_repose_test/` as the current working directory. For example, start an AGX terminal and browse to `/static_angle_of_repose_test/`. To start the static angle of repose test, run `runStaticAngleOfReposeTest.py` with appropriate arguments. Two examples are

```
python runStaticAngleOfReposeTest.py SimConfDefault.py material_A.json

python runStaticAngleOfReposeTest.py SimConfMySettings.py my_materials
```

The first example executes the static angle of repose test for `json/material_A.json` with simulation settings specified in `SimConfDefault.py`. The second example executes the static angle of repose test for all particle materials in the directory `json/my_materials/` with simulation settings specified in `SimConfMySettings.py`. A full description on how to specify arguments is found by consulting `runStaticAngleOfReposeTest.py`. As you hit enter, the static angle of repose test with be executed.

24.3.3.7. Inspect the test results

When the test or tests have completed, bar graphs (one per material) are stored as pdfs in `static_angle_of_repose_test/fig/` , and result files (one per material) are stored as `json` files in `static_angle_of_repose_test/results/`. We begin by considering the result file from a test with `material_A.json`, see Code snippet 3.


```
{
     "Description": {
          "date": "2022-10-16 19:21:24",
          "test": "static_angle_of_repose_test",
          "particleSpecification": "material_A"
     },
     "ResultsFromIndividualAnglesOfRepose": {
          "averageAngleOfRepose": 37.536869070161906,
          "standardDeviation": 0.8914828660521458,
          "minAngleOfRepose": 35.89691538629683,
          "maxAngleOfRepose": 39.4879976298608
     },
     "IndividualAnglesOfRepose": [
          39.4879976298608,
          37.89848285992389,
          38.58629975167365,
          38.59317117760724,
          37.86961668429997,
          37.1151311577757,
          36.796149118478375,
          37.78694513210245,
          38.57604638098495,
          36.59148912637279,
          36.88952461601244,
          37.17808882132403,
          37.46446064702305,
          36.13803741623653,
          36.67545555864243,
          37.976255819948385,
          37.111137010605276,
```

```
            38.14838087881512,
            35.89691538629683,
            37.957796229254164
        ],
        "ParticleProperties": {
            "particleRadiusMin": 0.0135,
            "particleRadiusMedium": 0.015,
            "particleRadiusMax": 0.0165,
            "particleWeightMin": 0.3,
            "particleWeightMedium": 0.4,
            "particleWeightMax": 0.3,
            "particleCohesiveOverlapRatio": 0.025,
            "particleCohesion": 0.0,
            "particleDamping": 4.5,
            "particleFriction": 0.45,
            "particleRestitution": 0.1,
            "particleTwistingResistanceCoefficient": 0.01,
            "particleRollingResistance": 0.25,
            "particleSpecificDensity": 2700.0,
            "particleYoungsModulus": 1000000000.0
        },
        "BaseProperties": {
            "baseCohesiveOverlapRatio": 0.025,
            "baseCylCohesion": 0.0,
            "baseCohesion": 0.0,
            "baseDamping": 4.5,
            "baseFriction": 0.45,
            "baseRestitution": 0.1,
            "baseTwistingResistanceCoefficient": 0.01,
            "baseRollingResistance": 0.25,
            "baseSpecificDensity": 2700.0,
            "baseYoungsModulus": 1000000000.0
        },
        "HollowCylProperties": {
            "hollowCylCohesiveOverlapRatio": 0.025,
            "hollowCylCohesion": 0.0,
            "hollowCylDamping": 4.5,
            "hollowCylFriction": 0.0,
            "hollowCylRestitution": 0.0,
            "hollowCylRollingResistance": 0.0,
            "hollowCylTwistingResistanceCoefficient": 0.01,
            "hollowCylSpecificDensity": 2700.0,
            "hollowCylYoungsModulus": 1000000000.0
        },
        "SimulationParameters": {
            "timeStep": 0.001,
            "numPPGSRestingIterations": 500,
            "useParallelPGS": true,
            "useGranularWarmStarting": true,
            "use32bitGranularBodySolver": true,
            "gravitationalAcceleration": -9.81
        }
        "TestParameters": {
            "baseRadius": 0.6,
            "numberOfSlices": 20,
            "numberOfCells": 15,
            "cellRadius": 0.02475,
            "hollowCylSpeed": 0.625
            "emissionDuration": 3,
            "postEmissionRelaxDuration": 2,
            "postGranularPileFormationRelaxDuration": 4,
        }
}
```


Code Snippet 3 - The result file for `material_A.json`: `result_static_angle_of_repose_test_material_A.json`

All essential settings of the test and key results are summarized in the result file (Code snippet 3).

The key results are located under `"ResultsFromIndividualAnglesOfRepose"`. `"averageAngleOfRepose"` is the average static angle of repose (which is the arithmetic mean of the individual static angles of repose in `"IndividualAnglesOfRepose"`), `"standardDeviation"` is the standard deviation of `"averageAngleOfRepose"`, `"minAngleOfRepose"` is the minimum static angle of repose (the smallest value in `"IndividualAnglesOfRepose"`), and `"maxAngleOfRepose"` is the largest static angle of repose (the largest value in `"IndividualAnglesOfRepose"`). We see that the central result of the test, for one standard deviation, is the following average angle of repose:

$$\theta = 37.54 \pm 0.89^\circ \ .$$

Under `"ParticleProperties"` the particle parameters specified by `"particleSpecification"` and used in the test that describe particle-particle interactions are given. Also, the particle radii used and their weights are reported. The objects `"ParticleProperties"` and `"ResultsFromIndividualAnglesOfRepose"` together constitute the sought mapping between particle parameters and bulk properties: the particle material with particle parameters given by `"ParticleProperties"` will yield a bulk material with the static angle of repose `"averageAngleOfRepose"`.

Under `"BaseProperties"` particle-base interactions are given, and under `"HollowCylProperties"` particle-hollow cylinder interactions are specified.

We now inspect the bar graph of the test, see Fig. 24.10.

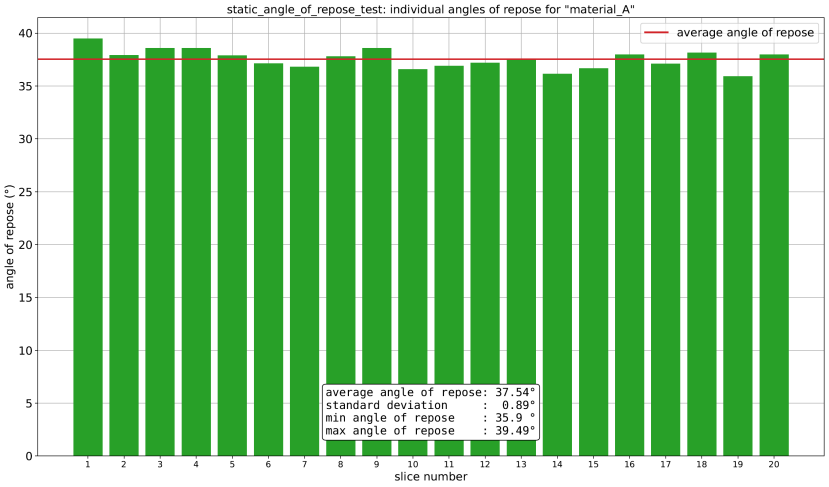


Fig. 24.10 The bar graph for `material_A.json`: `bar_graph_static_angle_of_repose_test_my_gravel.pdf`. Individual angles of repose for each slice are visualized, with quantities calculated from them summarized in a box.

The bar graph provides a convenient overview of the results of the test.

24.3.3.8. Evaluate test results

With the test results checked out, it's time to evaluate them. If `averageAngleOfRepose` agress sufficiently well with the corresponding value of your desired bulk material, the material calibration is complete with respect to the static angle of repose. You then enter the used particle parameters in the static angle of repose test in the applicable program (AGX Dynamics, agxTerrain, or Momentum Granular). This yields desired bulk properties with respect to the static angle of repose. If `averageAngleOfRepose` does not agree sufficiently well with the value of your bulk material, the test needs to be repeated with different particle parameters until a desired `averageAngleOfRepose` is obtained, see Selection of particle parameters.

24.4. The triaxial test

The triaxial test determines the bulk cohesion `c` and friction angle `phi` (described in Bulk properties). The test is carried out in real life or with a computer simulation. In this section we describe the test implemented in AGX Dynamics. In the test, a particle sample is confined in a container, and is subjected to stresses. We consider the container to be a cubic box surrounded by 6 walls, with the z-axis pointing upwards, and the bottom wall parallel to the xy-plane. In the horizontal directions (the x- and y-directions), the sample is subjected to a constant specified stress by moving the vertical walls in appropriate directions. This stress is denoted the *consolidation stress*. If the consolidation stress drops below the specified stress, the vertical walls move toward the center of the sample, and if it exceeds the specified stress, the vertical walls move away from the center. This keeps the consolidation stress constant. In the vertical direction (z-direction), the horizontal walls move toward the center with *constant* speed and a variable force is applied to these walls such that this constant speed is maintained. As the horizontal walls move toward the center, the vertical pressure increases such that it exceeds the consolidation stress. This difference in stresses in the vertical and horizontal directions introduces a shearing stress in the particle sample. The vertical stress increases up until the point where the sample fails by shearing. When the soil shears, the sample expands and the vertical stress drops. The consolidation stress and the vertical stress are stored regularly during the test. The test is then repeated several times for different consolidation stresses. By analyzing the stress data, bulk properties can be obtained.

In Test description we describe how the test works, and in User guide (detailed) we describe in detail how to carry out the test with step by step instructions and how to interpret the results. In User guide (brief) we offer a very brief user guide that can be used as a cheat sheet for readers familiar with the test. It is strongly recommended to first read the Test description as this will illuminate the steps in the User guide, particularly the interpretation of the results.

24.4.1. Test description

24.4.1.1. Setup

In the first part of the test setup particle parameters such as friction and rolling resistance are specified. These parameters specify particle-particle interactions and will be applied to the particles that represent the bulk material. Particle parameters are specified in a particle material file. Wall parameters — specifying particle-wall interactions — are specified in a wall material file.

24.4.1.2. Creation of an initial simulant

With particle parameters and simulation settings configured, the triaxial test and associated simulations can now be executed. This first part in this process is the creation of a so called *initial simulant*, using the model `CreateSimulant.agxPy`, see Fig. 24.11.

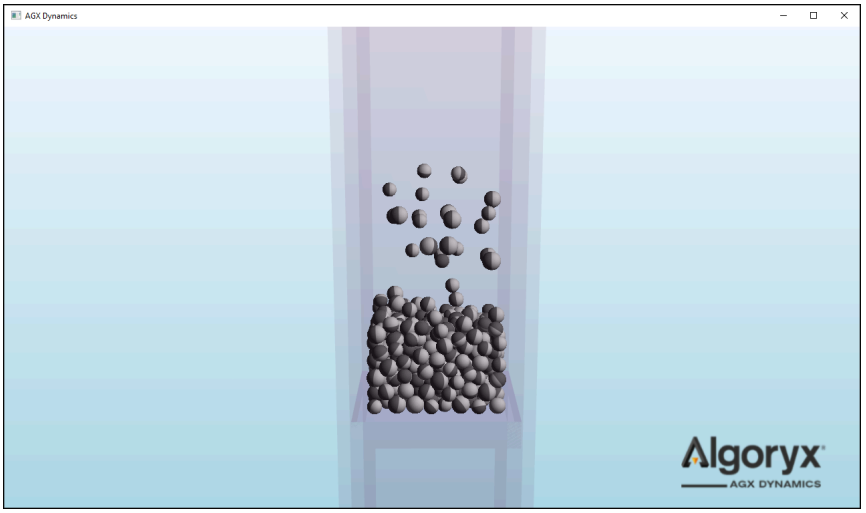


Fig. 24.11 Creation of an initial simulant.

During the creation of the initial simulant, particles a spawned in a rectangular container in a loose state until a rectangular sample has formed. Then, the sample is saved.

24.4.1.3. Consolidation of initial simulant

Next, the saved initial simulant from the previous step is loaded into a rectangular box using `consolidation.agxPy`, see Fig. 24.12.

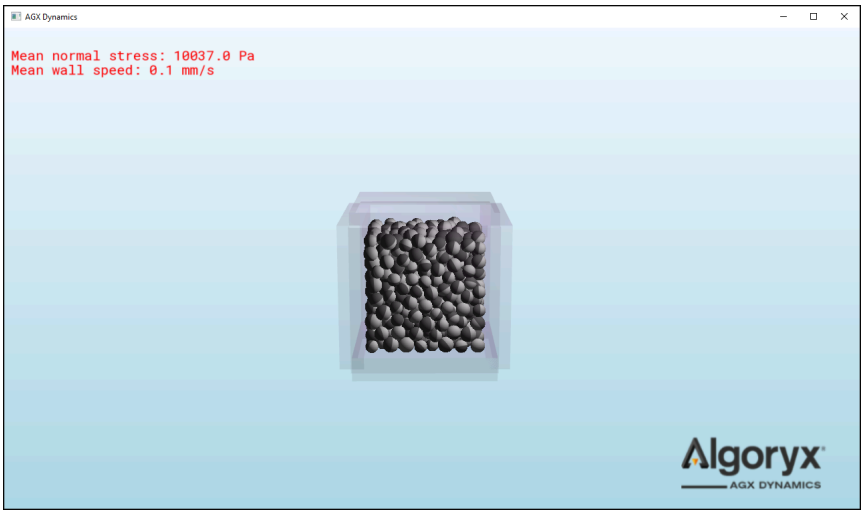


Fig. 24.12 Consolidation of the initial simulant.

During the consolidation the walls are moved such that the specified consolidation stress, which is 10 kPa in Fig. 24.12, is reached. If the stress drops below 10 kPa, the walls are driven towards the center of the sample. If the stress exceeds 10 kPa, the walls are driven away from the center. When the relative error between the actual measured stress and the specified stress goes below a small number, the soil sample is saved.

24.4.1.4. Triaxial test of the consolidated sample

Now the preparatory simulations are complete. Next, the saved consolidated sample from the previous step is loaded into the triaxial test using `triaxialTest.agxPy`, see Fig. 24.13.

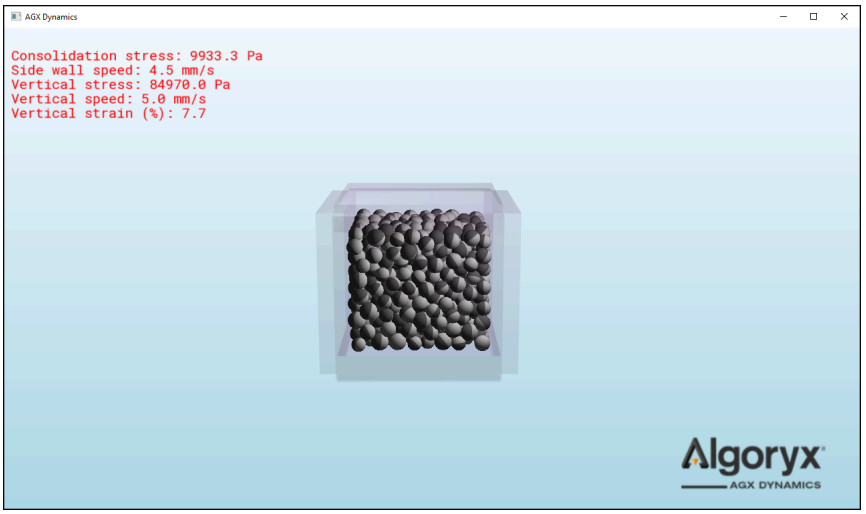


Fig. 24.13 The triaxial test.

During the triaxial test, the vertical walls move such that the horizontal stresses are maintained at the consolidation stress (10kPa in Fig. 24.13). The horizontal walls move towards the center in the vertical direction with constant speed, increasing the vertical pressure. We note that at a vertical strain (compression) of 7.7%, the vertical stress is 84.97 kPa. The pressures are continuously stored in a data file. By comparing Fig. 24.12 and Fig. 24.13, we note that the sample has been compressed in the vertical direction and expanded in the horizontal directions in Fig. 24.13.

24.4.1.5. Repetition of triaxial test

The consolidation and triaxial test are then repeated a number of times, say, 10 times, for different consolidation pressures. After these repetitions the test is complete.

24.4.1.6. Identifications of bulk properties

We recall that for each run, the consolidation stress and vertical stress are stored at regular short intervals. Also, the particular stresses at which the soil fails need to be identified in order to determine bulk properties. An automatically generated plot of the stresses as a function of vertical displacement from a triaxial test is shown in Fig. 24.14.

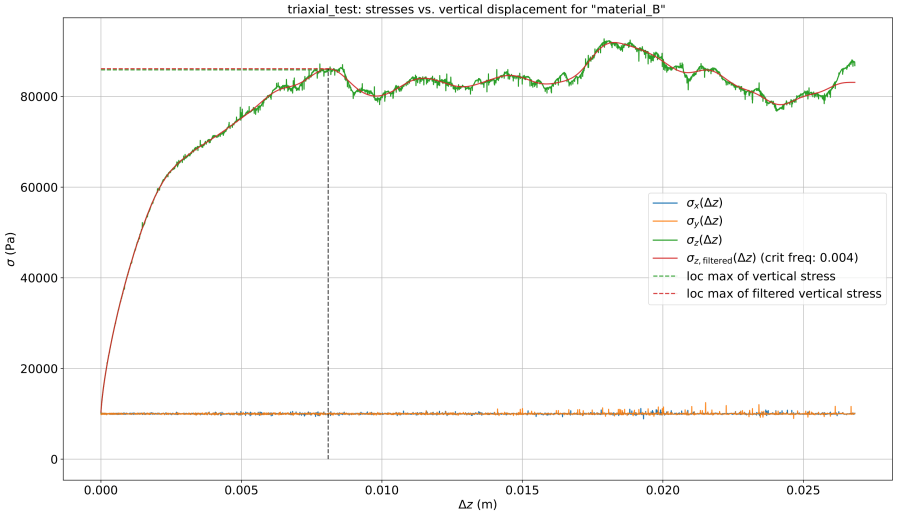


Fig. 24.14 Stresses (σ, `sigma`) as functions of the vertical displacment (Δz, `Delta z`) for a triaxial test with a consolidation stress of 10 kPa.

The plot in Fig. 24.14 shows the evolution of the stresses (y-axis) as a function of the vertical displacement of the walls (x-axis). The plot is from the triaxial test in Fig. 24.13. We note that the measured horizontal stresses, `sigma_x` and `sigma_y`, representing the consolidation stress, lie close to the specified consolidation stress of 10 kPa, as desired. As the sample is more and more compressed in the vertical direction, i.e. as the vertical displacement increases, we note that the vertical stress `sigma_z` increases. As described earlier, when the soil fails and shears, it expands, resulting in a drop in vertical pressure. An algorithm using a filtered version of the vertical stress seeks to identify this point of failure, which usually is the first local maximum of a filtered version of `sigma_z`. This identification is indicated by the dashed lines in Fig. 24.14. With a visual inspection, we note that this indeed seems to be where the vertical stress drops.

Now, the key results from the simulations are stored in a `.json` file. The following is an abbreviated example (`result_triaxial_test_material_B.json`).

```
{
        "Description": {
```

```
            "date": "2022-09-23 11:10:12",
            "test": "triaxial_test",
            "particleSpecification": "material_B"
        },
        "BulkPropertiesFromBulkData": {
            "c": 4.737650504718438,
            "phi": 46.09,
            "standardError_c": 0.4726372802858118,
            "standardError_phi": 0.46488510277406264
        },
        "BulkData": {
            "consolidationStress": [
                5069.832186961724,
                10036.720190218033,
                15334.126180643405,
                ...
            ],
            "verticalStress": [
                50377.87891396498,
                85868.52312832276,
                117089.52718754749,
                ...
            ]
        },
        "ParticleProperties": {
            "particleRadiusMin": 0.0135,
            ...
        },
        "SimulationParameters": {
            "timeStep": 0.0003333333333333333,
            ...
        },
        "FilterParameters": {
            "filterOrder": 4,
            ...
        }
}
```


Code Snippet 4 - `result_triaxial_test_material_B.json` (abbreviated, as indicated by the `...`)

Consider Code snippet 4. In the first run (with a specified consolidation stress of 5000 Pa), the algorithm has identified the point where the soils fails by shearing to be at the consolidation stress of 5069.8 Pa, and the vertical stress of 50378 Pa. Similarly, in the second run (with a specified consolidation stress of 10000 Pa), the consolidation and vertical stresses at failure are 10037 Pa and 85869 Pa, respectively (this is the failure that is indicated by the dashed lines in Fig. 24.14), and so on. The data in `BulkData` is used to calculate the bulk cohesion `c` and friction angle `phi`, and the standard errors `standardError_phi` (associated with `phi`) and `standardError_c` (associated with `c`). Let us see how by inspecting Fig. 24.15.

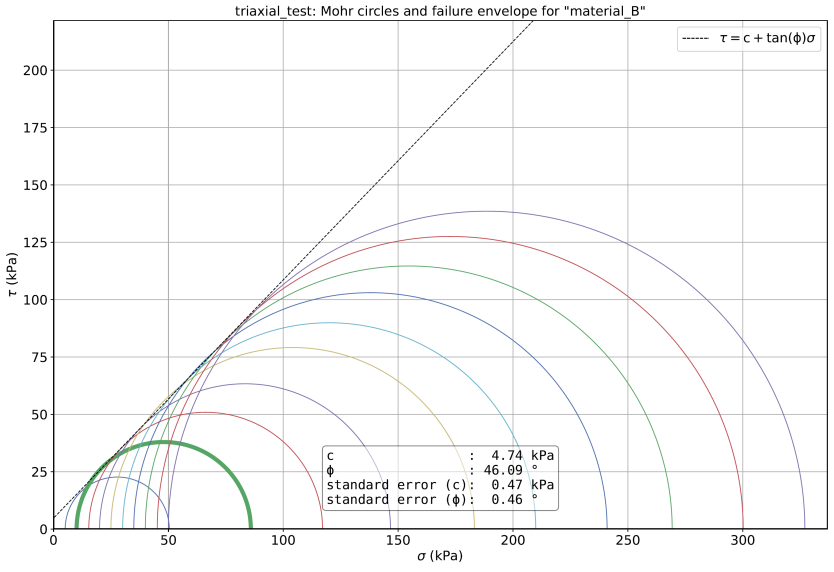


Fig. 24.15 Mohr circles and Mohr-Coulomb failure envelope created from the stresses at soil failure of the triaxial tests.

The plot in Fig. 24.15 is constructed from the stresses in `bulkData` in `result_triaxial_test_material_B.json` (see Code snippet 4). We note that there are 10 semi-circles known as *Mohr circles*, each one corresponding to a triaxial test with a particular specified consolidation pressure. For `result_triaxial_test_material_B.json`, the specified consolidation pressures are `[5000, 10000, ..., 45000, 50000]` Pa, hence 10 semi-circles. Each semi-circle is constructed such that the right-most *x*-coordinate of it is the vertical stress, and the left-most *x*-coordinate is the consolidation stress (these coordinates lie on the *x*-axis). The green semi-circle in Fig. 24.15, highlighted with a thick line, corresponds to the test with the specified consolidation stress 10 kPa, see Fig. 24.14. By inspecting Fig. 24.15, we see that the right-most coordinate of the thick green semi-circle is around 85 kPa, which is the recorded vertical stress of 85869 Pa at failure (see `result_triaxial_test_material_B.json`). Moreover, again by inspection of Fig. 24.15, we see that the left-most coordinate of the thick green semi-circle is around 10 kPa, which is the recorded consolidation stress of 10037 Pa at failure. If the algorithm fails to identify the point of soil failure, appropriate measures need to be taken (see User guide (detailed): Inspect the unvalidated test results).

The final step in order to obtain bulk properties is to construct a straight line that is tangent to all these semi-circles. This straight line is known as the *Mohr-Coulomb failure envelope*, and is on the form

```
tau = c + sigma * tan(phi),
```

where `tau` is shear stress, and `sigma` is normal stress. The failure envelope is fitted to be tangent to the semi-circles by linear least squares regression. The linear regression also yields standards errors: `std_err_slope` is associated with `phi`, and `std_err_intercept` is associated with `c`. In Fig. 24.15, the failure envelope is the dashed straight line. This particular failure envelope has `c=4.74` kPa, and `phi=46.09` degrees, as noted in the plot. These are the bulk properties yielded from the triaxial test. The triaxial test process is complete.

24.4.2. User guide (brief)

The triaxial test is performed by executing the following steps:

1. Find the bulk cohesion `c` and friction angle `phi` of your desired bulk material
2. Make sure that required python modules are installed (see `triaxial_test/requirements.txt`)
3. Create your particle material(s) (see e.g. `particle_soil/json/sand_1.json`)
4. Set up the simulation settings file (see `triaxial_test/SimConfDefault.py`)
5. Open an AGX terminal at `/triaxial_test/` and run the triaxial test (`runTriaxialTest.py`)
6. Inspect the unvalidated test result(s) (stored in `triaxial_test/results/`)
7. Inspect the failure envelope (stored in `triaxial_test/fig/`)
8. Validate the stress plots (stored in `triaxial_test/fig/`)
    1. If needed, adjust the simulation settings file and use `triaxial_test/replotAll.py`
    2. If needed, adjust vertical failure stress values in the test result(s) and use `triaxial_test/replotFailureEnvelope.py`
9. Inspect the validated test result(s). If the bulk cohesion `c` and the friction angle `phi` match the values of the desired bulk material, the material calibration is complete. If not, rerun the triaxial test with different particle parameters until desired bulk properties are yielded.

24.4.3. User guide (detailed)

In the following we provide detailed step by step instructions on how to execute the triaixal test.

24.4.3.1. Find the bulk properties `c` and `phi` of your desired bulk material

The bulk properties `c` and `phi` of a particular material can be found by consulting a table of soil or granular materials, if the material is not too uncommon. If tables are inaccessible or do not exist, physical experiments on the real material need to be carried out to measure properties. If such experiments are infeasible, properties need to be judiciously estimated.

24.4.3.2. Locate the triaxial test

Your AGX Dynamics installation path should be on the form `C:/.../AGX-<version>`. Then, the triaxial test is located at `AGX-<version>/data/python/material_calibration/particle_soil/triaxial_test`.

24.4.3.3. Install python modules

The test in written in python and the required modules are listed in `triaxial_test/requirements.txt`. Make sure that they are installed for the python version specified to be used with AGX Dynamics. To install the modules, use e.g. `python -m pip install -r requirements.txt`.

24.4.3.4. Create your particle material(s)

Browse to `/particle_soil/json/`. This is the directory in which particle materials are stored and should be created. The soil sample in the triaxial test will have particle parameters of a file from this directory or a subdirectory of it, say `/particle_soil/json/my_materials/`. More specifically, these

parameters specify particle-particle interactions. For convenience, copy one of the existing files, for example `sand_1.json`, rename the file to, say, `material_B.json` and enter desired particle parameters and save the file. An example is shown in Code snippet 5.

```
{
    "soilName" : "MATERIAL_B",
    "ParticleProperties" : {
      "particleCohesiveOverlapRatio" : 0.025,
      "particleCohesion" : 7500,
      "particleDamping" : 4.5,
      "particleFriction" : 0.35,
      "particleRestitution" : 0.1,
      "particleRollingResistance" : 0.35,
      "particleSpecificDensity" : 2200,
      "particleYoungsModulus" : 1e9
    }
}
```

Code Snippet 5 - `material_B.json`

You can create more than one material and run triaxial tests on more than one material in one go. Also, you can place several materials in a new directory in `/particle_soil/json/`, say `/particle_soil/json/my_materials/`, and perform triaxial tests on all materials in that folder.

Wall parameters — specifying particle-wall interactions — are specified in `triaxial_wall.json` (see `/particle_soil/json/`). Default wall parameters are specified in this file. (These values can be adjusted if needed.)

24.4.3.5. Set up the simulation settings file

The simulation settings file with default settings is `triaxial_test/SimConfDefault.py`. In Code snippet 6 an abbreviated version of this file is given.

```
class SimConf():
    def __init__(self, particleSpecificationPath=defaultSpecificationPath, particleSpecification="")

        ### Simulation parameters ###
        self.timeStep = 1/2000
        self.numPPGSRestingIterations = 400
        self.agxOnly = False
        ...

        ### Particle material ###
        self.particleRadius = 0.015
        self.particleRadiusMin = 0.9 * self.particleRadius
        self.particleRadiusMax = 1.1 * self.particleRadius
        self.particleWeight = 0.4
        self.particleWeightMin = 0.3
        self.particleWeightMax = 0.3
        ...

        ### Triaxial test parameters ###
        self.wallSideLength = 0.25
        ...

        ### Signal filter parameters ###
        self.filterOrder = 4
        ...
```

Code Snippet 6 - `SimConfDefault.json`

In this file simulation settings such as time step, particle radii, etc. are specified. If you desire to alter these simulation settings, it is recommended to save them in a new file, say `triaxial_test/SimConfMySettings.py`. Usually, the default values can be used for most settings. The settings of most interest are shown in Code snippet 6.

The time step (`self.timeStep`) and the number of iterations for the iterative solver (`self.numPPGSRestingIterations`) have an impact on the bulk properties. It is recommended, if possible, to use the same time step and number of iterations in the test as in the simulation in which you desire to use your bulk material. Computational time is roughly linearly proportional to the simulation frequency (the inverse of the time step) and linearly proportional to the number of iterations. For an explanation of `self.agxOnly`, see SimfConfDefaut (static angle of repose test).

For the setting of particle radii, see SimfConfDefaut (static angle of repose test).

Various parameters of the test can be specified such as length of walls (`self.wallSideLength`), wall speeds, etc. are specified. Default values are recommended for `### Triaxial test parameters ###`. See `SimConfDefault.py` for additional information.

For the identification of failure points a filtered signal of the vertical stress is used. The parameters of the filter are specified here. `self.filterOrder` specifies the order of the filter. Default values are recommended for `### Signal filter parameters ###`. See `SimConfDefault.py` for additional information.

24.4.3.6. Run the triaxial test

Start an AGX terminal and browse to `/triaxial_test/`. This is the directory from which the triaxial test should be run. To start the triaxial test, run `runTriaxialTest.py` with appropriate arguments. Two examples are

```
python runTriaxialTest.py SimConfDefault.py material_B.json

python runTriaxialTest.py SimConfMySettings.py my_materials
```

The first example executes the triaxial test for `material_B.json` with simulation settings specified in `SimConfDefault.py`. The second example executes the triaxial test for all particle materials in the directory `json/my_materials` with simulation settings specified in `SimConfMySettings.py`. A full description on how to specify arguments is found by consulting `runTriaxialTest.py`. As you hit enter, the triaxial test with associated simulations will be executed.

24.4.3.7. Inspect the unvalidated test result(s)

When the triaxial test has completed, plots describing strain, stress, etc. are stored in `triaxial_test/fig/` and key results are stored in `triaxial_test/results/`. For example, if your particle parameter file is `material_B.json`, then plots are stored in `triaxial_test/fig/material_B/`, and a key result file are stored in `triaxial_test/results/result_triaxial_test_material_B.json`. We begin by considering the result file. In Code snippet 7 an example result file is shown.


```
{
      "Description": {
          "date": "2022-09-23 11:10:12",
          "test": "triaxial_test",
          "particleSpecification": "material_B"
      },
      "BulkPropertiesFromBulkData": {
          "c": 4.737650504718438,
          "phi": 46.09,
          "standardError_c": 0.46488510277406264,
          "standardError_phi": 0.4726372802858118,
      },
      "BulkData": {
          "consolidationStress": [
                5069.832186961724,
                10036.720190218033,
                15334.126180643405,
                20042.03025732538,
                24971.475941893936,
                30081.725345084255,
                34924.68195807154,
                39947.85331379516,
                45142.31398754626,
                49976.82805899259
          ],
          "verticalStress": [
                50377.87891396498,
                85868.52312832276,
                117089.52718754749,
                146661.62243417173,
                183183.91905994146,
                209881.79120832373,
                240935.25233170495,
                269258.88806409494,
                300170.4442784592,
                327011.1903817603
          ]
      },
      "ParticleProperties": {
          "particleRadiusMin": 0.0135,
```

```
            "particleRadiusMedium": 0.015,
            "particleRadiusMax": 0.0165,
            "particleWeightMin": 0.3,
            "particleWeightMedium": 0.4,
            "particleWeightMax": 0.3,
            "particleCohesiveOverlapRatio": 0.025,
            "particleCohesion": 7500.0,
            "particleDamping": 4.5,
            "particleFriction": 0.35,
            "particleRestitution": 0.1,
            "particleRollingResistance": 0.35,
            "particleSpecificDensity": 2200.0,
            "particleYoungsModulus": 1000000000.0
        },
        "SimulationParameters": {
            "timeStep": 0.0003333333333333333,
            "numberOfPPGSRestingIterations": 500,
            "consolidationSpeed": 0.0025,
            "horizontalSpeed": 0.1,
            "verticalSpeed": 0.005,
            "horizontalAcceleration": 10,
            "verticalAcceleration": 0.01
        },
        "FilterParameters": {
            "filterOrder": 4,
            "criticalFrequency": 0.004,
            "minVertStressToConsStressRatio": 1.25
        }
}
```


Code Snippet 7 - `results_triaxial_test_material_B.json`

All essential settings of the triaxial test and key results are summarized in the result file (Code snippet 7). The date of the test and the name of the particle material are given under `Description`. The key results are located under `bulkPropertiesFromBulkData`. `c` is the bulk cohesion (in kPa), and `phi` is the friction angle (in degrees) associated with the particle material `material_B.json`. `standardError_phi` is the standard error associated with `phi`, and `standardError_c` is the standard error associated with `c`. The stresses at which soil failure occur are given in `BulkData`. For example, in the first test, soil failure occurs at the condolisation stress of 5069.8 Pa and the vertical stress of 50378 Pa. The data in `BulkData` is used to calculate the results in `bulkPropertiesFromBulkData`, as the name of that object suggests. The particle parameters used for the triaxial test, i.e. the particle parameters that are associated with the bulk properties in `bulkPropertiesFromBulkData`, are given in `particleProperties`. The simulating settings are located under `simulationParameters`, and the filter parameters used are given in `filterParameters`.

24.4.3.8. Inspect the Mohr-Coulomb failure envelope

Next we inspect the Mohr-Coulomb failure envelope, see Fig. 24.15.

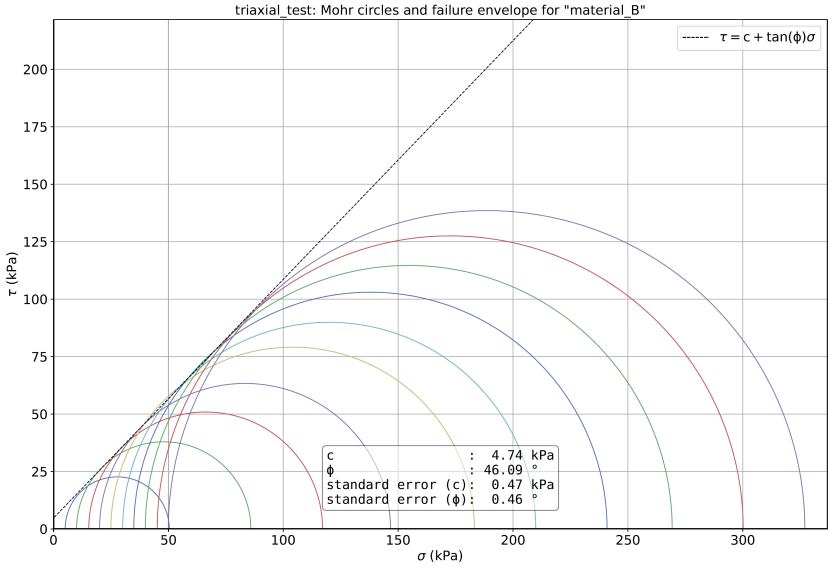


Fig. 24.16 The Mohr Coulomb failure envelope of `fig/bulk_properties_triaxial_test_material_B.pdf`.

If all or most semi-circles are just tangent to the failure envelope (the straight dashed line), this is an indication that test results are reliable. We note in Fig. 24.15 that this is the case. If one or more circles clearly deviate from this behaviour, this is an indication that the failure point identification algorithm has failed for the tests associated with those circles. (We note that the straight line is not entirely tangent to the semi-circle with right-most x-coordinate between 150 and 200 kPa, but the deviation is minor.) In any case, it is best to validate the stress plots, as described in the next step.

24.4.3.9. Validate the stress plots

To validate the results described in the previous step, we study the stress plots. In Fig. 24.17, we show `fig/material_B/triaxial_stress_triaxialTest_material_B_10000Pa.pdf`, with the ending of the file name indicating that the consolidation stress is specified to 10 kPa.

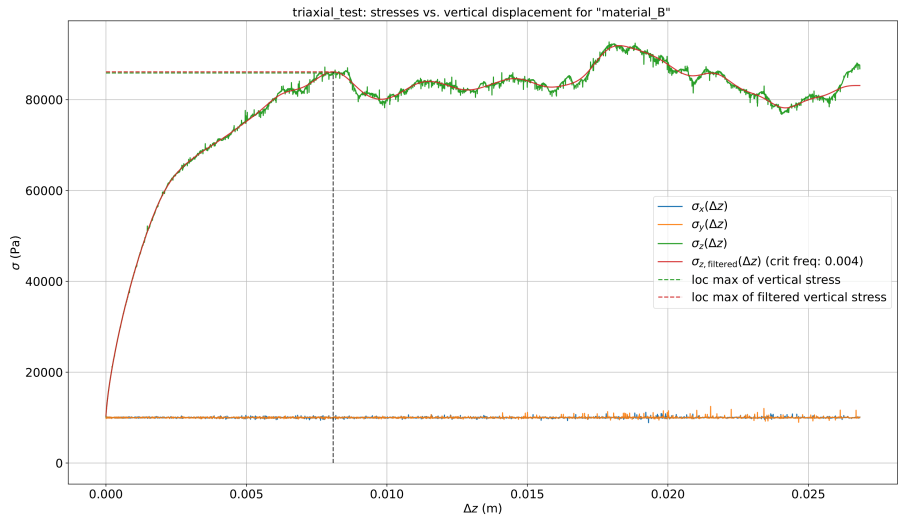


Fig. 24.17 Stresses (σ, `sigma`) as functions of the vertical displacment (Δz, `Delta z`) for a triaxial test with a consolidation stress of 10 kPa.

We check that the algorithm has correctly identified the point at which the soil fails, i.e. where the vertical stress drops. We see that the algorithm seems to have identified the correct point.

Now, since the consolidation stress is essentially constant throughout the test, as desired, the algorithm only needs to identify the correct vertical stress. This means that if the failure point reported by the algorithm is to the right (perhaps even far to the right) of the actual failure point (as determined by visual inspection), but the reported failure point has essentially the same vertical stress as the actual failure point, we can consider the algorithm to have correctly identified the failure point, since then the reported vertical stress of failure is essentially identical to the actual vertical stress of failure (and the reported consolidation stress of failure is essentially identical to the actual consolidation stress of failure, since the consolidation stress is essentially constant throughout the test). This is a subtle but important argument, and the takeaway is to check if the reported failure point agrees with the actual failure point with respect to only the y-axis, and not *both* the x- and y-axis.

All stress plots (one for each specified consolidation stress) should be inspected to see if the reported failure points agree decently with the actual failure points. However, sometimes it is difficult to visually identify the failure point even with keen observing due to the sensitivity of the test, noise and the complicated behaviour of the vertical stress curve. At any rate, if the reported failure points agree with the actual failure points, the test results are validated and you can move to the next step.

If however, one or more reported points differ substantially from the actual failure points, there are two appropriate actions, denoted **A** and **B**, that can be taken:

A. Reconfigure the simulation settings

Reconfigure the filter by changing filter parameters in the simulation settings file, see In Code snippet 6. Then use `runReplotAll.py` to only update the plots with the new filter parameters and recompute the results file. This is a fast computation and does not rerun the triaxial test. To carry out this action, do as follows. Let's say that you ran the triaxial tests with

```
python runTriaxialTest.py SimConfMySettings_A.py material_B.json
```

Then you can simply configure new filter parameters in, say, `SimConfMySettings_B.py`, and run

```
python runReplotAll.py SimConfMySettings_B.py material_B.json
```

This updates all plots and the test results file according to the settings in `SimConfMySettings_B.py`. If this yields satisfactory identification points, the test results are validated and you can move the the next step.

For more details, consult `runReplotAll.py`

B. Manually adjust vertical stresses

If different filter settings do not improve the identification of failure points satisfactorily, you need to manually specify vertical stresses in the result file. The procedure is as follows. Consider the abbreviated result file in Code snippet 8.

```
{
     "Description": {
          "date": "2022-09-23 11:10:12",
      "test": "triaxial_test",
          "particleSpecification": "material_B"
     },
     "BulkPropertiesFromBulkData": {
          "c": 4.737650504718438,
          "phi": 46.09,
          "standardError_phi": 0.4726372802858118,
          "standardError_c": 0.46488510277406264
     },
     "BulkData": {
          "consolidationStress": [
               5069.832186961724,
               10036.720190218033,
               15334.126180643405,
               ...
          ],
          "verticalStress": [
               50377.87891396498,
               85868.52312832276,
               117089.52718754749,
               ...
          ]
     },
     ...
}
```

Code Snippet 8 - `result_triaxial_test_material_B.json` with an incorrect vertical stress for the 3rd test.

If the reported failue point is incorrect in, say, the 3rd test, i.e. the vertical stress of 117089.5 Pa is incorrect, we need to adjust that value. Let's say that the actual vertical stress is 130000 Pa, as determined by visual inspection. Then, we replace the original value with the new value, such that the file in Code snippet 8 is obtained.

```
{
     "Description": {
          "date": "2022-09-23 11:10:12",
          "particleSpecification": "material_B"
     },
     "BulkPropertiesFromBulkData": {
          "c": 4.737650504718438,
          "phi": 46.09,
          "standardError_phi": 0.4726372802858118,
          "standardError_c": 0.46488510277406264
     },
     "BulkData": {
          "consolidationStress": [
               5069.832186961724,
               10036.720190218033,
               15334.126180643405,
               ...
          ],
          "verticalStress": [
               50377.87891396498,
               85868.52312832276,
               130000,
               ...
          ]
     },
     ...
  }
```

Code Snippet 8 - `triaxial_result_material_B.json` with the incorrect vertical stress for the 3rd test replaced with the correct value.

We then save the file (without chaning its name) and use `RunReplotFailureEnvelope.py`. This will update the plot of the failure envelope, and consequently also the results in `bulkPropertiesFromBulkData` since the results in than object is obtained from the failure envelope. To carry out this action, do as follows. Let's say that you ran the triaxial test with

```
python runTriaxialTest.py SimConfMySettings_A.py material_B.json
```

and perhaps also

```
python runReplotAll.py SimConfMySettings_B.py material_B.json
```

as described in **A**. (If you did or did not use `runReplotAll.py` does not matter.) Then, to update the plot of the failure envelope and the results in `bulkPropertiesFromBulkData`, run

```
python runReplotFailureEnvelope.py SimConfMySettings_C.py material_B.json
```

where `SimConfMySettings_C.py` is either `SimConfMySettings_A.py` or `SimConfMySettings_B.py` (it does not matter). As the manoeuver of replacing vertical stress values is independent of the filter algorithm and the triaxial test itself, this will, with respect to plots in `fig/material_B/`, only update the failure envelope plot and not the other plots.

After this manual adjustments and the execution of `runReplotFailureEnvelope.py`, the test results are validated and you can move to the next step.

For more details, consult `runReplotFailureEnvelope.py`.

24.4.3.10. Inspect the validated test result(s)

When the test results have been validated, it's time to compare them against the values of your desired bulk material. If `c` and `phi` agree with the values of the desired bulk material, the material calibration is complete with respect to `c` and `phi`. You then enter the used particle parameters in the triaxial test in AGX Dynamics, agxTerrain, or Momentum Granular. This yields desired bulk properties of the bulk material with respect to `c` and `phi`. If `c` and `phi` do not match the values of the desired bulk material, the test needs to be repeated with different particle parameters until desired bulk properties are obtained.

25. OpenPLX for AGX Dynamics

OpenPLX is a high-level modeling language designed to work seamlessly with **AGX Dynamics**. It provides a structured approach to defining physics simulations, allowing users to create and configure rigid bodies, constraints, motors, friction models, and more in a human-readable format.

By leveraging **AGX Dynamics**, OpenPLX enables the simulation of complex mechanical systems, vehicles, robotics, and deformable terrains with minimal effort.

Note

OpenPLX is a work in progress. This draft version will evolve with user feedback and experience. We welcome your input and collaboration.

This page explains how OpenPLX interacts with AGX Dynamics and provides a structured way to define physics simulations. For a complete overview of OpenPLX Syntax and Bundles, consult the OpenPLX Documentation.

Note

This version of AGX (2.41.0.0) is built with OpenPLX version 0.22.9. OpenPLX release log.

25.1. How OpenPLX Works with AGX Dynamics

OpenPLX acts as an *interface layer* for AGX Dynamics, converting structured OpenPLX definitions into AGX simulation components. When you load an OpenPLX file, it is transformed into an **AGX Dynamics simulation tree**, allowing for efficient physics-based interactions.

This means:

1. OpenPLX **describes** the physics setup and the scene.
2. The **AGX Dynamics engine** executes the simulation.
3. Users can interact with the scene through **OpenPLX files**.
4. **Python controllers** interact by sending or receiving OpenPLX signals.

25.2. Simulating an OpenPLX File in AGX Viewer

To simulate an *.openplx* file with AGX Dynamics, pass it to *agxViewer* like you would with an *.agx* file:

```
agxViewer myModel.openplx
```

25.3. Importing an AGX File in OpenPLX

AGX *.agx* files contain pre-defined physics models. OpenPLX supports seamless integration of AGX models into simulation systems.

1. The AGX Model describes the **basic physics setup**, including rigid bodies, constraints, collision geometries and visuals.
2. OpenPLX imports the AGX model and augments it with **additional components**, such as tracks and a drivetrain.
3. OpenPLX describes the **complete simulation scene**.
4. Controller scripts (python/C++) connects the OpenPLX model to the desired **control systems**.
5. The AGX Dynamics engine **executes the simulation**.

25.3.1. Syntax for Importing an AGX File

To import an *.agx* file into OpenPLX, use the following OpenPLX syntax:

```
import @"model.agx" as AGXModel

Scene is Physics3D.System:
  my_model is AGXModel
```

The imported AGX model (*AGXModel*) can now be referenced within OpenPLX.

To load the OpenPLX scene and view the AGX model, use:

```
agxViewer scene.openplx -p
```

25.3.2. Viewing an AGX File as an OpenPLX Model

You can open an AGX file to find the components you need to reference using the **anytoopenplx** command:

```
python -m agxPythonModules.openplx.anytoopenplx model.agx > model_reference.openplx
```

Use this file to identify signals and components to reference when augmenting your model with additional OpenPLX components.

Note

At the moment, the models generated OpenPLX file will duplicate the content of the agx file, creating a conflict when running the simulation. To accomodate this, change the file extension of the new .openplx file to .txt, or place it outside the root folder of your OpenPLX bundle.

25.4. OpenPLX Tutorials

Explore the OpenPLX Tutorials to get started with OpenPLX for AGX Dynamics.

25.5. AGX Annotations

Table 25.1 AGX Parameters

Annotation	Description	Type
agx_relaxation_time	Directly set the AGX Dampling. Applies to: `Physics3D.Interactions.Mate`	Real
agx_solve_type	Set the AGX Solve type for a constraint. Applies to `Physics3D.Interactions.Mate` Accepted values `"DIRECT" | "DIRECT_AND_ITERATIVE" | "ITERATIVE"`	String
agx_friction_solve_type	Sets the AGX Solve type for the fricion model. Applies to `Physics.Interactions.Dissipation.DefaultDissipation` Accepted values `"SPLIT" | "DIRECT" | "DIRECT_AND_ITERATIVE" | "ITERATIVE"`	String
agx_approximate_cone_friction	Enables AGX approximate cone friction.	Bool
agx_terrain_material	Sets an AGX terrain material preset. Applies to `Terrain.TerrainMaterial`	String
agx_actuator_internal_inertia	Sets the internal inertia of the input shaft of a agxPowerLine::RotationalActuator or the internal mass of the input rod of a agxPowerLine::TranslationalActuator. Applies to `DriveTrain.HingeActuator` and `DriveTrain.PrismaticActuator`	Real
agx_observer_frame	Maps a `Physics3D.Interactions.MateConnector` to a agx::ObserverFrame. Applies to `Physics3D.Interactions.MateConnector`	Bool
agx_debug_render_frame	Renders the axes of a `Physics3D.Interactions.MateConnector`, also maps it to a agx::ObserverFrame. Applies to `Physics3D.Interactions.MateConnector`	Bool

25.6. AGX Track Annotations

The following annotations can be applied to a Vehicles.Tracks.System to set AGX internal track properties.

Table 25.2 AGX Track Annotations

Annotation	Description	Type
agx_track_node_wheel_overlap	setTransformNodesToWheelsOverlap	Real
agx_track_on_initialize_merge_nodes_to_wheels	setEnableOnInitializeMergeNodesToWheels	Bool
agx_track_on_initialize_transform_nodes_to_wheels	setEnableOnInitializeTransformNodesToWheels	Bool
agx_track_hinge_range_lower, agx_track_hinge_range_upper	setHingeRangeRange	Real
agx_track_hinge_compliance	setHingeCompliance	Real
agx_track_hinge_relaxation_time	setHingeDamping	Real
agx_track_stabilizing_friction_parameter	setStabilizingHingeFrictionParameter	Real
agx_track_min_stabilizing_normal_force	setMinStabilizingHingeNormalForce	Real

Annotation	Description	Type
agx_track_node_wheel_merge_threshold	setNodesToWheelsMergeThreshold	Real
agx_track_node_wheel_split_threshold	setNodesToWheelsSplitThreshold	Real
agx_track_num_nodes_in_average_direction	setNumNodesIncludedInAverageDirection	Int
agx_set_enable_merge	setEnableMerge	Bool
agx_set_contact_reduction	setContactReduction Accepts: `"NONE"` | `"MINIMAL"` | `"MODERATE"` | `"AGGRESSIVE"`	String
agx_set_enable_lock_to_reach_merge_condition	setEnableLockToReachMergeCondition	Bool
agx_set_lock_to_reach_merge_condition_compliance	setLockToReachMergeConditionCompliance	Real
agx_set_lock_to_reach_merge_condition_relaxation_time	setLockToReachMergeConditionDamping	Real
agx_set_num_nodes_per_merge_segment	setNumNodesPerMergeSegment	Int

25.7. AGX Track Wheel Annotations

The following annotations can be applied to a Vehicles.Tracks.RoadWheel to set AGX Track Wheel properties.

Table 25.3 AGX Track Wheel Annotations

Annotation	Description	Type
agx_track_wheel_split_segments	SPLIT_SEGMENTS	Bool

25.8. AGX Plugin Annotations

When importing a .agx file there are a few annotations that change the import behavior.

Table 25.4 AGX Plugin Annotations

Annotation	Description	Type
discard_rigid_body_positions	Discards all position information for rigid bodies if set to `true`, allowing SNAP to reposition them.	Bool
regenerate_shape_uuids	Setting this to `true` will create new UUIDs for all AGX shapes. This is useful when importing the same `.agx` file twice.	Bool
material_naming_rule	Controls how imported materials will be named. This can be necessary to avoid collisions since materials are always put in a global scope. Accepted values `"UUID"` sets the material `unique_name` to the AGX UUID. `"OPENPLX_NAME"` sets the material `unique_name` to the OpenPLX name. `"AGX_NAME"` (default) sets the material `unique_name` to the AGX name. If UUIDs are not unique, try `"OPENPLX_NAME"` together with a namespace to generate unique names.	String
ignore_disabled_secondary_constraints	Secondary constraints that are disabled are not imported if this is set to `true`. Can improve performance when importing the AGX file.	Bool

26. OpenPLX → AGX: Mapping Guide

26.1. Scope

A guide showing how each OpenPLX types and attributes end up in AGX.

26.2. Core object / ID / transform mapping

OpenPLX	AGX	Notes
`Object::getName()`	`agx::Name`	The name of an object in AGX is generally the full OpenPLX name including all namespaces with dot separation.
`.uuid`	`getUuid()`	A .uuid: annotation in an OpenPLX object mapped to AGX will enforce that to the created AGX object. If no annotation exist AGX will generate a new uuid.
`Math::AffineTransform`	`agx::AffineMatrix4x4`	`mapLocalTransform` writes the local transform on assemblies, bodies and geometries.
`System` / `Body` / `Geometry`	`agxSDK::Assembly` / `agx::RigidBody` / `agxCollide::Geometry`	The OpenPLX system tree maps to an assembly tree in AGX. Systems include bodies and geometries. Bodies include geometries.
`MateConnector`	`agx::Frame` / `agx::ObserverFrame`	Pose computed from main axis/normal/position; observer frames preserved for AGX-imported connectors.

26.3. Bodies & mass properties

OpenPLX `Physics3D::Bodies::RigidBody`	AGX `agx::RigidBody`	Notes
`Physics3D::Bodies::RigidBodyKinematics`		Kinematics declaration for rigid bodies. Container of velocities and transforms.
`Physics3D::Bodies::Inertia`		Inertia declaration for rigid bodies. Contains scalar mass and inertia tensor.
`is_dynamic is Bool`	`RigidBody::MotionControl`	is_dynamic: True → DYNAMICS; otherwise KINEMATICS.
`inertia.mass()`	`MassProperties::setMass`	Negative mass → error. Auto-generate if not given.
`inertia.tensor()`	`MassProperties::setInertiaTensor`	CM rotation applied to tensor; SPD validation.
`kinematics.local_transform`	`RigidBody::getFrame()->setLocalMatrix`	Local transform relative owning system.
`kinematics.local_cm_transform`	`RigidBody::getCmFrame()->setLocalMatrix`	Local CM transform relative the local transform. In AGX the rotation must match model frame. Keep identity rotation!
`kinematics.initial_local_linear/angular_velocity`	`setVelocity` / `setAngularVelocity`	The local OpenPLX velocities will be transformed to world in AGX.

26.4. Geometry (contact shapes & attributes)

OpenPLX	Attributes	AGX mapping	Notes
`ContactGeometry`	`material is Physics::Geometries::Material`, `local_transform is Math.AffineTransform`, `enable_collisions is Bool`, `include_in_mass_properties is Bool`	`agxCollide::Geometry` with one `agxCollide::Shape`	An OpenPLX ContactGeometry maps to an agx::Geometry with a single shape.
`Box`	`size is Math::Vec3`	`agxCollide::Box`	Size mapped with 0.5 factor per axis for half-extents.
`Cylinder`	`radius is Real`, `height is Real`	`agxCollide::Cylinder`	The Y axis is the default cylinder axis in AGX.

OpenPLX	Attributes	AGX mapping	Notes
`Capsule(r,h)`	`radius is Real`, `height is Real`	`agxCollide::Capsule`	The Y axis is the default capsule axis in AGX.
`Sphere`	`radius is Real`	`agxCollide::Sphere`	
`ConvexMesh`	`vertices is Math.Vec3[]`	`agxCollide::Convex`	Built via `agxUtil::ConvexReaderWriter`.
`TriMesh`	`vertices is Math.Vec3[]`, `indices is Int[]`	`agxCollide::Trimesh`	Built via `TrimeshReaderWriter`.
`ExternalTriMeshGeometry`	`path is String`, `scale is Math.Vec3`	`agxCollide::Trimesh`	Path must be absolute; scale → diagonal of `Matrix3x3`

26.5. Materials & surface contact

OpenPLX	AGX	Notes
`Physics::Geometries::Material`	`agx::Material`	Name/UUID; density → `BulkMaterial::setDensity`.
`LinearElasticMaterial`	`Material::Bulk::setYoungsModulus`	A material model that models Young's modulus.

OpenPLX	Alternatives	AGX mapping	Notes
`SurfaceContact.Model`	Configurable with attributes listed below.	`agx::ContactMaterial`	Created/uniqued per material pair in AGX. , `tangential_restitution` and `normal_restitution`.
`normal_flexibility`	`LinearElastic`, `Rigid`, `PatchElasticity`, `PointwiseElasticity`	`stiffness` attribute of `LinearElastic` maps to `agx::ContactMaterial::setYoungsModulus`	`PatchElasticity` enables "contact area approach".
`dissipation`	`MechanicalDamping`, `ConstraintRelaxationTimeDamping`	`ContactMaterial::setDamping`	`ConstraintRelaxationTimeDamping` maps directly to Spook damping in AGX. `MechanicalDamping` = `stiffness` * `Spook damping`
`friction`	`DryConeFriction`, `DryBoxFriction`, `DryScaleBoxFriction`, `DryConstantNormalForceFriction`, `DryDepthScaledConstantNormalForceFriction`.	`agx::FrictionModel`	`agx_friction_solve_type` & `agx_approximate_cone_friction` annotations respected; `Traits.OrientedFriction` in the Physics3D bundle need `reference_body`/`reference_geometry` + `primary_direction`.
`clearance`	`ConstantDistanceClearance` and `ConstantAngleClearance`	`ConstantDistanceClearance` maps to `adhesiveOverlap` in `SurfaceMaterial::setAdhesion(force, adhesiveOverlap)`	Clearance also maps to slack in a SlackHinge, SlackPrismatic.
`normal_restitution`	A non zero scalar value.	`setRestitution`	Restitution in the direction of a contact normal.
`tangential_restitution`	A non zero scalar value.	`setTangentialRestitution`, OpenPLX value will be set in both primary and secondary direction.	Restitution orthogonal to the contact normal. Not mapped to support a different restitution in the secondary direction.

26.6. Interactions & constraints

An `agx::Constraint` is created using frames computed from its mate connectors and their respective `agx::RigidBody`. Each DOF's **compliance** and **damping** (relaxation time) is set from the corresponding OpenPLX flexibility/dissipation component (see section below). Friction controllers on supported constarints are configured from PLX friction.

> `friction` may become `DefaultDryFriction`, `AsymmetricLimitFriction` or `ConstantLimitFriction`. `AsymmetricLimitFriction` and `ConstantLimitFriction` will both set `agx::FrictionController::setMinimumStaticFrictionForceRange`.
>
> Names/UUIDs always propagated.

OpenPLX	Notes on Attributes	AGX mapping	DOFs & extras
`Physics3D::Interactions::Hinge`	Slack variant if `clearance` becomes `ConstantHingeClearance`.	`agx::Hinge` or `agx::SlackHingeJoint`	Free to rotate around the `main_axis`
`Physics3D::Interactions::Prismatic`	Slack if `clearance` becomes `ConstantPrismaticClearance`	`agx::Prismatic` or `agx::SlackPrismaticJoint`	Free to move along the `main_axis`
`Physics3D::Interactions:::Cylindrical`	`agx::CylindricalJoint` or `agx::SlackCylindricalJoint`	Slack variant if `clearance` becomes `ConstantCylindricalClearance`	Free to move along and rotate around the `main_axis`
`Physics3D::Interactions:::Ball`	No slack version. No mapping of ranges implemented.	`agx::BallJoint`	Free to rotate around all axes.
`Physics3D::Interactions:::Lock`	Slack variant if `clearance` becomes `ConstantLockClearance`;	`agx::LockJoint` or `agx::SlackLockJoint`	Slack if `ConstantLockClearance`.
`Physics3D::Interactions::Distance`	No slack version. No mapping of distance range or distance motor implemented.	`agx::DistanceJoint`	Constrained to keep the distance between the mate connectors, otherwise free to move.
`Physics::KinematicLock(bodies)`	Add all rigid bodies you like to merge to the `bodies` array of the KinematicLock.	`agx::MergedBody`	Bodies merged via `agx::MergedBody::EmptyEdgeInteraction`.

26.7. Regularization parameters

Most OpenPLX interactions, including mates and surface contact models, have attributes for `flexibility` and `dissipation` which will map to `compliance` and `damping` in AGX. The OpenPLX models are designed so that if the attributes does not become anything, the parameters will get the default AGX values. The attributes are unique for each mate, listing one parameter for each degree of freedom (DOF). Generally the local axes of a mate use the naming convention of the mate connectors, which are main, normal and cross. Each regularization parameter is defined either around or along one of these axes.

- **flexibility** alternatives:
  - `Rigid` → `compliance` set to machine-epsilon (nearly rigid).
  - `LinearElastic(stiffness)` → `compliance = 1/stiffness`. `stiffness == 0` → kill DOF (force range `0`).
- **dissipation** alternatives:
  - `ConstraintRelaxationTimeDamping` → maps directly to the AGX Spook damping parameter. `agx::Constraint::setDamping`.
  - `MechanicalDamping(damping)` which requires `flexibility` to be `LinearElastic(stiffness)` → `spook_damping = damping/stiffness`.

26.8. Controllers: motors, ranges, springs (1-DOF controller constraints)

OpenPLX interactions also include springs, motors and ranges. These map to AGX controller constraints. Each controlling one degree of freedom with the possibility to declare regularization parameters.

OpenPLX	AGX	Notes
`RotationalVelocityMotor` / `LinearVelocityMotor`	`agx::TargetSpeedController` via `SingleControllerConstraint1DOF`	`gain > 0` → `compliance = 1/gain`; `target_speed`; enabled flag; force range from `min_effort, max_effort`; `zero_speed_as_spring` locks at zero and applies spring flex/dissipation.
`TorqueMotor` / `ForceMotor`	Maps to `agx::TargetSpeedController` in AGX.	Controlled in runtime by updating the force range with required_effort = min = max.

OpenPLX	AGX	Notes
`RotationalRange` / `LinearRange`	`agx::RangeController`	Range `[start,end]`, force range, enable, `flexibility` and `dissipation` attributes.
`TorsionSpring` / `LinearSpring`	`agx::LockController`	Position/angle set; force range; enable; `flexibility` and `dissipation` attributes.
Annotation `agx_imported_controller_constraint`	Reuse existing AGX constraint's motor/range/lock	Present when importing an .agx file, looks up constraint by UUID and attaches controller to the correct DOF.
Annotation `agx_solve_type` for an OpenPLX Interaction	`Constraint::setSolveType`	`DIRECT` / `DIRECT_AND_ITERATIVE` / `ITERATIVE`.

26.9. Environment, collisions & signals

OpenPLX	AGX	Notes
`Simulation::Environment.gravity`	`UniformGravityField` or `PointGravityField`	Vector or center+magnitude mapped.
`Simulation::CollisionGroup`	Geometry/group assignment	Adds all bodies/geometries/systems to AGX groups; respects nested systems.
`Simulation::DisableCollisionPair`	`space->setEnablePair(g1,g2,false)`	Applied recursively in `disableCollisionPairs`.
`Physics3D::Signals::MateConnector::Output`	`ObserverFrame(s)`	Ensures frames exist for source and relative connectors (world excluded).

26.10. Friction on constraints (non-contacts)

OpenPLX	AGX	Notes
`Interactions::*::friction()`	`Constraint…::getFrictionController`	Dry/limit frictions converted to coefficient or static force ranges; controller disabled if unsupported in PLX side.

26.11. Annotations reference

- **Controller reuse**: `agx_imported_controller_constraint` (UUID of an AGX constraint to attach to).
- **Constraint solve**: `agx_solve_type` (`DIRECT`, `DIRECT_AND_ITERATIVE`, `ITERATIVE`).
- **Friction solve**: `agx_friction_solve_type` (`DIRECT`, `SPLIT`, `DIRECT_AND_ITERATIVE`, `ITERATIVE`) + `agx_approximate_cone_friction` (boolean).
- **Oriented friction**: `reference_body` / `reference_geometry` + `primary_direction (Vec3)`.
- **Terrain material**: `agx_terrain_material` (string).
- **Observer frames**: `agx_observer_frame_annotation` / `agx_debug_frame_annotation` on MateConnectors.
- **Track-specific**: `agx_track_*`, `agx_set_*` (see Tracks table).

26.12. Notes on error handling (high-level)

Invalid/duplicate material pairs, bad inertia tensors, negative mass, missing connected bodies, misplaced connectors, invalid OBJ/Trimesh, invalid friction frame, etc., are reported via the provided `ErrorReporter` and guarded with warnings/fallbacks.

27. AGX Util

AGXUtil is a namespace that is part of the agxPhysics shared library. AGX Util provides different utility functions and classes.

27.1. Jump Request

The method `agxUtil::jumpRequest` provides a way to move a collection of items, including wires, that are part of an `agxSDK::Assembly`.

```
/**
 This method will perform a "jump request" will all bodies in the assembly,
 including bodies which are part of wires. parentBody is the main body
 which will work as a parent, all other bodies will move relative to this
 rigid body.
 The transform parentBodyWorldTransform defines the new transform for
 the parent body, the target transform for parentBody.
 Also, make sure parentBody is part of the assembly collection

 \param collection - An Assembly which contain all bodies (including parentBody) and wires.
 \param parentBody - The main rigid body, all other bodies will move relative to this body
 \param parentBodyWorldTransform - The new requested transformation for the parentBody.
 \param wireOptions - options how to handle wire nodes
 \return the number of bodies that where moved, including parentBody
 */
 size_t jumpRequest( agxSDK::Assembly* collection,
                     agx::RigidBody* parentBody,
                     const agx::AffineMatrix4x4& parentBodyWorldTransform,
                     agx::UInt32 wireOptions = agx::UInt32( 0 ) );
```

The `wireOptions` allows for specifying how wire nodes should be handled when the node parent is not part of the collection:

- Contact nodes can be detached and turned into lumped nodes instead.
- Eye nodes can be changed into lumped nodes.

When this method executes, the items in collection will be updated.

27.2. Sphere-Mesh Skeletonisation

Under the `agxUtil` namespace lies the helper function `agxUtil::skeletoniseMesh`:

```
/**
Utilises the SphereSkeletoniser class to generate a SphereSkeleton from a given
trimesh structure.
\param vertices The vertices of the mesh
\param indices The indices of the mesh (always 3 per triangle, see agxCollide/Trimesh.h)
*/
SphereSkeleton skeletoniseMesh(const agx::Vec3Vector& vertices,
                               const agx::UInt32Vector& indices);
```

and a class `agxUtil::SphereSkeletoniser` which both allow using a trimesh to generate a skeleton of a mesh. This skeleton can then be traversed directly or processed using additional methods. It can be segmented using `agxUtil::SphereSkeleton::segmentSkeleton` or be processed to produce the longest path using `agxUtil::getLongestContinousPath`.

The main intended use of these functions are to aid the the creation of routes for `agxCable` and `agxWire` instances and an example of this can be seen in `example_mesh_to_cable.cpp` (here seen simplified):

```
auto cable = new agxCable::Cable(radius, resolution);
for (SphereSkeleton::dfs_iterator it = skeleton.begin(); !it.isEnd(); ++it)
    cable->add(new agxCable::FreeNode(it->position));
simulation->add(cable);
```

The quality of the skeletons produced relies heavily on the quality of the input mesh and as such it is recommended to use meshes with evenly distributed vertices along the meshes surface and to use simple shapes for the best results. Other meshes still produce results but some experimentation using the parameters available in `agxUtil::SphereSkeletoniser`, post-processing by hand, or using the processing functions may be necessary.

Note

AGX uses *Assimp* for importing meshes and by default allows splitting vertices for both normals and UV-coordinates. This behaviour can create non-manifold meshes so to ensure this doesn't affect the skeletonisation, we advise to use the *REMOVE_DUPLICATE_VERTICES* flag when importing meshes for this purpose.

28. AGX Model

agxModel is a library consisting of high level primitives which implement various simulation objects: trees, beams, terrain etc. They are built upon the basic construction elements found in the agxPhysics library: constraints, geometries, bodies and event listeners.

28.1. Granular Fields

A granular field is a physical quantity computed from discrete granular bodies. For example, in a silo filled with granular material, the mass field for each point in the silo can be computed from the discrete particles. In AGX, fields are diszretized into voxels, each voxel represents one tensor element. Each field element is computed from the particles that intersect the corresponding voxel. A particle can be in several voxels. In that case, a weight proportional to the intersecting volume is calculated.

28.1.1. Basic usage

Computation of any field is done by `agxModel::GranularFieldPostProcessor`. When creating the post-processor a cuboid volume is placed and discretized into equally sized voxels. This volume defines the boundary of the field. Particles within this boundary contributes to the field, particles outside of it is ignored. The smallest side length of one voxel is **not** allowed to be smaller than the diameter of the largest granular body. In that case the particle weight calculation will be wrong.

The example below creates a cuboid volume with side length `1.0` that is divided into $3 * 3 * 3 = 27$ voxels.

```
agx::Physics::GranularBodySystemRef gbs = new agx::Physics::GranularBodySystem();

// Create a granular post-processor
agx::Vec3u shape(3, 3, 3);
agx::Vec3 volume(1.0);
agxModel::GranularFieldPostProcessorRef gpp = new agxModel::GranularFieldPostProcessor(gbs, shape, volume);
```

After creating the post-processor, add the types of fields it is supposed to compute. Some fields depend on other fields. If such a field is added it will also add the type of field it depends on.

```
// Add the mass field
gpp->addField(new agxModel::GranularMassField());
```

The fields are computed by calling `gpp->calculateFields()`. All the fields added to the post-processor is then computed and saved in the granular field buffers. To access the computed granular field one can do

```
// Get the pointer to the computed mass field
agx::Real *data = gpp->getField(agxModel::GranularField::MASS)->getFieldData();
```

28.1.2. Fields

The existing fields are described below.

28.1.2.1. Mass field

The mass field in voxel V_{ijk} is given by

$$m_{ijk} = \sum_{p \in V_{ijk}} m_p w_p$$

where m_p is the mass of particle p and w_p is the weight associated with that particle and voxel V_{ijk}.

28.1.2.2. Momentum field

The momentum field in voxel V_{ijk} is given by

$$\bar{p}_{ijk} = \sum_{p \in V_{ijk}} m_p \bar{v}_p w_p$$

where m_p is the mass of particle p, v_p is the velocity of particle p and w_p is the weight associated with that particle and voxel V_{ijk}.

28.1.2.3. Velocity field

The velocity field in voxel V_{ijk} is given by

$$\bar{v}_{ijk} = \frac{\bar{p}_{ijk}}{m_{ijk}}$$

where m_{ijk} is the mass field in voxel ijk and p_{ijk} is the momentum-field in voxel V_{ijk}. The velocity field depends on both the mass field and the momentum field.

28.1.2.4. Custom field

It is possible to add custom real buffers to the `GranularBodySystem`. Using these buffers it is possible to calculate a custom field. For example, if each particle is assigned an unique concentration value using a custom buffer. One can then use this custom buffer to calculate a concentration field. It is important that the custom field get the name of the custom buffer.

```
agx::Physics::GranularBodySystemRef gbs = new agx::Physics::GranularBodySystem();
gbs->addCustomBufferReal("concentration");

// ....

// Create a granular post-processor
agx::Vec3u shape(3, 3, 3);
agx::Vec3 volume(1.0);
agxModel::GranularFieldPostProcessorRef gpp = new agxModel::GranularFieldPostProcessor(gbs, shape, volume);

// Add the custom buffer field. Use the same name ass the previously created buffer.
gpp->addField(new agxModel::GranularCustomBufferField("concentration"));
```

The custom field in voxel V_{ijk} is given by

$$c_{ijk} = \sum_{p \in V_{ijk}} c_p w_p$$

where c_p is the custom buffer value for particle p and w_p is the weight for particle p in voxel ijk.

28.2. SuctionGripper

The general idea of the `agxModel::SuctionGripper` is to model a gripper tool for robotics, where one or more `agxModel::SuctionCup` s are powered by a `agxModel::VacuumSystem`.

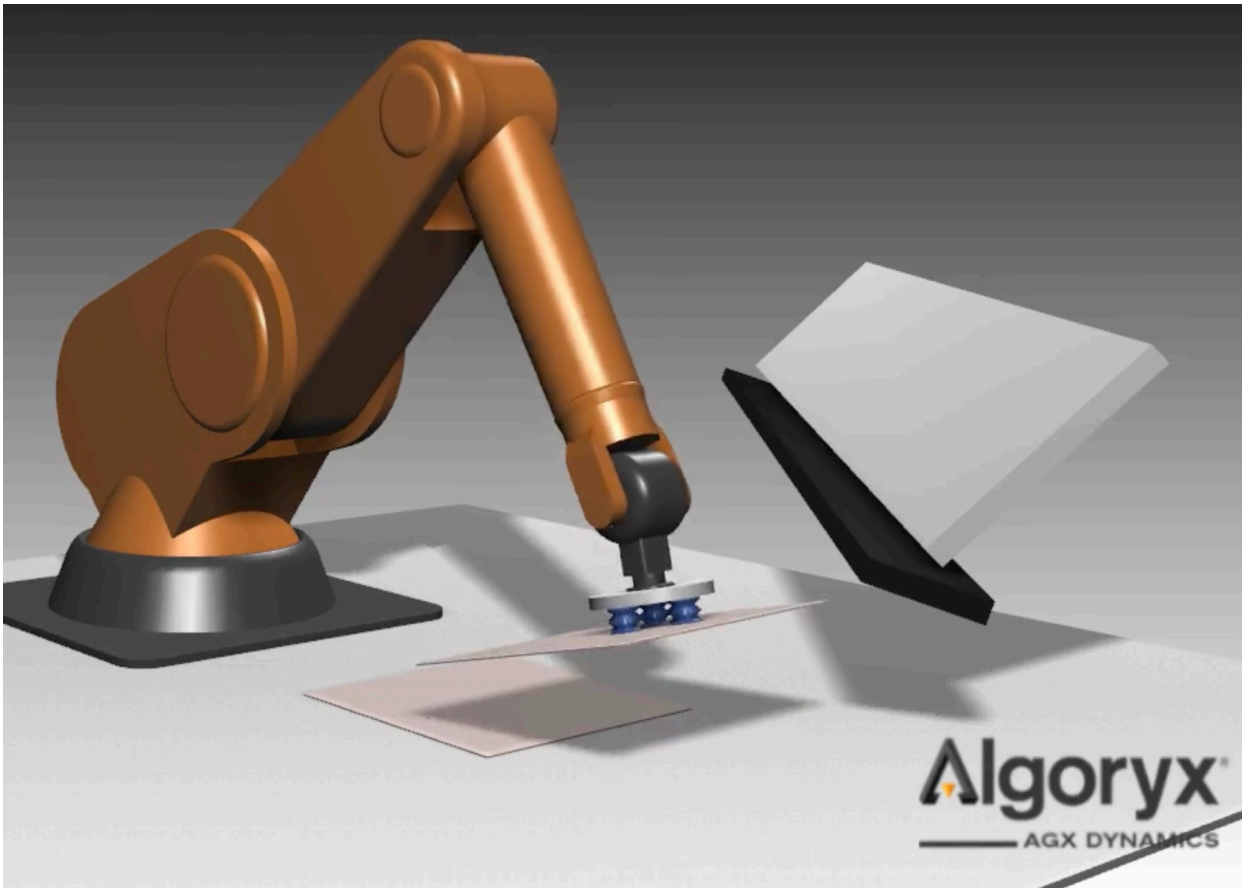


Fig. 28.1 Image capture from *data/python/tutorials/tutorial_custom_suction_gripper.agxPy*

The `agxModel::SuctionGripper` is based on the `agxSDK::Assembly` class which makes it possible to model the relative suction cup positions in a local coordinate system.

The SuctionGripper requires a internal holder rigid body to be set. Its purpose is for mounting suction cups and for connecting the gripper to, for example, a robot.

```
// Create SuctionGripper
agxModel::SuctionGripperRef suctionGripper = new agxModel::SuctionGripper();
// Create holder rigid body, which suction cups should be attached to
agx::RigidBodyRef holderBody = new agx::RigidBody("MyHolderBody");
suctionGripper->setHolderRigidBody(holderBody);
```

The heart of the SuctionGripper implementation lies in the collision detection handling for each individual suction cup and their one or more sensors. Each cup interaction will generate SuctionCupInteraction's which include information for a VacuumSystem algorithm to base its vacuum force distribution.

It is possible to inherit from the `agxModel::VacuumSystem`, `agxModel::SuctionCup` or the `agxModel::SuctionCupInteraction` to customize the default behaviour.

```
class MyVacuumSystem : public agxModel::VacuumSystem {
  // ..
};
// Replace VacuumSystem
suctionGripper->setVacuumSystem(new MyVacuumSystem());
```

28.2.1. Vacuum System

The purpose of the vacuum system is to maintain a desired vacuum for all SuctionCup's for one SuctionGripper. The desired vacuum level is set:

```
// Create vacuum system with desired vacuum level at 20% vacuum (80% of atmospheric pressure)
agx::Real desiredVacuum = agx::Real(0.2)
agxModel::VacuumSystemRef vacuumSystem = new agxModel::VacuumSystem(desiredVacuum);
// The desired vacuum can't be >= 1 (100% is perfect vacuum)
vacuumSystem->setDesiredVacuum(agx::Real(0.4));
```

The internal state for the VacuumSystem can be tuned in the following way:

```
// Disable the pump (enabled by default).
vacuumSystem->setEnablePump(false);

// Set the outside pressure in Pascal (atmospheric pressure by default)
agx::Real pressureOnMars = agx::Real(610);
vacuumSystem->setOutsidePressure(pressureOnMars);

// Set the outside gas density (by default set to 1.225 [kg / m^3])
agx::Real gasDensityOnMars = agx::Real(0.02);
vacuumSystem->setOutsideGasDensity(gasDensityOnMars);
```

For a custom VacuumSystem there are two virtual functions to consider implementing.

```
// This virtual function is responsible for calculating the actual force a suction cup can apply to objects, given the
current situation.
// Could consider for example current leakage near the cup.
virtual agx::Real MyVacuumSystem::calculateAvailableVacuumForce(agxModel::SuctionCup* cup) override {

// This virtual function for updating the vacuum system.
// Which allow for a dynamic vacuum system.
// The default system however will toggle between desired vacuum and zero vacuum, depending on if the pump is enabled.
virtual void MyVacuumSystem::step(const agx::Real& dt, const SuctionGripper* gripper) override {
```

28.2.2. Suction Cup Interaction

The data contained in a SuctionCupInteraction is only valid throughout the time step. The after the collision detection stage in AGX one SuctionCupInteraction is activated for each rigid body the suction cup sensors has collided with. The API available for the suction cup interactions are the

following:

```
// Directly after the collision detection we can go through the active interactions of one suction cup
for( auto interaction : cup->getActiveInteractionsVector() ) {

  // Interacting rigid body which will be affected by the vacuum force.
  agx::RigidBody* rigidBody = interaction->getRigidBody();

  // How many seal sensors the rigid body is interacting with
  size_t sealCount = interaction->getSealCount();

  // True if the object collides with the line sensor, located along the suction cup normal.
  bool centered = interaction->getCentered();

  // The distance to the closest point on the colliding rigid body along the center vector.
  agx::Real centerDistance = interaction->getCenterDistance();

  // The surface normal at the closest point of the colliding rigid body.
  agx::Vec3 surfaceNormal = interaction->getCenterNormalWorld();

  // The world position where the vacuum force will apply a wrench.
  agx::Vec3 worldForcePosition = interaction->getFoundSurfacePositionWorld();
}
```

After AGX has stepped the DynamicsSystem, forces and velocities are updated, and the following API calls are valid for examining the state of the interaction

```
// Directly after the collision detection we can go through the active interactions of one suction cup
for( auto interaction : cup->getActiveInteractionsVector() ) {

    // Last scalar vacuum force applied to the rigid body of the interaction
    agx::Real lastScalarForce = interaction->getCalculatedScalarForce();

    // Last directional vacuum force applied to the rigid body of the interaction
    agx::Vec3 lastDirectionalForce = interaction->getCalculatedForce();

    // Sum of all friction forces between the suction cup and the rigid body
    agx::Vec3 lastFrictionForce = interaction->getCurrentFrictionForce();

    // The total normal force between the suction cup and the rigid body
    agx::Vec3 normalForce = interaction->getCurrentContactNormalForce() const;

    // The maximal relative velocity between the suction cup and the rigid body considering all contact points.
    agx::Real relativeVelocity = interaction->getCurrentRelativeVelocity();
}
```

For a custom SuctionCupInteraction there is one virtual function to consider implementing.

```
class MySuctionCupInteraction : public agxModel::SuctionCupInteraction {
  // ..
};
// Virtual function responsible for updating the actual forces to apply to the interacting rigid body
// by calling this->setCalculatedScalarForce() and this->setCalculatedForce()
virtual agx::Real MySuctionCupInteraction::updateVacuumForce(const SuctionCup* cup, const VacuumSystem* vacuumSystem,
const agx::Real& fraction) override {
```

28.2.3. Suction Cup

The SuctionCup require one rigid body to be valid. The SuctionCup comes with built in handling of two types of sensors, seal (Spheres) and Line. The purpose of the sensors are to detect nearby geometries of dynamic rigid bodies and create SuctionCupInteraction's to contain all important information for vacuum forces to be computed.

The Line sensor is attached to each suction cup by default. The suction cup can be equipped with seal sensors, which help detecting if the interaction is sealed or not and will help defining a leakage area. It is up to the ::calculateAvailableVacuumForce function of the vacuum system to consider using a leakage area or not.

The local x-y plane for the suction cup rigid body define the suction cup plane. If seal sensors are used these are placed in this plane at lipRadius distance from the body origin. It is expected that the user constrains the cup body to the holder body, but it is not forced in any way.

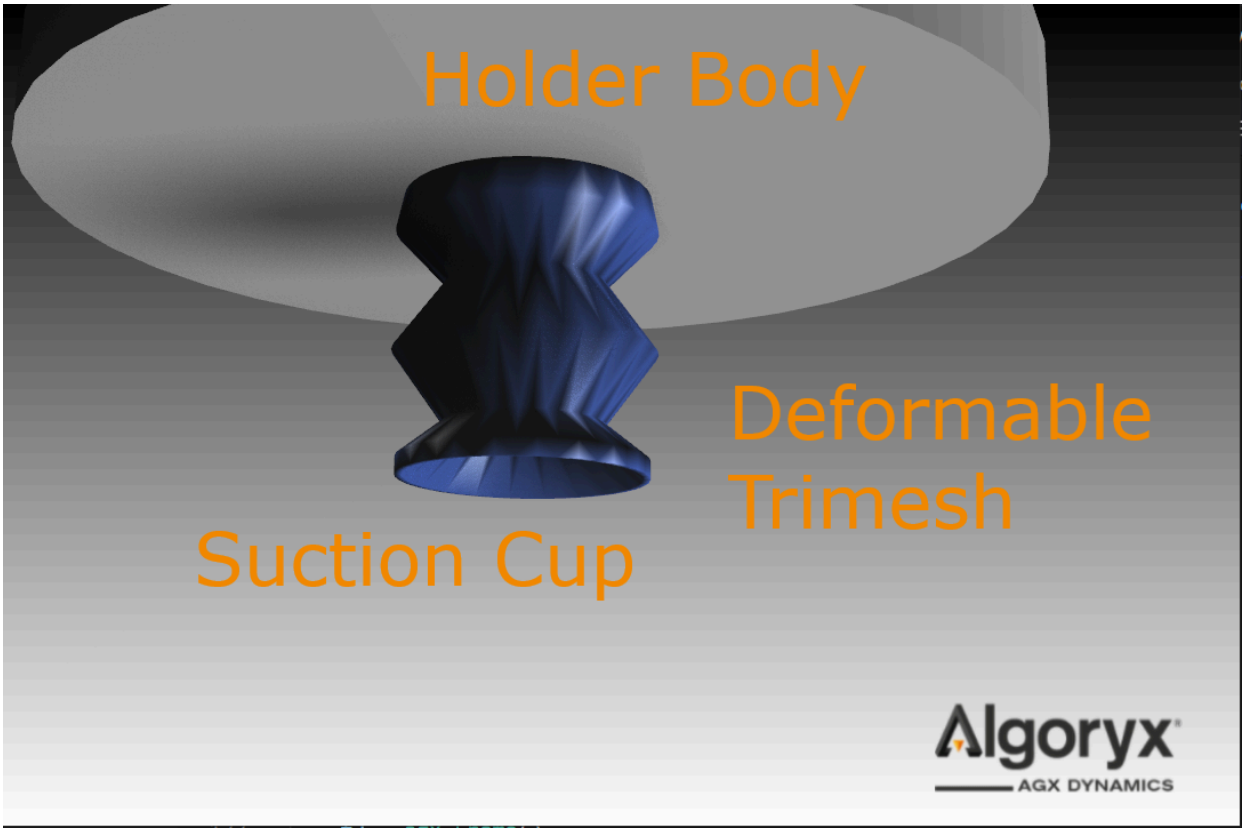


Fig. 28.2 The suction cup attached to the holder

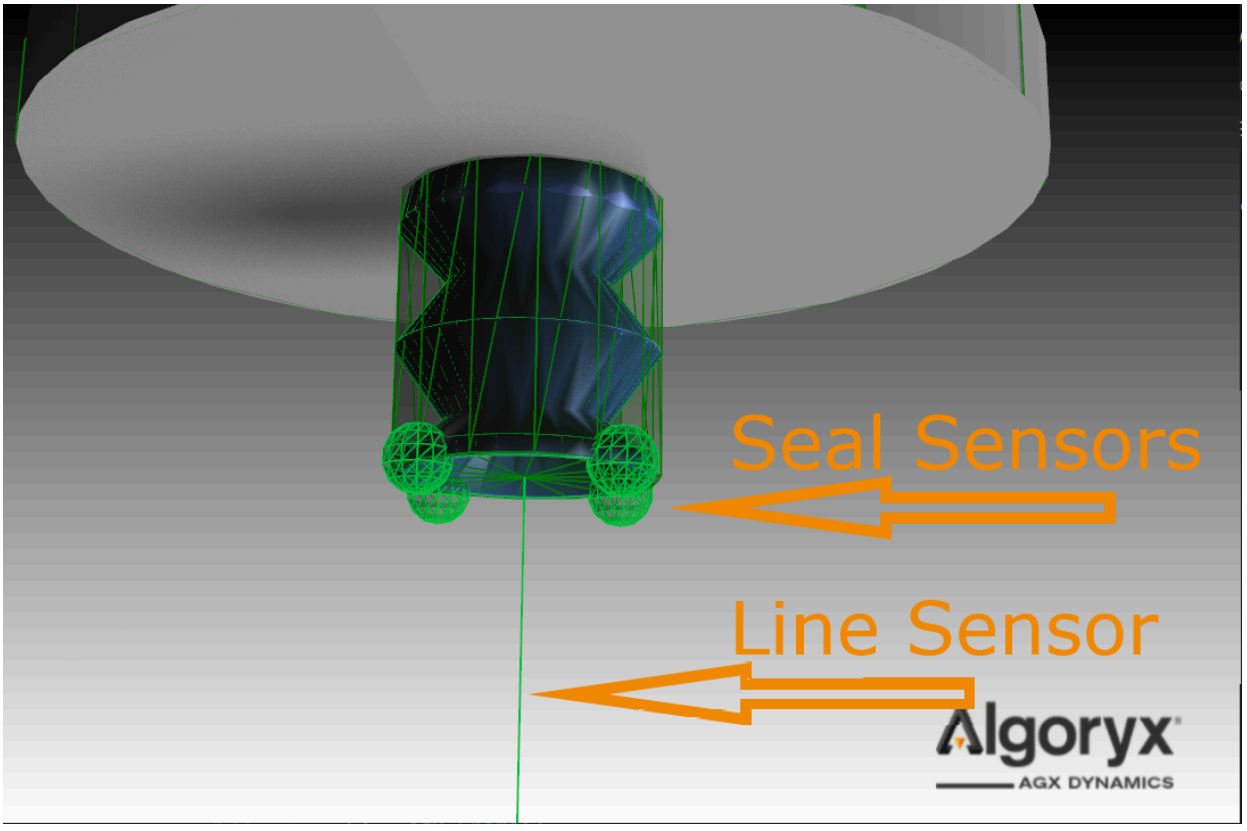


Fig. 28.3 Four seal sensors positioned in the suction cup local plane, at lip radius distance from the center.

If no rigid body is added to the constructor, a default rigid body is created. The default rigid body will get a cylindrical collision geometry using the lipRadius and lipHeight for dimensions. All suction cup rigid bodies will by default be given a `agxCollide::Line` shape geometry sensor. There are multiple ways to customize the suction cup on creation:

```
// Go with the default body
agx::RigidBody* cupBody = nullptr;
// Lip radius, which define the area for the cup interaction, and therefore the maximum vacuum force
agx::Real lipRadius = agx::Real(0.025);
// Lip height, the height of the default collision geometry.
agx::Real lipHeight = agx::Real(0.01);
// How many spherical sensors you like on the circumference of the lip.
size_t sealResolution = size_t(4);
// Reach of the line sensor
agx::Vec3 lineSensorVector = agx::Vec3(0,0,-0.1);
// Radius of the seal senors spheres
agx::Real sealSensorReach = agx::Real(0.01);
agxModel::SuctionCupRef cup = SuctionCup( cupBody, lipRadius, lipHeight, sealResolution, lineSensorVector,
sealSensorReach);
```

There is one variable to customize during runtime:

```
// Define a mounting radius > 0, defaults to half of the lip radius
// Will effect the pressure drop at the lip when the lip is not sealed.
agx::Real mountingRadius = 0.01;
cup->setMountingRadius(mountingRadius);
```

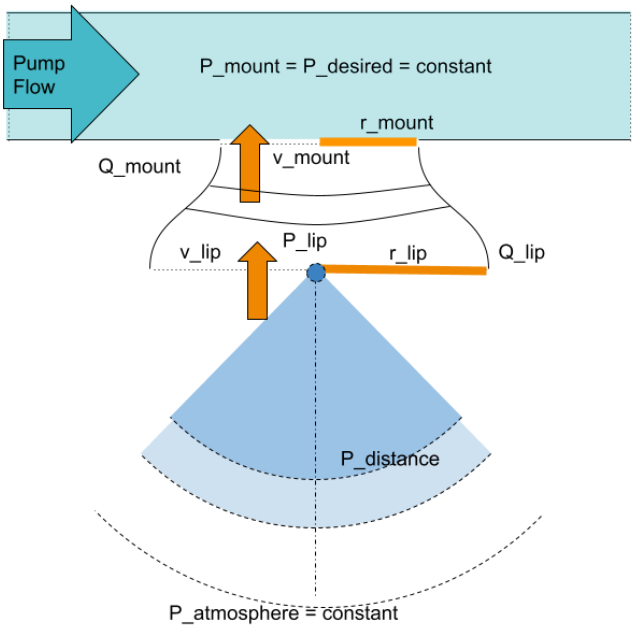


Fig. 28.4 Describe the vacuum system and how the pressure at the lip and nearby is modeled.

The default Vacuum system model a constant vacuum at the mount. The pressure at the lip P_l when there is leakage dropps according to Bernoulli's principle, assuming constant flow $Q_l = Q_m$ and zero air speed far away from the lip.

We define the velocity relation between the air speed at the mount v_m and the air speed at the lip v_l.

$$
\begin{aligned}
a_m &= \pi r_m^2 \\
a_l &= \pi r_l^2 \\
Q_l &= v_l \, a_l \, dt \\
Q_m &= v_m \, a_m \, dt \\
v_l &= \frac{v_m \, a_m}{a_l}
\end{aligned}
$$

We find the air velociy at the mount from Bernoulli's principle given the pressure at the mount P_m, air density ρ_{air}, atmospheric pressure P_a and assuming zero air velocity far away from the cup.

$$
P_a = P_m + \frac{\rho_{air} \, v_m^2}{2}
$$

Pressure drop at the lip then comes from the calculated air speed at the lip:

$$
P_l = P_a - \frac{\rho_{air} \, v_l^2}{2}
$$

The pressure gradient decay quadratically relative to the distance from the suction cup center. This from assuming the flow is constant at every imaginary half sphere away from the cup. The forces due to vacuum will be applied to the objects identified by the cup sensors as well as an opposite reaction force will be applied to the cup.

28.3. Tree

`agxModel::Tree` is a structure of Branch objects in a mathematical Tree structure where a Branch can have several child branches. Even though the class is to save a tree structure, it can be used to create a simulation of an actual tree. To use it for such things, it requires you to create your own geometries and add them to the tree structure. Tree holds functions to cut, split and in other ways change the Tree structure.

```
// Create a tree
agxModel::TreeRef tree = new agxModel::Tree();
```

A newly created Tree will be empty except for its root. The root is the base branch of the tree. It can be accessed as:

```
agxModel::Tree::BranchRef treeRoot = tree->getRoot();
```

To create a new branch, to attach to the root, we need to know how it should be rotated and where it should be attached. The easiest way to do this is to decide a start and end position of the branch. Note that the end position is only used to decide in which direction the branch should point. It is also possible to just set a transform of the branch and send it in as you create it. When the branch is created it is possible to attach it to another branch, which will then make the new branch a child to that one.

```
// Create new branch
agxModel::Tree::BranchRef newBranch = new agxModel::Tree::Branch(startPosition, endPosition);
// or
agxModel::Tree::BranchRef newBranch = new agxModel::Tree::Branch(transform);

// Attach new branch
branch->addBranch(newBranch);
```

When the branch is created, it will create a rigid body, but it has no geometries. To represent the branch it is possible to use any geometry, but only capsules and cylinders fully support all functionalities for branches by default. By doing `branch->getRigidBody()` we can get the rigid body connected to the branch. The body will be placed and rotated as specified when we created the branch. Adding geometries to the branch is done by simply adding them to the rigid body, as you would normally do.

There are a few different things we can do with a tree. One useful function `makeStatic`, which can be used to make all the rigid bodies in the tree static, and remove all constraints from the simulation.

```
tree->makeStatic(true);
```

The boolean set to true enables the tree to wake up when exposed to a contact. By default, any contact can wake it up, but if you want to specify this more explicitly, an `agxModel::Tree::TreeExecuteFilter` can be set when calling `makeStatic`.

The tree will keep being dynamic after the contact, so if you want it to go back to being static, `makeStatic` must be called again. A static tree can also be made dynamic doing:

```
tree->makeDynamic();
```

Another feature of `agxModel::Tree` is the possibility to cut branches off in an easy way. Cutting a branch means that the constraint holding the given branch to the tree is removed and the branch that is cut off will be the root of a new tree, where all its child branches are part of it. There are two ways to cut a branch.

```
agxModel::TreeRef newTree = tree->cut(branch);
// or
agxModel::TreeRef newTree = tree->cut(branchGeometry, point, minShapeHeight);
```

`cut(branch)` will cut it off at the start of the branch. `cut(branchGeometry, point, minShapeHeight)` can be used to cut a branch in the middle of it, or in any point of the given geometry. `minShapeHeight` sets the limit for how small a geometry segment can become. If a cut would resolve in a geometry shorter than `minShapeHeight`, the cut is instead made between two geometries or directly at a branch connection. By default cutting a branch geometry in the middle is however only supported by capsules and cylinders. Note that cut assumes that the geometries in a branch are ordered in position with first geometry furthest down in branch body coordinates.

To handle certain events that can happen to a tree, it is possible to use a `BranchEventListener`. This is a class that contains a few methods that you can implement. By default, they will do nothing. It contains; `onCreate`, `onHightLoad` and `clone`. `onCreate` is called when the branch has been added to the simulation and when it has been configured with position, orientation and constraint to parent (if present). `onHighLoad` is called when a branch has a load applied to it that exceeds the set limit value. By default a branch is infinitely strong, so to reach the limit, a lower value must be set manually. There are two parts to this limit, one for the tensile strength and one for the bending strength.

```
// Set the limit values for load on a branch
branch->setMaxLoad(tensileStrength, bendingStrength);
// or
branch->setMaxTranslationalLoad(tensileStrength);
branch->setMaxRotationalLoad(bendingStrength);
```

28.4. Contact Measurement

The `agxModel::ContactMeasurement` model is designed to calculate the energy dissipated at the point of impact for emitted rigid bodies. This functionality is showcased in the tutorial `tutorial_contactMeasurement.cpp`, found in the C++ tutorials section. For more info regarding emitted rigid bodies, see RigidBody emitters.

The formula used to compute the dissipated energy during a contact is as follows:

$$W = \lambda \times \frac{(v_t + v_{t+1})}{2}$$

where λ represents the Lagrange multiplier derived from the impact phase, often referred to as the impact lambda. Here, v_t is the velocity before impact, and v_{t+1} is the velocity after impact. The ContactMeasurement model calculates the dissipated energy for all contacts involving an emitted body, both between emitted bodies and between emitted bodies and none emitted rigid bodies.

To measure other kind of energy losses in your system, please see the :ref:` Energy Manager<energy_calculations_ref>`.

To create a new contact measurement model:

```
// Create a new contact measurement model
agxModel::ContactMeasurement contactMeasurementModel = new agxModel::ContactMeasurement();
```

To fetch an existing contact measurement model in the simluation:

```
// Get contact measurement model in simulation
agxModel::ContactMeasurement contactMeasurementModel = agxModel::ContactMeasurement::getListener(getSimulation());
```

To get the maximum impact energy an emitted body has registered during the simulation:

```
agxModel::ContactMeasurement contactMeasurementModel = agxModel::ContactMeasurement::getListener(getSimulation());

// Get max impact for a body
agx::Real maxImpact = contactMeasurementModel->getMaxImpactEnergy(body);
```

28.4.1. Limitations

Measuring dissipated energy on contacts with a very large penetration will not be accurate. It is advised to use a small time step when using the Contact Measurement model. Performing a frequency analysis is one way of determining if the time step size is sufficiently small. Below is an example of an apple colliding with a rigid wall. Consider an apple with the following characteristics:

Weight(m): 0.1kg
Young's modulus (E): 130 MPa
Approximate Contact area (A): $10^{-4}\,m^2$
Characteristic lenght(L_0): 0.1m

The stiffness (k) and natural frequency (f) calculations are as follows:

$$k = \frac{E \times A}{L_0}$$
$$f = \frac{1}{2\pi}\sqrt{\frac{k}{m}}$$

A higher natural frequency suggests a system with faster dynamics, necessitating a smaller time step to accurately resolve the dissipated energy during impacts within the system. This analysis should be undertaken whenever there's uncertainty about whether the selected time step is sufficiently small, such as when simulations yield unpredictable and excessively high impact energies. It is advisable to use a time step that is at least small enough to capture the frequency over three intervals. Note that the *contact area* depends on the *impact velocities* and the bodies shapes in the dynamic system. A larger impact velocity will result in a larger contact area and, consequently, a higher natural frequency.

In the mentioned example, we observe a natural frequency of approximately 200 Hz. To accurately resolve the dissipated energy in such impacts, a time step small enough to model this frequency across three intervals is required. Following the guideline from the example, a time step corresponding to 600 Hz is recommended.

There is currently an issue with bodies getting the wrong max impact during journal playback when the number of emitted bodies exceeds 5000.

28.5. Reconfigure Request

Reconfigure request have some similarities with a jump request, a set of items are specified and there is a reference body. But they have different purpose:

- The jump request **maintain the relative transforms** of the input items while changing all transforms so that the parent/reference body gets a new specified transform.
- The reconfigure request allow for **changing the relative transforms** while keeping the reference body fixed.

The reconfigure functionality can be used to change the pose of a machine with kinematic loops. For simpler structures which can be defined as serial chains one should consider if Forward kinematics can be used instead to perform the computation more efficiently.

Important

There are limitations regarding what ReconfigureRequest can handle:

Wires are not supported. A change of pose for a mechanical structure could lead to the wire coming in contact with geometry, new contact nodes added, leading to resolution changes and this does not fit well with the output format for the computed results. It is recommended to remove the wire(s), reconfigure the machine and then add wires again so they can be re-routed.

Tracks are partially supported. They are seen as ordinary bodies and constraints in the background simulation and to maintain stability, they are merged with AMOR with the sprocket. Hence requesting a new angle for the joint driving the sprocket will not work. What can be done is for example keeping the chassi fixed and rotating the base of an excavator and the tracks will still be positioned correctly on the excavator base.

The functionality is available in the class `ReconfigureRequest` where the main member function is `computeTransforms`:

```
/*
\param collection The bodies and constraint making up the structure that
                  should have a new configuration
\param referenceBody This body will be kept fixed when performing the movements.
\param positions Vector with constraint positions.
\param results Vector with (body,transform) pairs.
\param maxViolation Optional output of largest constraint violation found at new pose.
\param includeGeometry If geometry should be included in the copies and used
                       in the computations.
\param additionalSteps User configurable number of extra steps to take when moving
                       non-stiff structures. Increases computation time, but improves
                       results for e.g cables.
*/
bool computeTransforms(const agxSDK::Collection* collection,
                       const agx::RigidBody* referenceBody,
                       const agxModel::ConstraintPositionVector& positions,
                       agxModel::BodyTransformVector& results,
                       agx::Real* maxViolation = nullptr,
                       bool includeGeometry = false,
                       size_t additionalSteps = 0);
```

What this method does is that it creates clones of the supported items in the input collection. Then the clones are used in a background simulation that uses constraint controllers to drive the constraints to the requested positions (translational or rotational).

The resulting body transforms are then available in the `results` vector.

In contrast to `jumpRequest`, so far nothing is changed in the user Simulation. The next step is to update the bodies. This can either be done manually or by using `applyTransforms`:

```
/**
This method applies the results from computeTransforms by updating body transforms.
To maintain a stable simulation and reduce risk of artifacts after the instant
modification of transforms, additional steps are also performed.
- Hinges which had new joint positions requested in the input positions vector will
  have their winding number checked and corrected if needed.
- Constraints in the input positions vector that had Lock controllers enabled
  will have them disabled and instead motor with 0 speed enabled.
- All constraints part of the input collection will be checked to avoid an old
  position being used by a motor controllers if setLockedAtZero is enabled.

\warning This method can perform additional steps to help with maintaining a
         good state after updating the body transforms, but only for features
         supported by ReconfigureRequest that also are in the agxPhysics shared library.

         Therefore extra handling can be needed by the user even if this
         method is used. One such example is when hydraulics is used
         with PistonActuators. If the constraint connected to the piston
         have its position changed when body transforms are updated,
         then there will be a mismatch between the hydraulic cylinder and
         the body state. An example of how this can be adjusted
         by updating fluid in the piston chambers is available in the
         python tutorial tutorial_machine_reconfiguration.agxPy.
*/
void applyTransforms(agxSDK::Collection* collection,
                     const agxModel::ConstraintPositionVector& positions,
                     agxModel::BodyTransformVector& transforms,
                     bool zeroVelocities = true);
```

Changing body transforms instantaneous when the bodies are part of constraints can cause side effects when simulating forward using the new transforms.

If a constraint have an enabled motor with zero speed as well as setLockedAtZero enabled, then that controller will have an angle held internally. The new pose after reconfigure likely does no longer match that angle. Toggling the flag off and on again will update the angle.

Rotational changes for hinges can result in unexpected winding numbers due to the new pose alone can not provide information about the rotation direction. A rotation of +270 degrees or -90 degrees appear the same. `applyTransforms` can make sure winding numbers have the expected value for constraints part of the `positions`.

It is important to note that `applyTransforms` can not check and make all adjustments that might be needed after updating transforms. PowerLLine components, especially hydraulics, is something the user must handle explicitly. The main thing to check is that piston positions agree with the new pose. See `tutorial_machine_reconfiguration.agxPy` for an example of how this can be done.

29. AGX Model: Deformable Beam

Deformable beam is a lumped element model where the bulk properties of a beam is defined given choices of Beam Model, Beam Model Properties and the length of the beam. The lumped element model is independent of resolution (number of elements) of the beam.

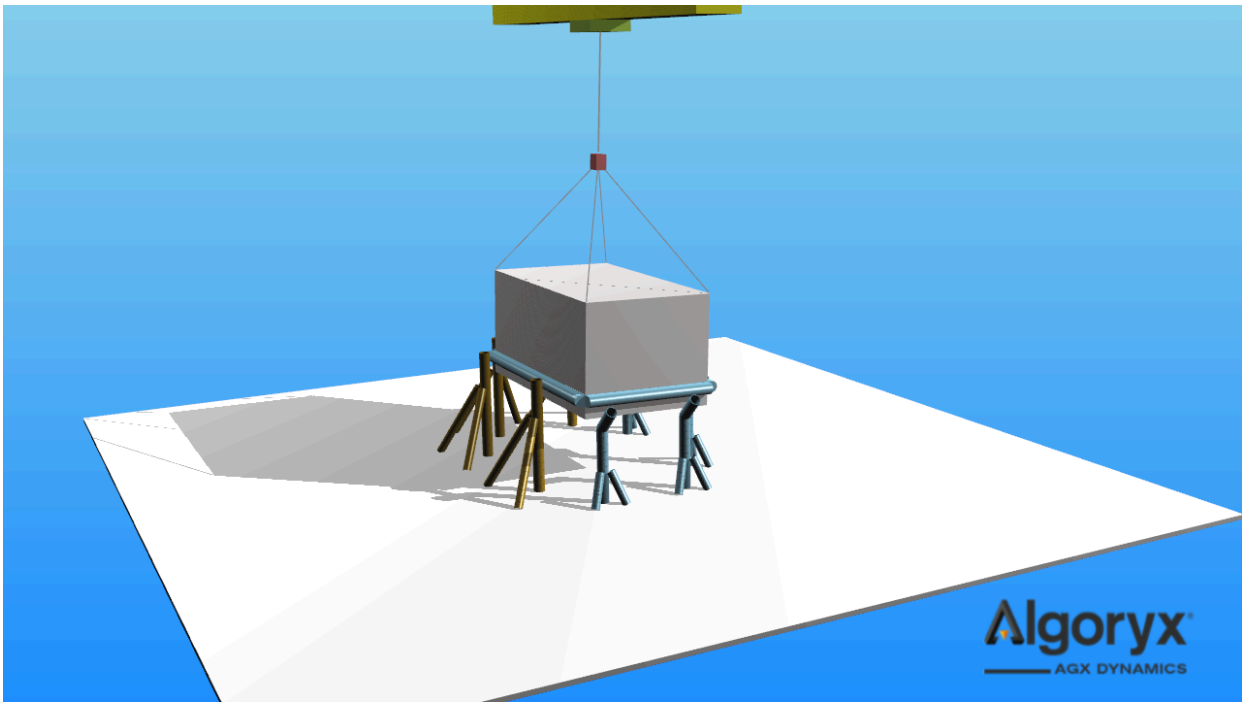


Fig. 29.1 Traverse crane placing heavy load with the help of Hollow Circular Beam model guide beams.

29.1. Beam Model

The beam model (`agxModel::BeamModel`) is the base class of numerous implementations and is describing the geometry and bulk properties - such as stretching, bending and torsion - of beams. Each beam model implementation contains immutable parameters describing the profile/cross section and is responsible for calculating the cross section area, moment of inertia (second moment of area and polar moment) and creating the template geometry used in each segment rigid body of the beam the model is associated with.

A beam model instance may be shared with an arbitrary number of Beam instances.

```
agxModel::CircularBeamRef circularModel_r05 = new agxModel::CircularBeam( /* radius */ 0.05 );
agxModel::BeamRefVector beams;
for ( size_t i = 0; i < 20; ++i )
  beams.push_back( new agxModel::Beam( /* model */ circularBeam_05,
                                       /* resolution */ 2 * (i + 1),
                                       /* length */ 3.0 ) );
```

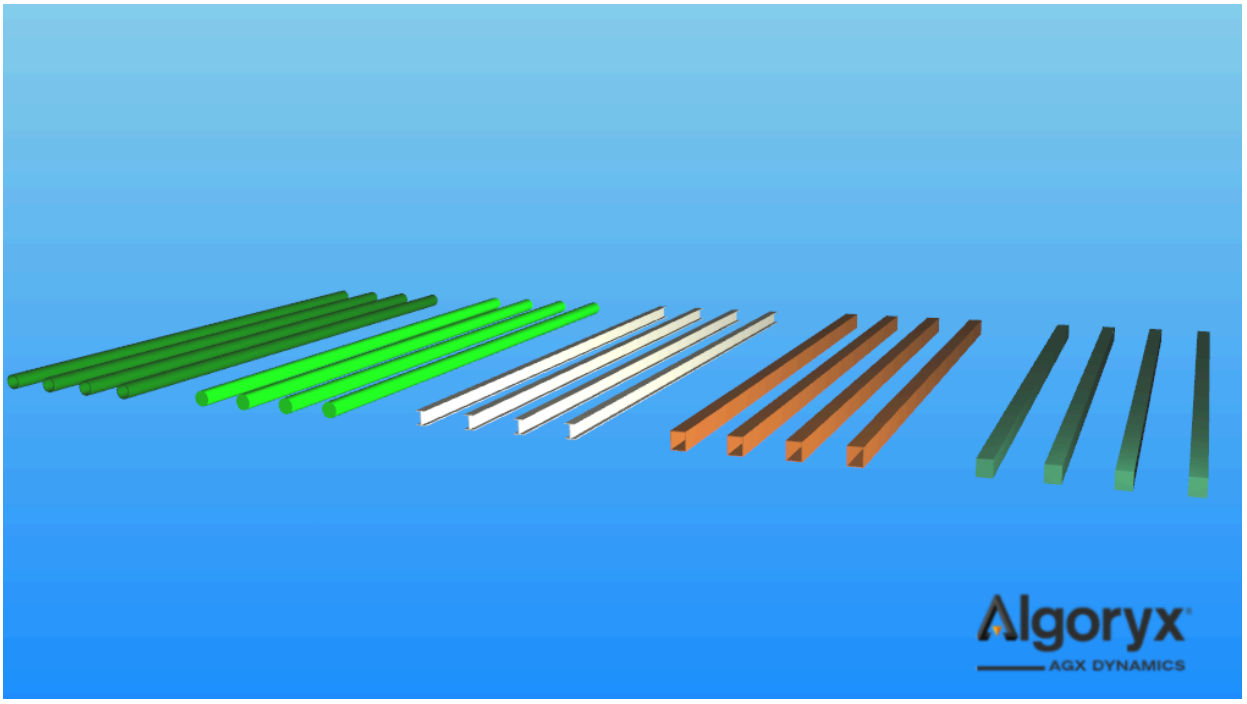


Fig. 29.2 Beams with various (10, 15, 20 and 25 elements) resolutions and models. The beam models, from left to right: Hollow Circular Beam, Circular Beam, I-Beam, Hollow Rectangular Beam and Rectangular Beam.

29.1.1. Beam Model Properties

Beam model properties (`agxModel::BeamModelProperties`) carries bulk material parameters such as **Young's Modulus**, **Poisson's Ratio** and **Damping Time**. These parameters are translated to compliance and damping in the constraints between beam segments, which also depends on the beam model (cross section area, moment of inertia) and resolution of the beam.

Property	Default
Young's Modulus	200.0E9
Poisson's Ratio	0.3333
Damping Time	0.03333

The **density**, which affects the mass properties of the beam, is taken from the (`agx::Material`) material instance associated to the beam. The mass properties of the beam is calculated when the beam is being initialized or when the material is being assigned, i.e., changing density in an already assigned material of an already initialized beam won't affect the mass properties of the beam.

Young's Modulus, **Poisson's Ratio** and **Damping Time** are synchronized in the associated beams several times per time step but the propagation (to the constraints in a beam) isn't instant, but are guaranteed to be used the next time the beam is being solved.

Note

Damping Time: For real-time simulations, i.e., time step size >> 0 ~ 0.01, there is no correlation between the material properties of a beam and the damping time parameter. If the time step size is decreased close to the true damping time of the simulated material, it's possible to map the damping time parameter to the viscosity of the material. **For real-time simulations it's important that the damping time parameter is larger than the time step, typically two times the time step size, for unconditionally stable simulations**.

A beam model properties instance may be shared with an arbitrary number of Beam Model instances, and a beam model will always contain a valid reference to a beam model properties instance.

```
agx::MaterialRef steelBeamMaterial = new agx::Material( "steel" );
steelBeamMaterial->getBulkMaterial()->setDensity( 7400.0 );

agxModel::BeamModelPropertiesRef steelBeamProperties = new agxModel::BeamModelProperties();
steelBeamProperties->setYoungsModulus( 210.0E9 );
steelBeamProperties->setPoissonsRatio( 0.31 );

agxModel::CircularBeamRef circularModel1 = new agxModel::CircularBeam( /* radius */ 0.1 );
circularBeam1->setProperties( steelBeamProperties );

agxModel::CircularBeamRef circularModel2 = new agxModel::CircularBeam( /* radius */ 0.12,
                                                                       /* properties */ steelBeamProperties );
```

```
agxModel::CircularBeamRef circularModel3 = new agxModel::CircularBeam( /* radius */ 0.12 );
circularModel3->getProperties()->setYoungsModulus( steelBeamProperties->getYoungsModulus() );
circularModel3->getProperties()->setPoissonsRatio( steelBeamProperties->getPoissonsRatio() );
```

In the example above, all three models has identical bulk properties but `circularModel1` and `circularModel2` shares the `steelBeamProperties` instance, so when any value is updated in `steelBeamProperties`, all beams based on `circularModel1` and `circularModel2` will have changed bulk properties while `circularModel3` will have the old values when it was assigned copies of the parameters.

29.1.2. Rectangular Beam

Rectangular cross section beam model constructed given *width* and *height*. The geometry in each beam segment contains a single `agxCollide::Box` of size $(0.5h, 0.5w, 0.5\Delta L)$ where ΔL is determined by the resolution and length of the Beam this model is part of.

```
agxModel::RectangularBeamRef
   rectangularModel = new agxModel::RectangularBeam( /* width */ 0.05,
                                                     /* height */ 0.10 );
```

```
rectangular_model = agxModel.RectangularBeam(width = 0.05,
                                             height = 0.10)
```

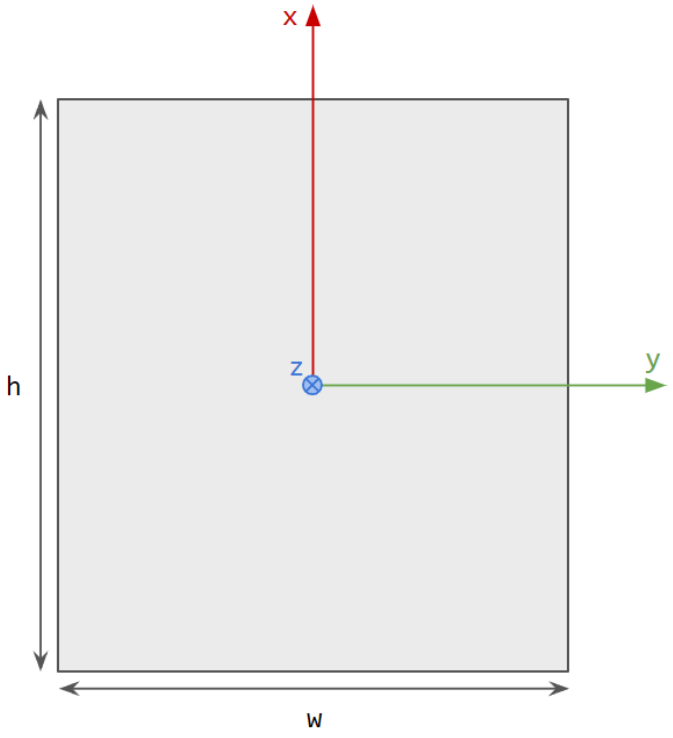


Fig. 29.3 Rectangular cross section defined by width (**w**) and height (**h**).

Moment of inertia:

$$
\begin{aligned}
I_{xx} &= \frac{w^3 h}{12} \\
I_{yy} &= \frac{w h^3}{12} \\
J &= ab^3 \left(\frac{16}{3} - 3.36\frac{a}{b}\left(1 - \frac{b^4}{12a^4}\right)\right) \\
&\text{where} \\
a &= 0.5 \max(w, h) \\
b &= 0.5 \min(w, h)
\end{aligned}
$$

The polar moment J is an approximation that's more accurate than $J = I_{xx} + I_{yy}$. See Moment Of Inertia Scale for more information how to tune the accuracy even further.

Cross section area:

$$A = wh$$

29.1.3. Hollow Rectangular Beam

Hollow rectangular cross section beam model constructed given *width*, *height* and *thickness*. The geometry in each beam segment contains four `agxCollide::Box` instances of sizes - bottom and top *flange* $(0.5t, 0.5w, 0.5\Delta L)$, left and right *web* $(0.5h_i, 0.5t, 0.5\Delta L)$ - where ΔL is determined by the resolution and length of the Beam this model is part of.

```
agxModel::HollowRectangularBeamRef
   hollowRectangularModel = new agxModel::HollowRectangularBeam( /* width */ 0.12,
                                                                 /* height */ 0.15,
                                                                 /* thickness */ 4.0E-3 );
```

```
hollow_rectangular_model = agxModel.HollowRectangularBeam(width = 0.12,
                                                          height = 0.15,
                                                          thickness = 4.0E-3)
```

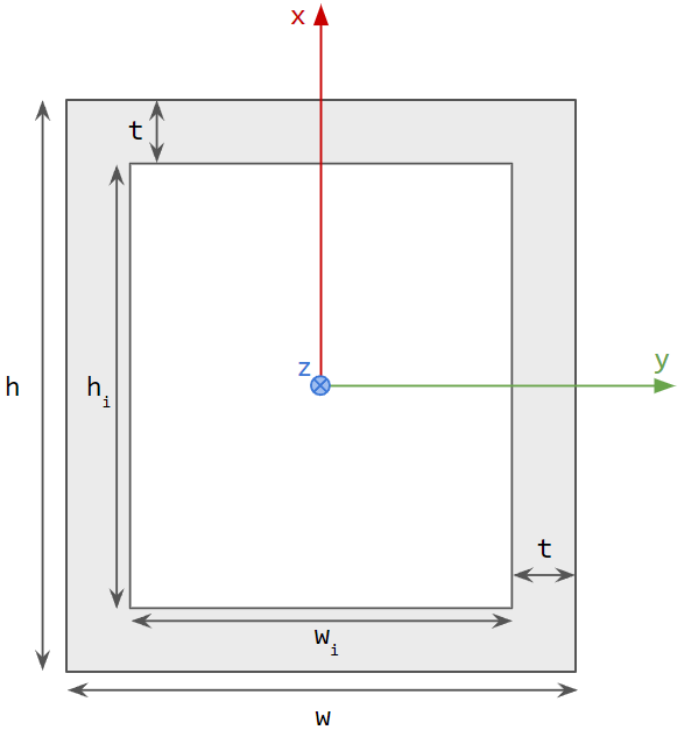


Fig. 29.4 Hollow rectangular cross section defined by width (**w**), height (**h**) and thickness (**t**).

Moment of inertia:

$$\begin{aligned} I_{xx} &= \frac{w^3 h - w_i^3 h_i}{12} \\ I_{yy} &= \frac{w h^3 - w_i h_i^3}{12} \\ J &= I_{xx} + I_{yy} \end{aligned}$$

The used polar moment $J = I_{xx} + I_{yy}$ is not very accurate but given specific sizes and data it's possible to tune this model using Moment Of Inertia Scale.

Cross section area:

$$A = wh - w_i h_i$$

29.1.4. Circular Beam

Circular cross section beam model constructed given *radius*. The geometry in each beam segment contains a single `agxCollide::Cylinder` of size $(r, \Delta L)$ where ΔL is determined by the resolution and length of the Beam this model is part of.

```
agxModel::CircularBeamRef
   circularModel = new agxModel::CircularBeam( /* radius */ 0.05 );
```

```
circular_model = agxModel.CircularBeam(radius = 0.05)
```

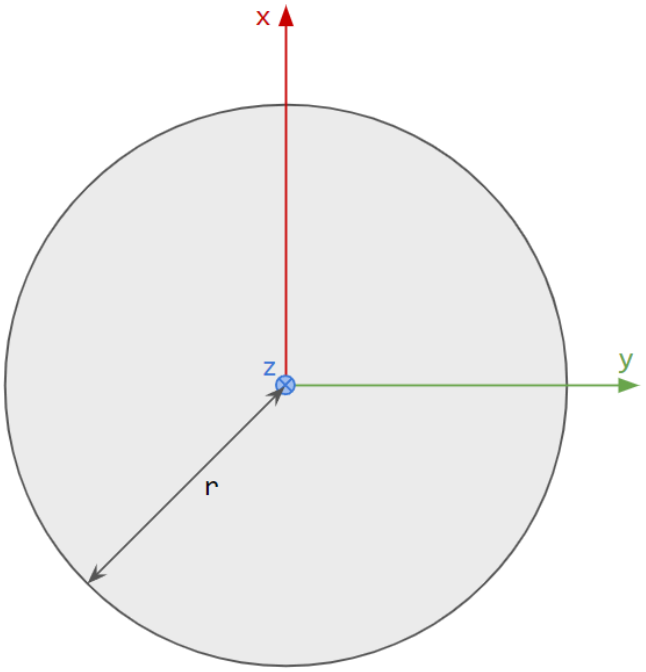


Fig. 29.5 Circular cross section defined by radius (**r**).

Moment of inertia:

$$
\begin{aligned}
I_{xx} &= \pi \frac{r^4}{4} \\
I_{yy} &= \pi \frac{r^4}{4} \\
J &= I_{xx} + I_{yy}
\end{aligned}
$$

Cross section area:

$$A = \pi r^2$$

29.1.5. Hollow Circular Beam

Hollow circular cross section beam model constructed given *radius* and *thickness*. The geometry in each beam segment contains a single `agxCollide::HollowCylinder` of size $(r_i, \Delta L, t)$ where ΔL is determined by the resolution and length of the Beam this model is part of.

```
agxModel::HollowCircularBeamRef
   hollowCircularModel = new agxModel::HollowCircularBeam( /* radius */ 0.07,
                                                           /* thickness */ 3.0E-3 );
```

```
hollow_circular_model = agxModel.CircularBeam(radius = 0.07,
```

```
                                                thickness = 3.0E-3)
```

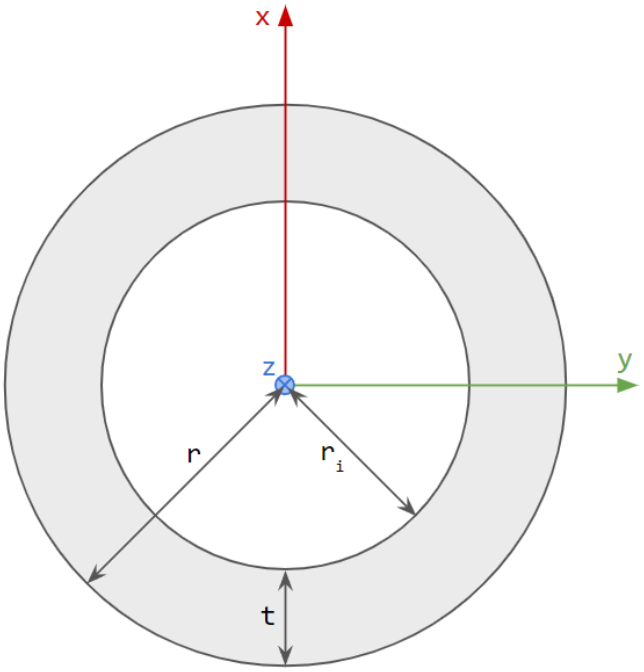


Fig. 29.6 Hollow circular cross section defined by radius (**r**) and thickness (**t**).

Moment of inertia:

$$
\begin{aligned}
I_{xx} &= \pi \frac{r^4 - r_i^4}{4} \\
I_{yy} &= \pi \frac{r^4 - r_i^4}{4} \\
J &= I_{xx} + I_{yy}
\end{aligned}
$$

Cross section area:

$$A = \pi(r^2 - r_i^2)$$

29.1.6. I-Beam

I cross section beam model constructed given *width*, *height*, *flange* and *web* thickness. The geometry in each beam segment contains three `agxCollide::Box` instances if sizes - bottom and top *flange* $(0.5t_f, 0.5w, 0.5\Delta L)$ and the web $(0.5h_w, 0.5t_w, 0.5\Delta L)$ - where ΔL is determined by the resolution and length of the Beam this model is part of.

```
agxModel::IBeamRef
   iModel = agxModel::IBeam( /* width */ 0.15,
                             /* height */ 0.17,
                             /* flangeThickness */ 4.0E-3,
                             /* webThickness */ 5.5E-3 );
```

```
i_model = agxModel.IBeam(width = 0.15,
                         height = 0.17,
                         flangeThickness = 4.0E-3,
                         webThickness = 5.5E-3)
```

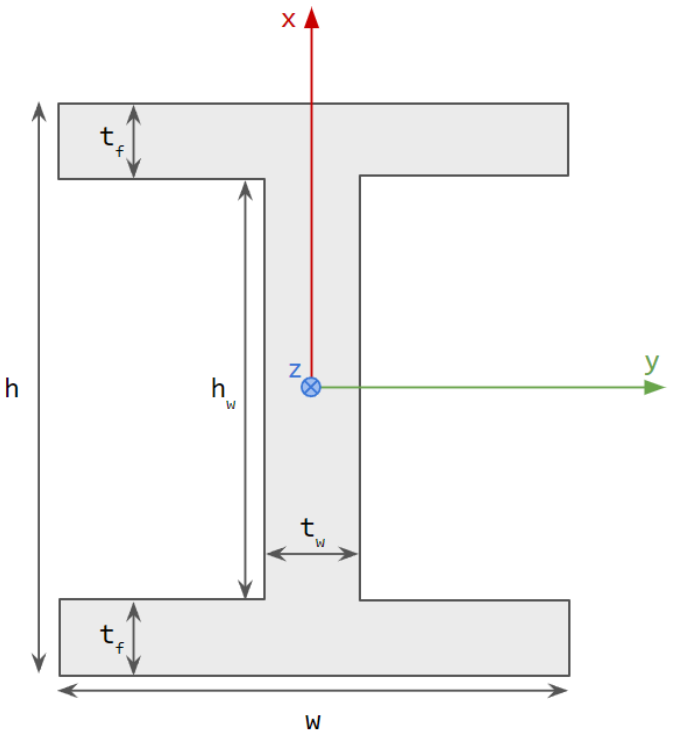


Fig. 29.7 I cross section defined by width (**w**), height (**h**), flange thickness (t_f) and web thickness (t_w).

Moment of inertia:

$$
\begin{aligned}
I_{xx} &= \frac{t_f^3 h_w}{12} + \frac{w^3 t_f}{6} \\
I_{yy} &= \frac{h_w^3 t_f}{12} + \frac{w h^3}{12} - h_w^3 \\
J &= I_{xx} + I_{yy}
\end{aligned}
$$

The used polar moment $J = I_{xx} + I_{yy}$ is not very accurate but given specific sizes and data it's possible to tune this model using Moment Of Inertia Scale.

Cross section area:

$$A = 2wt_f + h_w t_w$$

29.1.7. Moment Of Inertia Scale

The Beam Model base class carries a moment of inertia scale, default (1, 1, 1), that scales the moment of inertia implemented by the model. This enables models, of non-circular cross sections (Rectangular Beam, Hollow Rectangular Beam, I-Beam), of a given size, to be tuned against validated data. This is mostly relevant for the polar moment J, related to the target torsion angle given external forces and torques applied to the beam.

Note

`agxModel::BeamModel::getMomentOfInertia()` returns the scaled moment of inertia and should be used in most cases. The moment of inertia of a model is implemented in `agxModel::BeamModel::calculateMomentOfInertia()` which returns the unscaled, model specific implementation of the moment of inertia.

For example, if the polar moment is know for a given model and sizes, e.g., `J = 1.234E-5`, set the scale as:

```
const auto J = 1.234E-5;
const auto polarScale = J / model->calculateMomentOfInertia().z();
model->setPolarMomentOfInertiaScale( polarScale );
```

which will result in the final polar moment to be the known `J`.

29.2. Beam

A beam is constructed given a Beam Model, *number of elements* and *length*. The beam segments, i.e., bodies, geometries and constraints, are created when `initialize()` is called or when the beam is added to a simulation.

```
agxModel::RectangularBeamRef model = new agxModel::RectangularBeam( 0.15, 0.15 );

agxModel::BeamRef beam1 = new agxModel::Beam( model, 15u, 5.0 );
// Explicit initialization.
if ( !beam1->initialize() )
  std::cerr << "beam1 initialization failed!" << std::endl;
else
  simulation->add( beam1 );

agxModel::BeamRef beam2 = new agxModel::Beam( model, 25u, 3.0 );
// Implicit initialization.
simulation->add( beam2 );
if ( !beam->isInitialized() ) {
  std::cerr << "beam2 initialization failed!" << std::endl;
  simulation->remove( beam2 );
}
```

See Beam Frame for information where the segments are created and transformed, and Creating Beams for different ways to create, initialize and transform beams.

29.2.1. Beam Frame

The beam frame is the frame in which all beam segments lives in. All segment rigid bodies are inheriting this frame, but the beam frame is static so it won't move with the beam during a simulation, it's mostly used for the initial transform.

The beam frame is defined by:

1. The **forward** direction, the direction *along* the beam, is by definition the **+z axis** (inherited from constraints and the constraint axis).
2. The **up** direction is chosen to be the **+x axis** (+x or -x are the choices and +x feels more natural).
3. The **right** direction becomes the **+y axis**.
4. The start position of the first segment is located at the origin in the beam frame.
5. For a given beam length L, the end position of the last segment is located at $(0, 0, L)$ in the beam frame.

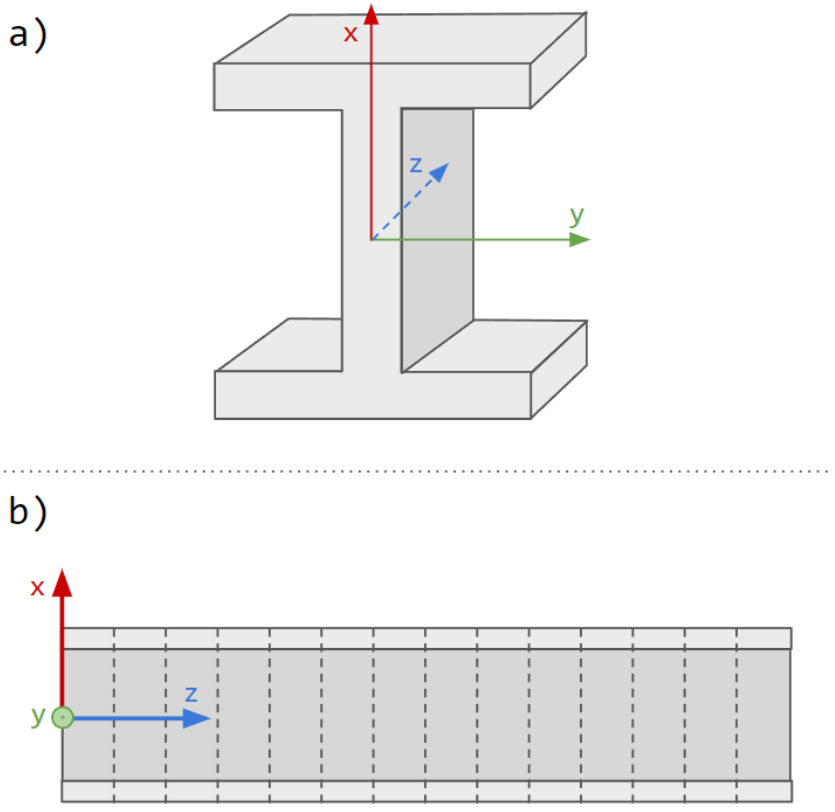


Fig. 29.8 **a)** Beam segment (rigid body) frame at any location along the beam with **up** along **+x**, **right** along **+y** and **forward** along **+z** (into the screen). **b)** Beam frame instantly after the beam has been initialized with **up** along **+x**, **right** along **+y** (out of the screen) and **forward** along **+z**.

Note

The **beam** frame (initially), **beam segments** (rigid bodies) frames and **beam constraints** frames **coincides**.

29.2.2. Creating Beams

There are several ways to create and initialize beams. The default transform will coincide with the world frame where the beam segments are created along world +z axis with up axis along world +x and right axis along world +y.

```
// 5 m long beam with 5 segments (results in 1 m each).
agxModel::BeamRef beam = new agxModel::Beam( new agxModel::CircularBeam( 0.1 ),
                                             5u,
                                             5.0 );
beam->initialize();
for ( const auto segment : beam->getSegments() )
  // segment->getBeginPosition() == segment->getRigidBody()->getPosition()
  std::cout << segment->getBeginPosition() << std::endl;
std::cout << "End: " << beam->getEndPosition() << std::endl;
```

```
[0.0, 0.0, 0.0]
[0.0, 0.0, 1.0]
[0.0, 0.0, 2.0]
[0.0, 0.0, 3.0]
[0.0, 0.0, 4.0]
End: [0.0, 0.0, 5.0]
```

29.2.2.1. Set Rotation

Beam rotations can be calculated using the static method `agxModel::Beam::calculateRotation`, taking desired **forward** direction of a beam in world coordinate frame and **world up** (default world +z axis) direction. The **world up** is *a hint* of the direction the up (x) axis of the beam should align with. The following example is creating the beams along the world x axis, but *world up* is alternated (z, y, x + y).

```
agxModel::IBeamRef model = new agxModel::IBeam( 0.15, 0.20, 0.01, 0.015 );
const auto beamForwardWorld = agx::Vec3::X_AXIS();

agxModel::BeamRef beam1 = new agxModel::Beam( model, 5u, 5.0 );
beam1->setRotation( agxModel::Beam::calculateRotation( /* forward */ beamForwardWorld,
                                                       /* worldUp */ agx::Vec3::Z_AXIS() ) );
// The is the start position, in world, of beam1.
beam1->setPosition( 0, 1.5, 0 );
simulation->add( beam1 );

agxModel::BeamRef beam2 = new agxModel::Beam( model, 5u, 5.0 );
beam2->setRotation( agxModel::Beam::calculateRotation( /* forward */ beamForwardWorld,
                                                       /* worldUp */ agx::Vec3::Y_AXIS() ) );
// The is the start position, in world, of beam2.
beam2->setPosition( 0, 0.5, 0 );
simulation->add( beam2 );

agxModel::BeamRef beam3 = new agxModel::Beam( model, 5u, 5.0 );
beam3->setRotation( agxModel::Beam::calculateRotation( /* forward */ beamForwardWorld,
                                                       /* worldUp */ agx::Vec3::Y_AXIS() +
                                                                     agx::Vec3::Z_AXIS() ) );
// The is the start position, in world, of beam3.
beam3->setPosition( 0, -0.5, 0 );
simulation->add( beam3 );
```

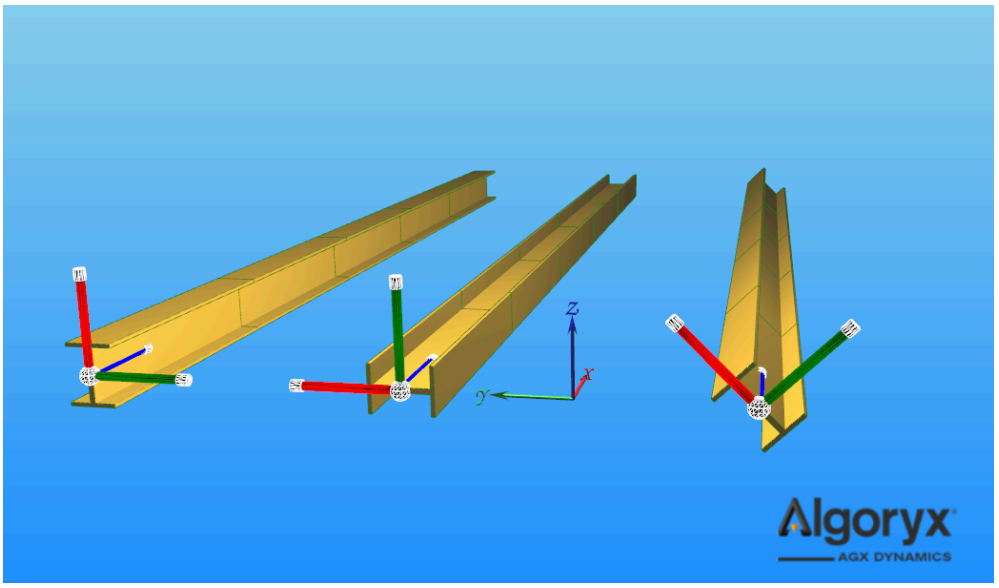


Fig. 29.9 The world frame is the frame without connection to a beam. Initialized beam with rotation along world x with given **world up**, from left to right: z, y, (y + z) / |y + z|

Note

The beam will be aligned with the given **forward** direction, while the **world up** direction is used to find the remaining beam frame directions - *up* (x) and *right* (y). The **world up** direction is normally just the up world axis you use in your simulation.

29.2.2.2. Set Transform

Similar to Set Rotation, beam transforms can be calculated using the static method `agxModel::Beam::calculateTransform`, taking an additional argument **start position** in world coordinate frame.

```
agxModel::IBeamRef model = new agxModel::IBeam( 0.15, 0.20, 0.01, 0.015 );
const auto beamForwardWorld = agx::Vec3::X_AXIS();

agxModel::BeamRef beam1 = new agxModel::Beam( model, 5u, 5.0 );
beam1->setRotation( agxModel::Beam::calculateTransform( /* startPosition */ agx::Vec3( 0, 1.5, 0 ),
                                                        /* forward */ beamForwardWorld,
                                                        /* worldUp */ agx::Vec3::Z_AXIS() ) );
simulation->add( beam1 );

agxModel::BeamRef beam2 = new agxModel::Beam( model, 5u, 5.0 );
beam2->setRotation( agxModel::Beam::calculateTransform( /* startPosition */ agx::Vec3( 0, 0.5, 0 ),
                                                        /* forward */ beamForwardWorld,
                                                        /* worldUp */ agx::Vec3::Y_AXIS() ) );
simulation->add( beam2 );

agxModel::BeamRef beam3 = new agxModel::Beam( model, 5u, 5.0 );
beam3->setRotation( agxModel::Beam::calculateTransform( /* startPosition */ agx::Vec3( 0, -0.5, 0 ),
                                                        /* forward */ beamForwardWorld,
                                                        /* worldUp */ agx::Vec3::Y_AXIS() +
                                                                      agx::Vec3::Z_AXIS() ) );
simulation->add( beam3 );
```

The final beam configurations are identical to the scene in Set Rotation where `setPosition` is used.

29.2.2.3. From To Position

Beams can be instantiated, initialized and transformed using the static method `agxModel::Beam::create` - taking beam **start position** and **end position** in world coordinate frame, Beam Model, **resolution** and **world up**. The resulting beam length will become $L = |\text{endPosition} - \text{startPosition}|$.

```
agxModel::IBeamRef model = new agxModel::IBeam( 0.15, 0.20, 0.01, 0.015 );
const auto beamForwardWorld = agx::Vec3::X_AXIS();

agxModel::BeamRef beam1 = agxModel::Beam::create( /* startPosition */ agx::Vec3( 0, 1.5, 0 ),
                                                  /* endPosition */ agx::Vec3( 5.0, 1.5, 0 ),
                                                  /* model */ model,
```

```
                                                /* resolution */ 5u,
                                                /* worldUp */ agx::Vec3::Z_AXIS() );
simulation->add( beam1 );

agxModel::BeamRef beam2 = agxModel::Beam::create( /* startPosition */ agx::Vec3( 0, 0.5, 0 ),
                                                /* endPosition */ agx::Vec3( 5.0, 0.5, 0 ),
                                                /* model */ model,
                                                /* resolution */ 5u,
                                                /* worldUp */ agx::Vec3::Y_AXIS() );
simulation->add( beam2 );

agxModel::BeamRef beam3 = agxModel::Beam::create( /* startPosition */ agx::Vec3( 0, -0.5, 0 ),
                                                /* endPosition */ agx::Vec3( 5.0, -0.5, 0 ),
                                                /* model */ model,
                                                /* resolution */ 5u,
                                                /* worldUp */ agx::Vec3::Y_AXIS() +
                                                              agx::Vec3::Z_AXIS() );
simulation->add( beam3 );
```

The final beam configurations are identical to the scene in Set Rotation.

29.2.3. Attaching Beams

Beams has utility methods for attaching segments to other bodies (or world). The attachment constraint frames are aligned with frame of the beam in context, and the constraint is created, configured and then returned, i.e., not added nor tracked by the beam and is not added to the simulation.

Example of a cantilever beam, attached rigidly to world at begin, and a fixed ended beam, attached rigidly to world at begin and end.

```
/**
Utility method to attach the first segment of this beam at begin (of the segment) to
another rigid body and optionally adopt the material properties of the model of this beam.
\note The returned lock joint won't be added to the simulation or to this beam and this beam
      has to be initialized.
\param otherRb - other rigid body to attach first segment to - nullptr is world
\param adoptModelProperties - calculate compliance and damping of the attachment given the
                              beam model of this beam
\return lock joint if valid, otherwise nullptr (e.g., \p segment is null or not part of this beam)
*/
agx::LockJointRef attachBegin( agx::RigidBody* otherRb = nullptr,
                               agx::Bool adoptModelProperties = false ) const;

/**
Utility method to attach the last segment of this beam at end (of the segment) to
another rigid body and optionally adopt the material properties of the model of this beam.
\note The returned lock joint won't be added to the simulation or to this beam and this beam
      has to be initialized.
\param otherRb - other rigid body to attach first segment to - nullptr is world
\param adoptModelProperties - calculate compliance and damping of the attachment given the
                              beam model of this beam
\return lock joint if valid, otherwise nullptr (e.g., \p segment is null or not part of this beam)
*/
agx::LockJointRef attachEnd( agx::RigidBody* otherRb = nullptr,
                             agx::Bool adoptModelProperties = false ) const;
```

```
agxModel::RectangularBeamRef model = new agxModel::RectangularBeam( 0.10, 0.12 );

// Cantilever beam, attached rigidly at begin, to world.
auto cantileverBeam = agxModel::Beam::create( agx::Vec3( 0, 0, 0 ),
                                              agx::Vec3( 3, 0, 0 ),
                                              model,
                                              150u );
agx::LockJointRef cantileverAttachment = cantileverBeam->attachBegin();
cantileverAttachment->setCompliance( 1.0E-14 );
simulation->add( cantileverAttachment );
simulation->add( cantileverBeam );

// Fixed ended beam, attached rigidly at begin and end, to world.
```

```
auto fixedBeam = agxModel::Beam::create( agx::Vec3( 0, 0.5, 0 ),
                                         agx::Vec3( 3, 0.5, 0 ),
                                         model,
                                         150u );
agx::LockJointRef fixedBeginAttachment = fixedBeam->attachBegin();
agx::LockJointRef fixedEndAttachment = fixedBeam->attachEnd();
fixedBeginAttachment->setCompliance( 1.0E-14 );
fixedEndAttachment->setCompliance( 1.0E-14 );
simulation->add( fixedBeginAttachment );
simulation->add( fixedEndAttachment );
simulation->add( fixedBeam );
```

Another example with guide and support beams. Both beam starts at and is attached to ground/world. The support beam end center is attached to the guide beam surface. This example also examines how the guide vs. support beam attachment frame differs depending on context.

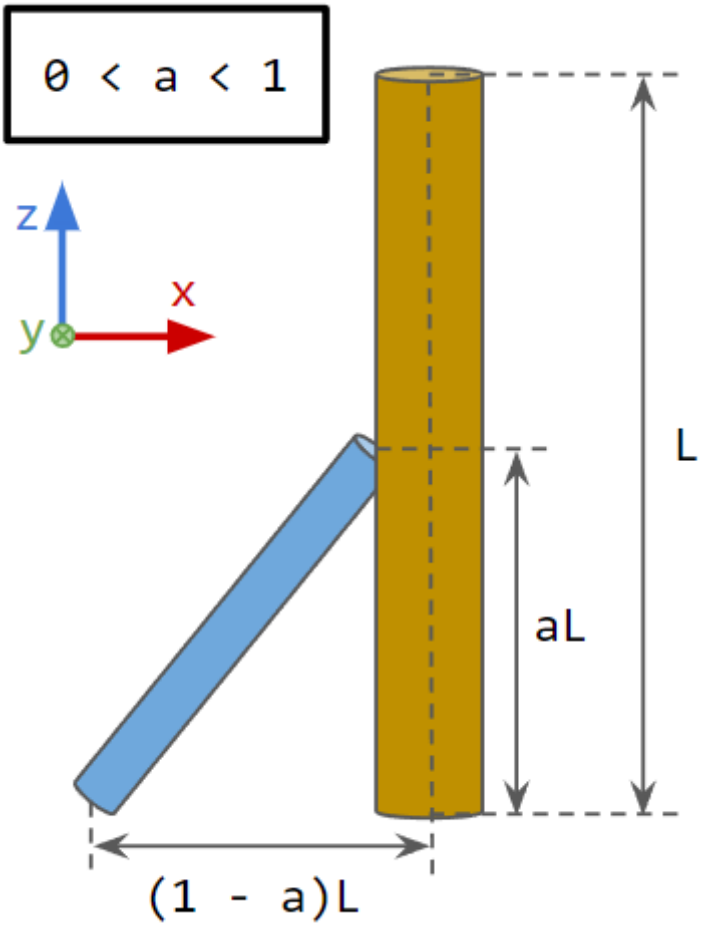


Fig. 29.10 Guide and support beams where the support beam start and end depends on the guide beam length and a scale.

```
/**
Utility method to attach a given segment (of this beam) at a local segment position to
another rigid body and optionally adopt the material properties of the model of this beam.
\note The returned lock joint won't be added to the simulation or to this beam and this beam
      has to be initialized.
\param segment - segment of this beam to attach
\param localSegmentPosition - beam segment attachment position given in local segment frame
\param otherRb - other rigid body to attach \p segment to - nullptr is world
\param adoptModelProperties - calculate compliance and damping of the attachment given the
                              beam model of this beam
\return lock joint if valid, otherwise nullptr (e.g., \p segment is null or not part of this beam)
*/
agx::LockJointRef attach( BeamSegment* segment,
                          agx::Vec3 localSegmentPosition,
                          agx::RigidBody* otherRb = nullptr,
                          agx::Bool adoptModelProperties = false ) const;
```

```
agxSDK::AssemblyRef createGuideSupport( agx::Real radius,
                                        agx::UInt resolution,
                                        agx::Real L,
                                        agx::Real a,
                                        agx::Bool guideIsAttachmentParent )
{
  agxSDK::AssemblyRef assembly = new agxSDK::Assembly();

  const auto supportRadius = 0.85 * radius;
  const auto guideThickness = 5.0E-2 * radius;
  const auto supportThickness = 5.0E-2 * supportRadius;
  const auto supportResolution = agx::UInt( agx::Real( resolution ) * a + 0.5 );

  agxModel::BeamModelPropertiesRef properties = new agxModel::BeamModelProperties( 190.0E9,
                                                                                   0.31 );
  agxModel::HollowCircularBeamRef guideModel = new agxModel::HollowCircularBeam( radius,
                                                                                 guideThickness,
                                                                                 properties );
  agxModel::HollowCircularBeamRef supportModel = new agxModel::HollowCircularBeam( supportRadius,
                                                                                   supportThickness,
                                                                                   properties );

  agxModel::BeamRef guideBeam = agxModel::Beam::create( agx::Vec3( 0, 0, 0 ),
                                                        agx::Vec3( 0, 0, L ),
                                                        guideModel,
                                                        resolution );
  assembly->add( guideBeam );

  const auto supportBeginPosition = guideBeam->getBeginPosition() -
                                    ( 1.0 - a ) * L * agx::Vec3::X_AXIS();
  auto attachmentGuideSegment = guideBeam->getSegment( a * L );
  const auto attachmentGuideFrame = attachmentGuideSegment->getRigidBody()->getFrame();
  // Distance along the local z axis of the guide beam segment that
  // we're attaching the support beam.
  const auto localAttachmentLength = a * L -
                                     // Rest length to the start of the segment.
                                     guideBeam->findRestLengthTo( attachmentGuideSegment );
  const auto supportEndPosition = attachmentGuideFrame->transformPointToWorld( guideModel->getRadius(),
                                                                               0.0,
                                                                               localAttachmentLength );

  agxModel::BeamRef supportBeam = agxModel::Beam::create( supportBeginPosition,
                                                          supportEndPosition,
                                                          supportModel,
                                                          supportResolution );
  assembly->add( supportBeam );

  agxUtil::setEnableCollisions( guideBeam, supportBeam, false );

  agx::LockJointRef guideSupportAttachment = nullptr;
  // If true, we create the attachment given 'guideBeam', so the
  // attachment frames will align with the guide beam frame.
  if ( guideIsAttachmentParent ) {
    guideSupportAttachment = guideBeam->attach( attachmentGuideSegment,
                                                agx::Vec3( guideModel->getRadius(),
                                                           0,
                                                           localAttachmentLength ),
                                                // Support beam last segment rigid body is
                                                // attached to the guide beam.
                                                supportBeam->getSegments().back()->getRigidBody() );
    // Applying constraint properties (compliance and damping) of the
    // guide beam model to the attachment, but twice as stiff when the
    // attachment can be assumed to be more rigid.
    guideBeam->calculateSegmentStiffnessDamping().apply( guideSupportAttachment, 2.0 );
  }
  // If false, we create the attachment given 'supportBeam' and
  // the attachment frames will align with the support beam frame.
  else {
```

```
    guideSupportAttachment = supportBeam->attachEnd( attachmentGuideSegment->getRigidBody() );
    // Applying constraint properties (compliance and damping) of the
    // support beam model to the attachment, but twice as stiff when the
    // attachment can be assumed to be more rigid.
    supportBeam->calculateSegmentStiffnessDamping().apply( guideSupportAttachment, 2.0 );
  }
  assembly->add( guideSupportAttachment );

  return assembly;
}
```

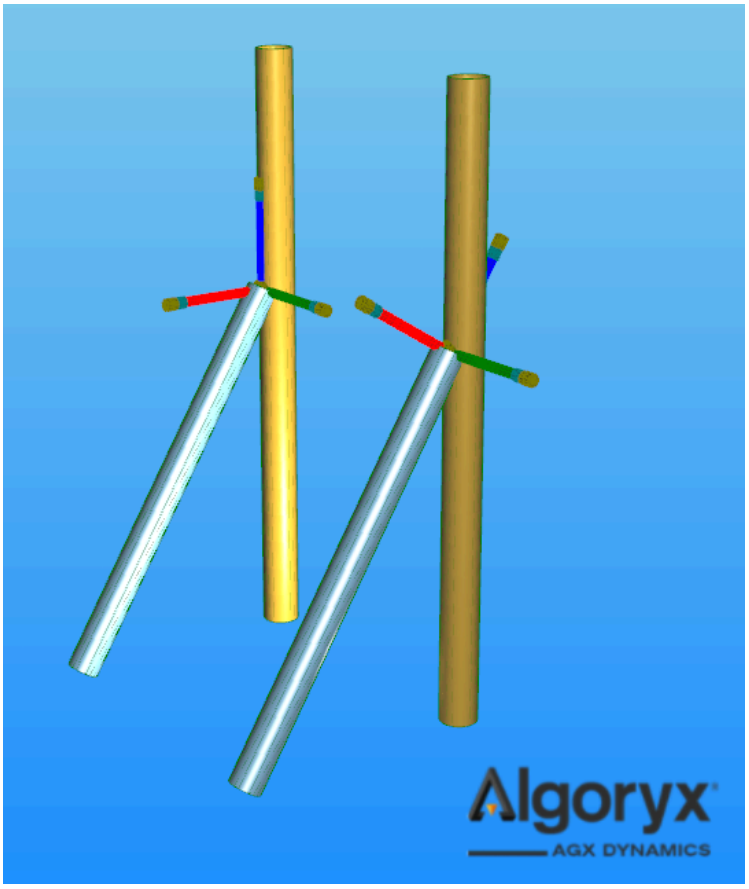


Fig. 29.11 Result calling `createGuideSupport` twice with `guideIsAttachmentParent` as `true` (left) and `false` (right).

29.2.3.1. Adopt Model Properties

Constraint compliance and damping time in the constraints between beam segments depends on the resolution and length of the beam, and cross section area and moment of inertia of the model. When the beam is attached to another object or world, it's possible to transfer the beam model properties to the attachment constraint. E.g.,

```
// Stiffness and damping time of the constraints in the beam.
agxModel::BeamStiffnessDamping beamStiffnessDamping = beam->calculateSegmentStiffnessDamping();
// Attach begin of the beam to world.
agx::LockJointRef attachment = beam->attachBegin();
// Apply beam properties as if 'attachment' was part of the beam.
beamStiffnessDamping.apply( attachment, /* stiffness scale */ 1.0 );
simulation->add( attachment );
```

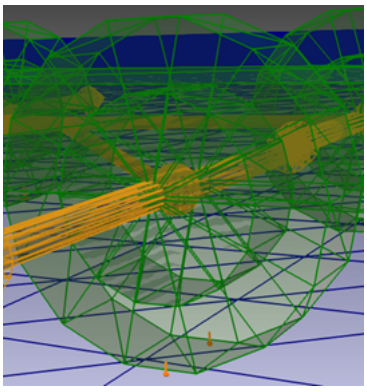
Similarly, all `agxModel::Beam::attach*` methods has an argument `adoptModelProperties`, which defaults to `false`, that will if `true`, transfer the beam properties to the constraint as if the attachment is rigidly attached.

Through tests, by performing Newton's method to minimize the deflection error (Euler-Bernoulli) for cantilever beams, it has been shown that the beam model properties cannot be applied without modification. The scale of the stiffness depends on the length and resolution of the beam but a decent approximation is that the stiffness has to be scaled in the range `[2.0, 2.5]`. If `adoptModelProperties = true` the stiffness is scaled by `2.3`, identical as,

```
agx::LockJointRef attachment = beam->attachBegin();
// Approximately rigidly attached to world as the default implementation.
beamStiffnessDamping.apply( attachment, /* stiffness scale */ 2.3 );
simulation->add( attachment );
```

30. AGX Model: Tire

agxModel::TwoBodyTireModel is an implementation of a tire model with two separate rigid bodies; the tire and the rim. The user has control over the deformation of the different degrees of freedom by setting stiffness and damping:



Creating a TwoBodyTireModel requires two bodies, one for the rim and one for the rigid body. It is assumed that the two bodies are positioned and added to the simulation before or after the creation of the TwoBodyTireModel

```
agx::Real tireRadius = 1;  // Not important, but can give information of
agx::Real rimRadius = 0.5; // amount of deformation of tire.
agx::Real stiffness = 2E6;

agxModel::TwoBodyTireModelRef tire =
      new agxModel::TwoBodyTireModel(tireBody, tireRadius, rimBody, rimRadius );
```

30.1. Stiffness

The stiffness of the tire will control the amount of deformation for the tire. There are four degrees of freedom for which the stiffness can be specified.

The unit for translational stiffness is **force/displacement** (if using SI: N/m). The unit for rotational stiffness is **torque/angular displacement** (if using SI: Nm/rad)

```
// Configure the deformation stiffness for all DOF of the tire
tire->setStiffness(stiffness*2, agxModel::TwoBodyTire::RADIAL)
tire->setStiffness(stiffness*10, agxModel::TwoBodyTire::LATERAL)
tire->setStiffness(stiffness*4, agxModel::TwoBodyTire::BENDING)
tire->setStiffness(stiffness*0.1, agxModel::TwoBodyTire::TORSIONAL)
```

30.2. Damping

The damping corresponds to the rate of which the deformation should be restored. It has the same degrees of freedom as the stiffness.

The unit for translational damping is **force * time/displacement** (if using SI: Ns/m). The unit for the rotational damping coefficient is **torque * time/angular displacement** (if using SI: Nms/rad)

```
dampingCoefficient = 70000

tire.setDamping(dampingCoefficient*2, agxModel.TwoBodyTire.RADIAL)
tire.setDamping(stiffness*10, agxModel.TwoBodyTire.LATERAL)
tire.setDamping(stiffness*4, agxModel.TwoBodyTire.BENDING)
tire.setDamping(stiffness*0.1, agxModel.TwoBodyTire.TORSIONAL)
```

31. AGX Model: PID Controller

This chapter describes the PID controller implementation in AGX Model.

In `agxModel::PidController1D` there is an implementation of a PID controller with the classic gains for proportional, integral, and derivative gain described by the following mathematical expression:

$$u(t) = MV(t) = K_p\, e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de}{dt},$$

where u is the out signal, also known as the Manipulated Variable (MV), and K_p, K_i, and K_d are the tuning parameters or proportional gain, integral gain, and derivative gain. In the figure below, Fig. 31.1, there is a schematic representation of the PID controller including the process the PID is controlling, known as the Plant.

The above equation is discretized with the forward Euler Method and this leads to an explicit solution of $u(t)$. The error, e, is calculated as the difference between the measured Process Variable (PV) on the Plant and the desired Set Point (SP), the derivative of the error is discretized as $\frac{de}{dt} = \frac{e - e_{n-1}}{t - t_{n-1}}$, where e_{n-1} is the error from the previous time step and $t - t_{n-1}$ is the time step. The integral part is approximated by adding up the error, e, multiplied by the time step. This explicit solution of $u(t)$ is relatively sensitive to high frequencies in the Plant.

The PID controller has an *integral windup* algorithm with both “clamping” and/or an integral back-calculation algorithm. The clamping is disabled per default and the back-calculation is activated if the PID controller are given constraints on the Manipulated Variable (MV). The backward calculation is done by recomputing the integral term to its previous state before the windup started and by that disabling the integration.

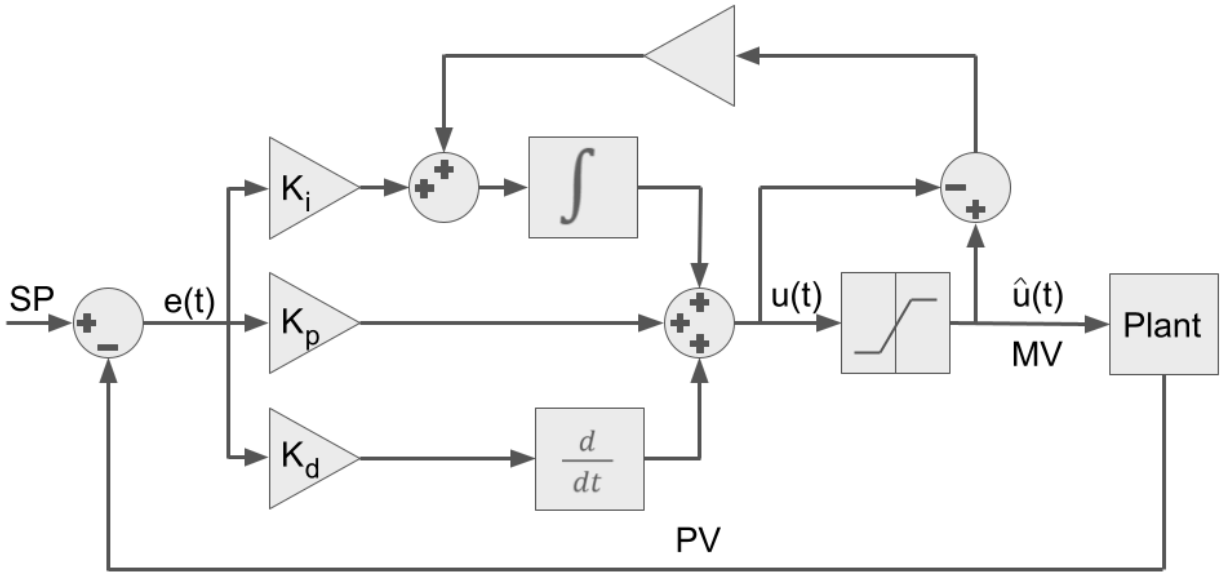


Fig. 31.1 Above is a schematic diagram over the PID controller. From left to right we have the error which is the difference between the Set Point (SP) and the Process Variable (PV). K represents the three gains and the block to the right represent the integral windup algorithm activated when the output signal, Manipulated Variable (MV), is outside the Plant input limits.

31.1. C++ example of PidController with ControllerHandler

It is possible to use the PID controller in a stand alone application but the *agxModel::PidController1D* is preferable to be used together with the *agxModel::ControllerHandler* which is a *StepEventListener*. The *agxModel::ControllerHandler* reads the Process Variable (PV) from the Plant, or controlled process, and lets the PID controller calculate a new Manipulated Variable (MV) with respect to the current Set Point (SP), and then that value is sent to the Plant.

In the tutorial below we create a pendulum with a load attached via a wire to a winch, see Fig. 31.2. The winch is controlled with a PID controller where the Plant is the winch together with the pendulum. The Manipulated Variable is the winch speed, the pendulum weight z-position is the Process Variable, and the Set Point is the desired height of the load above the xy-plane at z equal to zero. The winch base has an oscillating vertical motion which moves the pendulum up and down while the PID controller is counteracting this vertical motion plus the vertical motion due to the swing of the pendulum load. Note that the tutorial code doesn’t contain any graphical nodes.

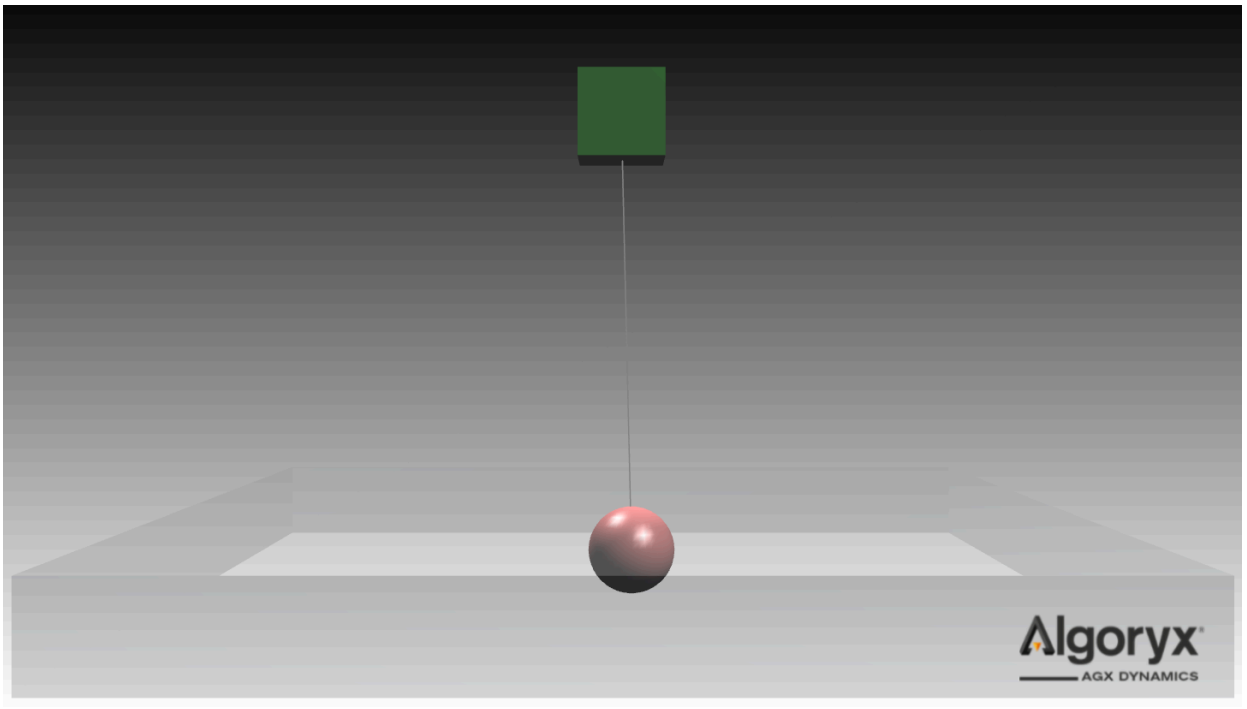


Fig. 31.2 This is the pendulum scene in the tutorial. The green cube represents the winch with an oscillating vertical motion. The red sphere is the pendulum load that the PID controller is stabilizing and the gray transparent box is the vertical Set Point for the PID controller.

The first part of our tutorial is an implementation of the *agxModel::ControllerHandler::Plant* base class that can read process data and set signal values on the system we are controlling. In this specific case the *Plant* Process Variable is the z-position of the pendulum weight and the Manipulated Variable is the winch speed controlling the length of the wire. The class overrides two abstract methods in *agxModel::ControllerHandler::Plant*, the methods *getProcessVariable* and *setManipulatedVariable*, see code below. Both the *getProcessVariable* and the *setManipulatedVariable* are called in the post step of the *agxModel::ControllerHandler*.

```
/// A thin implementation of the Plant base class. This implementation reads the
/// pendulum position and sets the winch speed.
class PlantWinch : public agxModel::ControllerHandler::Plant
{
public:
  PlantWinch(agxWire::Winch* winch, agx::RigidBody* pendulumWeight) :
      m_winch(winch), m_pendulumWeight(pendulumWeight)
  {
  }

  /// \return The current measured Process Variable (PV).
  agx::Real getProcessVariable(const agx::TimeStamp& time) override
  {
    // Update the pendulum base oscillating motion.
    // Note: It is not considered best practice to modify the simulation in this method.
    agx::Real winchBaseVelocity = 0.5 * std::cos(1 / 10.0 * time);
    m_winch->getRigidBody()->setVelocity(agx::Vec3(0, 0, winchBaseVelocity));

    // Here the Plant Value is the z position of the load attached to the wire in the winch.
    return m_pendulumWeight->getPosition().z();
  }

  /// Set the Manipulated Variable (MV) controlling the Plant, also known as Control Variable.
  void setManipulatedVariable(agx::Real manipulatedVariable) override
  {
    // The PID manipulate variable is set as the winch speed.
    // Negative since a negative error should pull in the weight.
    m_winch->setSpeed(-manipulatedVariable);
  }

private:
  agxWire::WinchRef m_winch;
  agx::RigidBodyRef m_pendulumWeight;
};
```

The next part of our example is to setup the pendulum and add a Control System with a PID controller to the simulation. In our control system, *agxModel::ControllerHandler*, the above specified *PlantWinch* is added together with an *agxModel::PidController1D*.

```
agxSDK::SimulationRef sim = new agxSDK::Simulation;

RigidBodyRef pendulumWeight = new RigidBody(new agxCollide::Geometry(new agxCollide::Sphere(1)));
pendulumWeight->setPosition(-0.5, 0.5, -10);

// Create wire to hang between the winch base and the winch load
agxWire::WireRef wire = new agxWire::Wire(0.015, 1, false);
// Attach wire to load
wire->add(new agxWire::BodyFixedNode(pendulumWeight));
// Wire nodes
wire->add(new agxWire::FreeNode(agx::Vec3(0, 0, 0)));

agx::RigidBodyRef winchBase = new agx::RigidBody("winchBase");
winchBase->add(new agxCollide::Geometry(new agxCollide::Box(agx::Vec3(1, 1, 1))));
winchBase->setMotionControl(agx::RigidBody::KINEMATICS);

agxWire::WinchRef winch = new agxWire::Winch(winchBase, agx::Vec3(0, 0, 0), agx::Vec3(0, 0, 1));
agx::Real pulledInLength = 10;
// attach wire to winch
wire->add(winch, pulledInLength);

// Create the Plant describing the process we should control. The class, PlantWinch,
// implements the base class agxModel::ControllerHandler::Plant which reads the
// pendulum z-position and sets the winch speed.
agxModel::ControllerHandler::PlantRef plant = new PlantWinch(winch, pendulumWeight);

agx::Real setPoint = -10;
agx::Real proportionalGain = 0.1;
agx::Real integralGain = 0.05;
agx::Real derivativeGain = 5;
agxModel::PidController1DRef pidController = new agxModel::PidController1D();
pidController->setGains(proportionalGain, integralGain, derivativeGain);
pidController->setSetPoint(setPoint);

agxModel::ControllerHandlerRef controller_handler =
  new agxModel::ControllerHandler(pidController, plant);
sim->add( pendulumWeight );
sim->add( winchBase );
sim->add( wire );
sim->add( controller_handler );
```

For the full source listing, see `tutorials/agxOSG/tutorial_pidController.cpp`.

31.2. Python example of PID control on Hinge with driveline

The PID controller class can be used stand alone as well. An example of this is shown with an arm that has one joint. The arm consists of a long and thin box connected to a static body with a hinge. To drive the hinge a drive line with an engine is created, connected to the hinge with a gear. The engine is mainly used since it makes it simple to control what torque is applied to the drive line. The torque is added as the only value in the engine lookup table.

```
hinge.getMotor1D().setEnable(False)

powerLine = agxPowerLine.PowerLine()
sim.add(powerLine)

engine = agxDriveTrain.Engine()
engine.setPowerGenerator(agxPowerLine.TorqueGenerator(engine.getRotationalDimension()))
engine.setThrottle(1)
lookupTable = engine.getPowerGenerator().getPowerTimeIntegralLookupTable()
powerLine.add(engine)

actuator = agxPowerLine.RotationalActuator(hinge)
gear = agxDriveTrain.HolonomicGear()
```

```
engine.connect(gear)
gear.connect(actuator)
```

The PID controller must know what angle the joint should have at what time. We let the arm move up to $\frac{\pi}{2}$ rad in 2 seconds, rest there for a second and then move down again in 2 seconds. To get a trajectory the controller gets one angle position for each time step, except when the arm should be standing still.

```
time = np.append(np.linspace(0, 2.0, 2000), np.linspace(3.0, 5.0, 2000))
angle = np.append(np.linspace(0, agx.PI/2, 2000), np.linspace(agx.PI/2, 0, 2000))
```

Having created a trajectory, it is possible to create a controller and add it to the simulation.

```
pid = agxModel.PidController1D()
pid.setGains(300.0, 50.0, 50.0)

simulation().add(ControlLoop(hinge, lookupTable, pid, time, angle))
```

The PID tuning parameters of the controller are only a suggestion, but running the example for the single joint arm with those values, should give the results seen in the figure below.

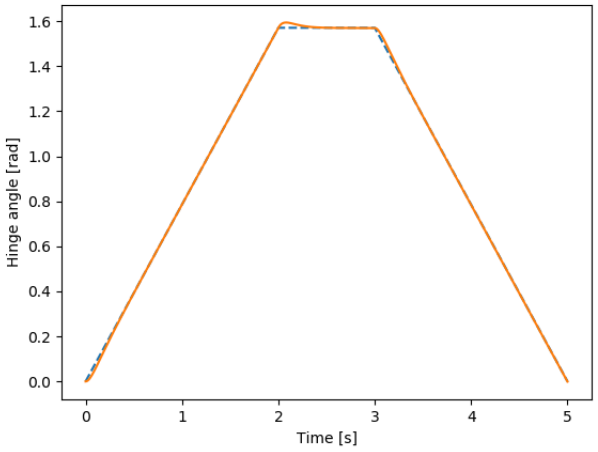


Fig. 31.3 Hinge angle for each time step. Dashed line is the reference.

For more details see `data/python/tutorials/tutorial_pid_controller.agxPy`.

31.3. Python example of PID Control of a robot

A natural extension to the previous example is to have multiple joints being controlled. To achieve that we instead use the `GenericRobot` from `agxPythonModules.robots.generic_robot`. Having a pre-existing trajectory file for the robot, it is possible to try to move the robot in the same pattern using a PID controller for each joint. Just as for the single joint example an engine in a drive train is connected to each hinge with a gear and in this case we set the engine inertia and gear ratio to a given value. Then a wrapper class `RobotJoint` is used to keep the hinges, lookup tables and PID:s grouped together per joint.



Fig. 31.4 The robot called generic robot.

```
# Save all the hinges of the robot in a list
hinges = [robot.hinges["bottom"], robot.hinges["bottomMiddle"], robot.hinges["middleTop"],
          robot.hinges["topHead"], robot.hinges["head"], robot.hinges["headPlate"]]

# Lists with PID parameters
# Different parameters can be used for the different joints and the
# parameters can be tuned if desired. Currently lists with 6 identical values.
Kp = [900] * 6
Ki = [200] * 6
Kd = [600] * 6

# List of values for motor inertia and gear ratio.
motorInertia = [0.27, 0.046, 0.036, 0.15, 0.15, 0.18]
gearRatio = [125.0, 171.0, 143.0, 60.0, 67.0, 50.0]

powerLine = agxPowerLine.PowerLine()
sim.add(powerLine)

joints = []
for i in range(0, len(hinges)):
    hinges[i].getMotor1D().setEnable(False)
    engine = agxDriveTrain.Engine()
    engine.setPowerGenerator(agxPowerLine.TorqueGenerator(engine.getRotationalDimension()))
    lookupTable = engine.getPowerGenerator().getPowerTimeIntegralLookupTable()
    engine.setThrottle(1.0)
    engine.setInertia(motorInertia[i])
    powerLine.add(engine)

    actuator = agxPowerLine.RotationalActuator(hinges[i])
    gear = agxDriveTrain.HolonomicGear()
    gear.setGearRatio(gearRatio[i])

    engine.connect(gear)
    gear.connect(actuator)

    joints.append(RobotJoint(hinges[i], lookupTable, Kp[i], Ki[i], Kd[i]))
```

A controller can then be created. The joints should all be put in a list, that is sent into the controller.

```
simulation().add(ControlLoop(joints, fullFileName))
```

Running the tutorial with the square example with the above controller should give the result seen in the figure below.

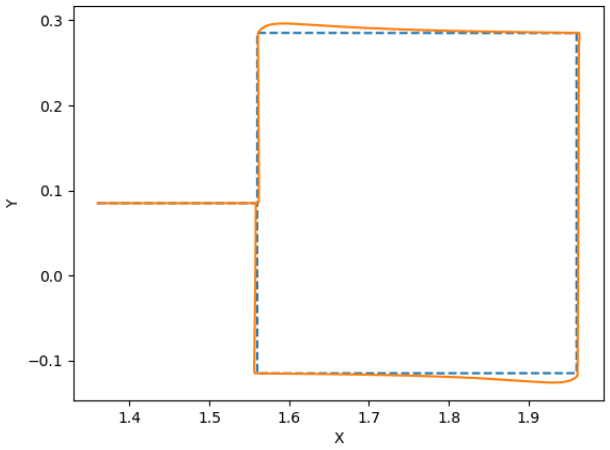


Fig. 31.5 Robot tool position in the x-y-plane. The values are in meters and the dashed line is the reference. The robot follows the straight line to the square, then moves clockwise around the square path.

For more details see `data/python/tutorials/RobotControl/tutorial_robot_pid_control.agxPy`.

32. AGX Sensor

The AGX Sensor module is used to simulate accurate real time sensors that allow detection of objects and changes connected to the physics simulation. To simulate sensors, an `agxSensor::Environment` will be used as a separate analogue to `agxSDK::Simulation`, to house references to all objects and features detectable by sensors, and the sensors themselves.

Currently, the AGX Sensor module supports the following sensor types:

- Lidar
- Inertial Measurement Unit (IMU)
- Accelerometer
- Gyroscope
- Magnetometer
- Encoder
- Odometer

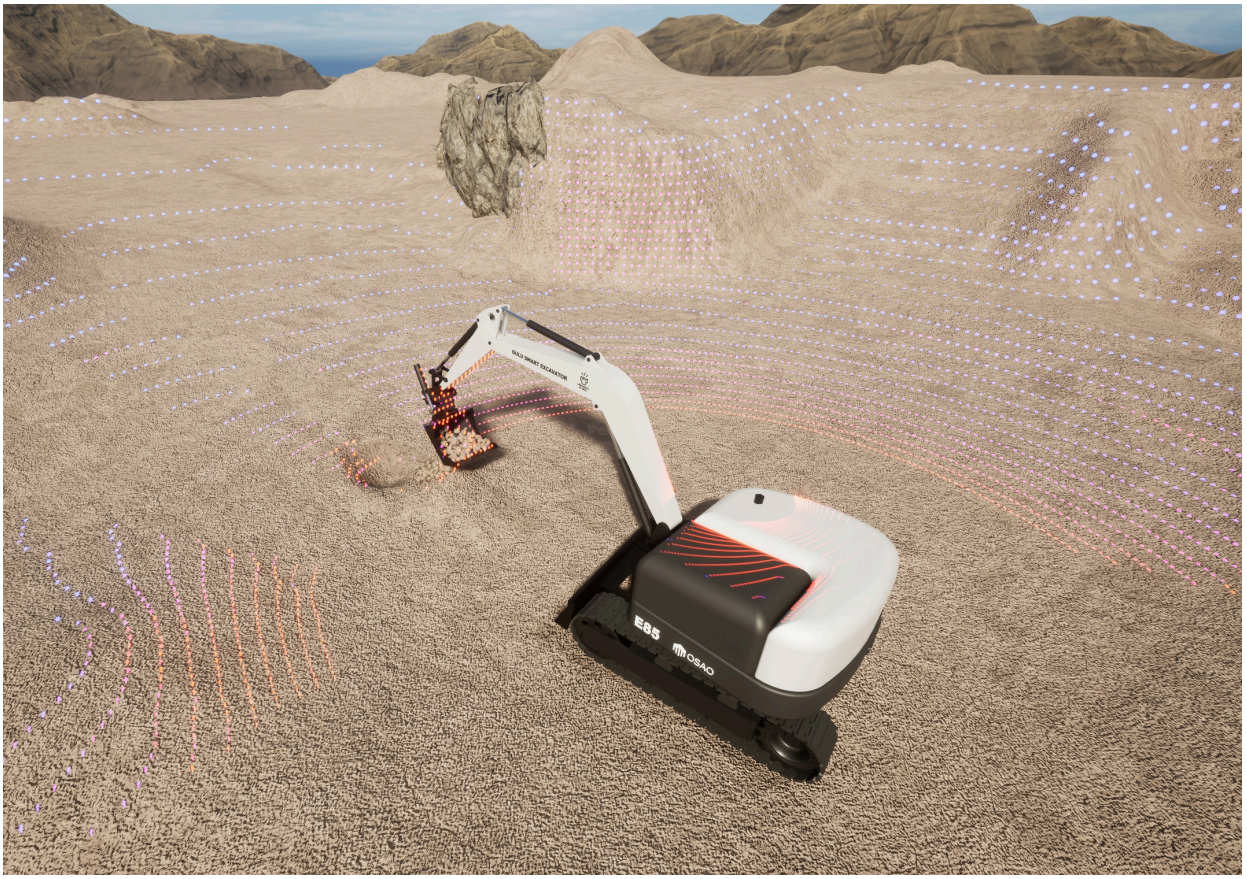


Fig. 32.1 Lidar attached to an excavator.

32.1. Environment

`agxSensor::Environment` primarily contains references to all objects that should be detectable by a sensor. It is associated with an `agxSDK::Simulation` and any object's movement in the `agxSDK::Simulation` will automatically be reflected in the `agxSensor::Environment`. To create an `agxSensor::Environment` for a specific `agxSDK::Simulation`, the procedure is as follows:

```
agxSensor::EnvironmentRef environment = agxSensor::Environment::getOrCreate( simulation );
```

Where `simulation` is an instance of an `agxSDK::Simulation`. For an object to be detectable by a sensor, it must be added to the `agxSensor::Environment` instance:

```
// Create a Rigid Body with a Box Geometry.
agxCollide::GeometryRef geometry = new agxCollide::Geometry( new agxCollide::Box( 1000, 1000, 1 ) );
```

```
agx::RigidBodyRef body = new agx::RigidBody( geometry );

// Add the Rigid Body to the agxSDK::Simulation.
simulation->add( body );

// Add the Rigid Body to the agxSensor::Environment.
environment->add( body );
```

The position and orientation of the Rigid Body will automatically be updated in the `agxSensor::Environment` as it is moved by the physics simulation handled by the `agxSDK::Simulation`.

Currently, the following types are supported and can be added to the `agxSensor::Environment`:

- `agx::Physics::GranularBodySystem`
- `agx::RigidBody`
- `agxCollide::Shape`
- `agxCollide::Geometry`
- `agxSDK::LinkedStructure`
    - `agxCable::Cable`
    - `agxModel::Beam`
    - `agxVehicle::Track`
- `agxTerrain::Terrain`
- `agxTerrain::TerrainPager`
- `agxWire::Wire`

Apart from adding objects to the `agxSensor::Environment`, any active sensor must also be added to it. How to create and configure an `agxSensor::Lidar` sensor is described in more detail later in the section about Lidar.

32.1.1. Surface Material

Surfaces in the real world reflect light at different intensities in different directions depending on the underlying material of the object and the characteristics of the object's surface. AGX Sensor allows modeling of a wide range of commonly occurring surfaces using surface materials derived from `agxSensor::RtSurfaceMaterial`. The various `agxSensor::RtSurfaceMaterial` types model the compound reflection characteristics of a surface, taking into account both the reflectivity of the underlying material and the microscopic features of the surface. Physically, the surface materials supplied as the various `agxSensor::RtSurfaceMaterial` types specify the Bidirectional Reflectance Distribution Function (BRDF), $f(\hat{l}, \hat{v})$ from the reflectance equation:

$$L(\hat{v}) = \int_{\Omega} L_i(\hat{l}) f(\hat{l}, \hat{v})(\hat{l} \cdot \hat{n}) \mathrm{d}\omega$$

Which describes the light $L(\hat{v})$ reflected on the surface in the solid angle Ω, from incident light $L_i(\hat{l})$.

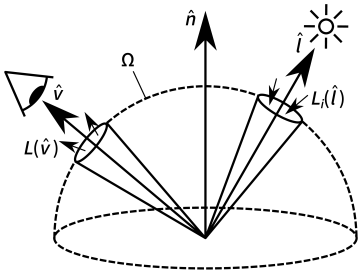


Fig. 32.2 Configuration of the reflectance equation.

Currently, AGX Sensor implements the following surface materials:

Material	Model	Parameter Count	Usage
`agxSensor::RtLambertianOpaqueMaterial`	Lambertian Diffuse	1	Most non-specular surfaces, especially matte diffuse surfaces.

`agxSensor::RtGgxAndOrenNayarMaterial`	GG-X and Oren-Nayar	5	Specular surfaces, rough diffuse surfaces or diffuse surfaces with a specular top-coat.
`agxSensor::RtBrdfExplicitMaterial`	Explicit BRDF	(samples)	Advanced fallback for opaque surfaces where the other models will not suffice.

32.1.1.1. Opaque Lambert Surface

The simplest of the `agxSensor::RtSurfaceMaterial` types comes on the form of the `agxSensor::RtLambertianOpaqueMaterial`. This material describes a simple matte diffuse Lambertian surface, with the BRDF as:

$$f(\hat{l}, \hat{v}) = \frac{C_D}{\pi}$$

For some total diffuse reflectivity parameter of C_D. This surface material should generally be sufficient to achieve a good result for most non-specular, that is not mirror-like, surfaces, and is the default material applied to all surfaces with $C_D = 0.8$ if no other material is specified.

To add an `agxSensor::RtLambertianOpaqueMaterial`, with a reflectivity of $C_D = 0.2$, to an object in the sensor environment, the procedure is as following:

```
// Create the Lambertian material
agxSensor::RtLambertianOpaqueMaterialRef material = agxSensor::RtLambertianOpaqueMaterial::create();

// Set the reflectivity
material->setReflectivity( 0.2f );

// Assign the material to the object
material->assignTo( object );
```

32.1.1.2. GG-X and Oren-Nayar Surface

For more complex surfaces the two-layer material `agxSensor::RtGgxAndOrenNayarMaterial` can be used. This material describes a surface with a GG-X microfacet specular top-layer and a secondary Oren-Nayar rough diffuse layer. The GG-X specular layer contributes to the BRDF according to:

$$f_{\text{Specular}}(\hat{l}, \hat{v}) = \frac{R_{\text{Fresnel}}(\hat{l}, \hat{h}) G_{\text{Smith}}(\hat{l}, \hat{v}, \hat{h}) D_{\text{GG-X}}(\hat{h})}{4(\hat{l} \cdot \hat{n})(\hat{v} \cdot \hat{n})}$$

Where R_{Fresnel} is the Fresnel reflectance, G_{Smith} is Smith's G-function and $D_{\text{GG-X}}$ is the GG-X microfacet distribution function. Meanwhile, the Oren-Nayar diffuse layer contributes to the total BRDF according to:

$$f_{\text{Diffuse}}(\hat{l}, \hat{v}) = \frac{C_{\text{D}}}{\pi}(\hat{l} \cdot \hat{n})(A_{\text{Oren-Nayar}} + B_{\text{Oren-Nayar}} \frac{s_{\text{Oren-Nayar}}}{t_{\text{Oren-Nayar}}})$$

Where $A_{\text{Oren-Nayar}}$, $B_{\text{Oren-Nayar}}$, $s_{\text{Oren-Nayar}}$ and $t_{\text{Oren-Nayar}}$ are the various sub-sections of the Oren-Nayar diffuse model.

As this surface material is significantly more complex than the Lambertian diffuse material, see Opaque Lambert Surface, the parameter count is likewise greater. The parameters for the `agxSensor::RtGgxAndOrenNayarMaterial` are as following:

Parameter	Method	Description	Default
n	`setRefractiveIndexReal`	Real component of the specular top-layer refractive index.	1.4517
k	`setRefractiveIndexImaginary`	Imaginary component of the specular top-layer refractive index. This value is related to absorption of light at the top-layer.	0.0
σ_{Beckman}	`setBeckmanRoughness`	Roughness value of the specular top-layer. This value represents the root-mean-square slope of the specular top-layer microfacets.	0.3
$\sigma_{\text{Oren-Nayar}}$	`setOrenNayarRoughness`	Roughness value of the diffuse secondary layer. This value represents the standard deviation of the normal vectors of the diffuse layer microfacets.	0.3
C_D	`setDiffuseReflectivity`	Reflectivity of the diffuse secondary layer.	0.8

To add an `agxSensor::RtGgxAndOrenNayarMaterial` to an object in the sensor environment, the procedure is as following:

```
// Create the GG-X and Oren-Nayar material
agxSensor::RtGgxAndOrenNayarMaterial material = agxSensor::RtGgxAndOrenNayarMaterial::create();
```

```
// Set the material parameters (Silver mirror)
material->setRefractiveIndexReal( 0.037f )
         .setRefractiveIndexImaginary( 5.57f )
         .setBeckmanRoughness( 0.001f )
         .setOrenNayarRoughness( 0.0f )
         .setDiffuseReflectivity( 0.02f );

// Assign the material to the object
material->assignTo( object );
```

32.1.1.3. Explicit BRDF Surface

If neither of the other surface materials would suffice, the BRDF can also be explicitly specified using `agxSensor::RtBrdfExplicitMaterial`. This material simply accepts BRDF samples on the form of a four dimensional array, flattened in memory, where the indices specify:

0. Sample index in θ (vertical) direction for the light incidence direction.
1. Sample index in ϕ (horizontal) direction for the light incidence direction.
2. Sample index in θ (vertical) direction for the view direction.
3. Sample index in ϕ (horizontal) direction for the view direction

The supplied samples should represent a non-isotropic BRDF, thus specifying the full ϕ range from 0 to 2π. θ ranges from 0 to $\frac{\pi}{2}$. BRDF samples are suitably attained from measurement data.

32.1.2. Ambient Material

The medium through which the light travels to, or from, a sensor may contain disturbances which can affect the measured data. AGX Sensor supplies a simple way to model homogeneous atmospheric disturbances through `agxSensor::RtAmbientMaterial`. The `agxSensor::RtAmbientMaterial` has a number of parameters which control the behavior of light signals in the medium:

Parameter	Method	Description	Default
n	`setRefractiveIndex`	Ambient material refractive index.	1.000273
α	`setAttenuationCoefficient`	Signal attenuation coefficient.	0.000402272 m^{-1}
A	`setReturnProbabilityScaling`	Atmospheric return gamma-distribution scaling.	1.58899 * 10^{-5}
k	`setReturnGammaDistributionShapeParameter`	Atmospheric return gamma-distribution shape parameter.	9.5
θ	`setReturnGammaDistributionScaleParameter`	Atmospheric return gamma-distribution scale parameter.	0.52 m

The refractive index n and attenuation coefficient α describe the general propagation speed and attenuation of the signal. Meanwhile, the A, k and θ parameters describe the probability at, and location where, the signal may spontaneously reflect in the medium.

To simplify the configuration of ambient materials, `agxSensor::RtAmbientMaterial` also specifies a number of helper methods for common weather phenomena:

Method	Description
`configureAsAir`	Configures the `agxSensor::RtAmbientMaterial` with the transmission parameters expected in regular air at a given visibility, as expected by inverse meteorological optical range (MOR).
`configureAsFog`	Configures the `agxSensor::RtAmbientMaterial` with the transmission parameters expected in foggy conditions, as described by the Al-Naboulsi model, at a given visibility and signal wavelength. A maritimeness parameter can be used to specify an interpolation value between fine droplet continental radiation fog (0.0) and large droplet maritime fog (1.0).
`configureAsRainfall`	Configures the `agxSensor::RtAmbientMaterial` with the transmission parameters expected in rainy conditions, as described by the Carbonneau model, at a given precipitation rate. A tropicalness parameter can be used to specify an interpolation value between common rain of smaller drop size (0.0) and tropical rain of larger drop size (1.0).
`configureAsSnowfall`	Configures the `agxSensor::RtAmbientMaterial` with the transmission parameters expected in snowy conditions, as described by the Carbonneau model, at a given precipitation rate and signal wavelength.

An `agxSensor::RtAmbientMaterial`, configured for continental fog of 2.0 km visibility, is added to the sensor environment through the following procedure:

```
// Create the ambient material
agxSensor::RtAmbientMaterialRef material = agxSensor::RtAmbientMaterial::create();

// Configure as fog for a 905 nm signal
material->configureAsFog( 2.0f, 905.0f );

// Add to sensor environment
environment->getScene()->setMaterial( material );
```

32.1.3. Magnetic Field

For magnetometers to work properly, the `agxSensor::Environment` has a magnetic field specified that spans the Environment. This magnetic field is per default set as an `agxSensor::UniformMagneticField` instance, see Uniform Magnetic Field, with its uniform magnetic field vector mimicking one that could be found on the surface of the Earth. To change the Environment's magnetic field from the default, the magnetic field is simply set for the Environment as:

```
// Configure the magnetic field in the environment
environment->setMagneticField( magneticField );
```

Where `environment` is an instance of an `agxSensor::Environment` and `magneticField` is an instance of a type derived from `agxSensor::MagneticField`. To specify the `magneticField` instance, one either use one of the pre-defined magnetic field types, or create one's own type inheriting from `agxSensor::MagneticField`.

32.1.3.1. Uniform Magnetic Field

An `agxSensor::UniformMagneticField` instance defines a simple magnetic field that is uniform across the entire sensor Environment, and suitable for fairly local navigation. This magnetic field instance is created by specifying a magnetic field vector that will be used across the entire Environment:

```
// Find a suitable magnetic field vector (in tesla)
agx::Vec3 magneticFieldVector{ -19.462e-6, 44.754e-6, 7.8426e-6 };

// Create a uniform magnetic field
agxSensor::UniformMagneticFieldRef magneticField =
  new agxSensor::UniformMagneticField( magneticFieldVector );

// Configure the magnetic field in the environment
environment->setMagneticField( magneticField );
```

32.1.3.2. Dipole Magnetic Field

For longer range navigation it may sometimes be viable to use a dipole approximation of the Earth's magnetic field, using an instance of `agxSensor::DipoleMagneticField`. This magnetic field instance is created by specifying a magnetic moment vector and the center position of the dipole in world coordinates:

```
// Find a suitable magnetic moment vector (in m²A)
agx::Vec3 magneticMoment{ 2.69e19, -7.65e22, 1.5e22 };

// Find a suitable center for the dipole
agx::Vec3 dipoleCenter{ -1.9e-10, 20.79e3, -6.369e6 };

// Create a dipole magnetic field
agxSensor::DipoleMagneticFieldRef magneticField =
  new agxSensor::DipoleMagneticField( magneticMoment, dipoleCenter );

// Configure the magnetic field in the environment
environment->setMagneticField( magneticField );
```

32.1.4. Different Rate Stepping

In certain cases, such as when running the dynamics simulation at a very high step rate, it may be useful to run certain sensors at a different, lower, step rate. This may be especially useful for sensors that rely on synchronization between the CPU and GPU, such as the Lidar. One way to achieve this is by

adding the sensors to an `agxSensor::SensorGroupStepStride`, which steps the sensors at a lower rate compared to the `agxSensor::Environment` as specified by a stride value:

```
// How many environment steps per sensor step?
agx::UInt32 stride = 5u;

// Create the group
agxSensor::SensorGroupStepStrideRef group = new agxSensor::SensorGroupStepStride( stride );

// Add the sensor to the environment through the group
group->add( sensor );
environment->add( group );
```

Any sensor added to an `agxSensor::SensorGroupStepStride` will produce output at the lower step rate of the group, and thus not at the full environment step rate.

32.2. Lidar

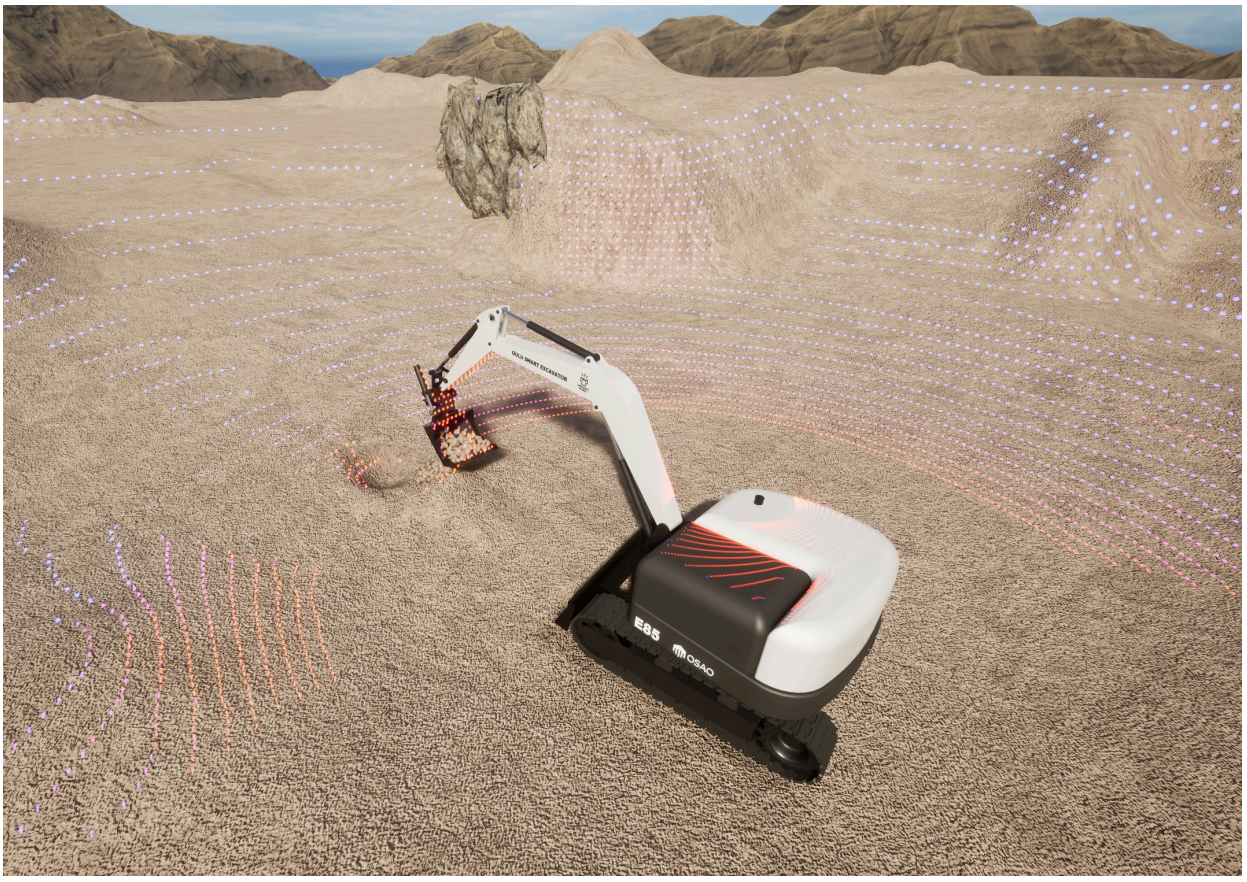


Fig. 32.3 Lidar attached to an excavator.

`agxSensor::Lidar` is a sensor type in the AGX Sensor module. It uses GPU accelerated ray tracing to enable accurate real time simulation of Lidar and provides a means to access the generated point cloud data. In order to be active and able to detect objects within an environment, the Lidar must be added to the `agxSensor::Environment`:

```
agxSensor::LidarRef lidar = new agxSensor::Lidar( frame, model );
environment->add( lidar );
```

Where `environment` is an instance of an `agxSensor::Environment`, `frame` is an optional `agx::Frame` and `model` is an `agxSensor::LidarModel`. The `frame` parameter can be set to, for example, attach the Lidar to anything moving in the simulation, such as a Rigid Body. Meanwhile, the `model` determines the behavior and characteristics of the Lidar.

Specifying which output a Lidar should produce, and later accessing the resulting point cloud data, is done using the `agxSensor::RtOutputHandler`, see Output Handler for details.

An example of how to set up a Lidar simulation scenario in Python can be found in `Algoryx/AGX-version/data/python/tutorial_lidar.agxPy`. A C++ tutorial can be found in `Algoryx/AGX-version/tutorials/tutorial_lidar.cpp`.

32.2.1. Lidar Model

An `agxSensor::LidarModel` is what determines the behavior and characteristics of an `agxSensor::Lidar`. A number of built in Lidar Models are available, below is an example of how to use the built-in `agxSensor::LidarModelOusterOS1` to create a horizontally sweeping Lidar mimicking the Ouster OS1 Lidar:

```
agxSensor::LidarModelRef model = new agxSensor::LidarModelOusterOS1();
agxSensor::LidarRef lidar = new agxSensor::Lidar( frame, model );
environment->add( lidar) ;
```

Where `environment` is an instance of an `agxSensor::Environment` and `frame` is an optional `agx::Frame`, which can be set to, for example, attach the Lidar to anything moving in the simulation, such as a Rigid Body.

It is also possible to create custom Lidar Models where, for example, a custom ray pattern can be implemented:

```
agxSensor::LidarModelRef model = new agxSensor::LidarModel( myCustomRayPatternGenerator,
                                                            agx::RangeReal32{ 0.5, 30.0 },
                                                            lidarProperties,
                                                            lidarOutputNoises );
```

Where `myCustomRayPatternGenerator` is an implementation of Ray Pattern Generator, which configures when and in which direction the Lidar sensor will emit its rays, `lidarProperties` is an optional parameter specifying additional properties of the Lidar and `lidarOutputNoises` is an optional `agxSensor::RtOutputNoiseRefVector` which can be used to define distortions applied to the Lidar output on a per-model basis. For more details about the Ray Pattern Generator, see Ray Pattern Generator.

32.2.2. Predefined Lidar Models

32.2.2.1. Ouster

A number of predefined Lidar models based on popular Lidar sensors are included with the AGX Sensor module, each with their own set of hardware configurations and software settings.

- **Ouster OS0** - Ultra-Wide View High-Resolution Imaging Lidar
- **Ouster OS1** - Mid-Range High-Resolution Imaging Lidar
- **Ouster OS2** - Long-Range High-Resolution Imaging Lidar

The OSx series currently have the following predefined configuration options:

- Hardware configuration of *32*, *64* or *128* channels
- Beam spacing options *uniform*, *above horizon* and *below horizon*
- Configurable frequency / horizontal resolution settings *512x10*, *512x20*, *1024x10*, *1024x20*, *2048x10*

32.2.2.2. Livox

The Livox lidar models included with the AGX Sensor module are simulated using a repeating but large set of ray angles, coming close to the non-repeating pattern displayed by the Livox hardware lidars. The only configuration option available is the **Downsample** option, which skips ray angles in the pattern in order to get the same shape but less points per second. The list of available models are:

- **Livox Avia** - Mid-range 70 by 77 degree fov lidar with non-repeating pattern.
- **Livox HAP** - Mid-range 120 by 25 degree fov lidar with non-repeating pattern for automotive applications.
- **Livox Horizon** - Mid-to-long-range 82 by 25 degree fov lidar with non-repeating pattern.
- **Livox Mid40** - Short-to-mid-range circular 38 degree fov lidar with non-repeating pattern.
- **Livox Mid70** - Short-to-mid-range circular 70 degree fov lidar with non-repeating pattern.
- **Livox Mid360** - Short range 360 by 59 degree fov lidar with non-repeating pattern.
- **Livox Tele** - Long range 14 by 16 degree fov lidar with non-repeating pattern.

The ray pattern files are sampled from the Livox Sensor SDK and then included with this package compressed to binary. The ray pattern files are released by Livox under MIT license. These same files are under the same license in this module.

32.2.3. Ray Pattern Generator

The `agxSensor::LidarRayPatternGenerator` is responsible for generating the ray pattern used by the Lidar. Either, one of the built-in Ray Pattern Generators such as `agxSensor::LidarRayPatternHorizontalSweep` can be used, or a custom Ray Pattern Generator can be implemented by the

user.

The ray pattern is represented by an array of transforms stored within the Ray Pattern Generator. For each simulation step, an interval within this array is chosen to be active during that simulation step. This way, the whole ray pattern can be set once, and parts of the ray pattern can then be activated each simulation step in an efficient manner.

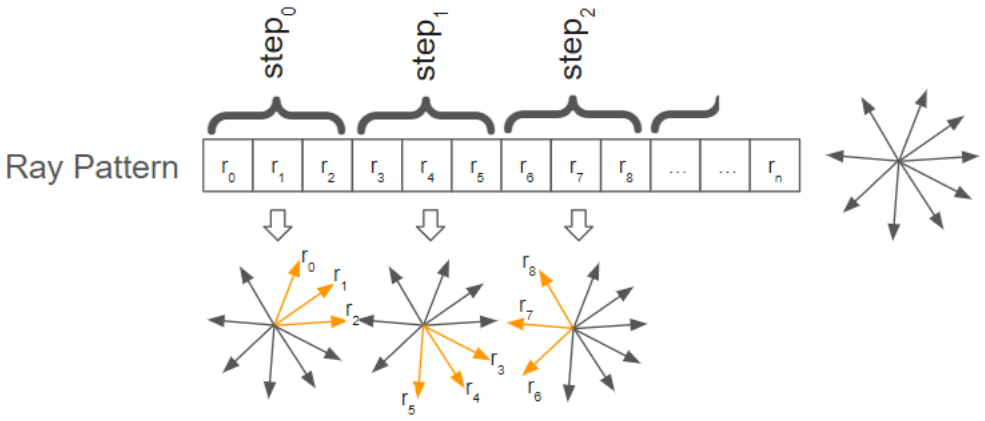


Fig. 32.4 Different parts of a ray pattern is active during different simulation steps.

The order of the rays stored in the `agxSensor::LidarRayPatternGenerator` determines the order of the output data once it is read via the Output Handler. Each ray is cast from the position determined by the stored transform origin and is cast along its z axis. The ray transforms are relative to the transform of the Lidar.

Each simulation step, during **PRE_STEP**, see Step Events, `getNextInterval()` in the `agxSensor::LidarRayPatternGenerator` instance will be called, and it is the responsibility of the Ray Pattern Generator to return an interval within the stored ray pattern array. The ray pattern array is set by calling `setRayTransforms()` in the `agxSensor::LidarRayPatternGenerator` instance. The ray pattern array can be set once, for example in the constructor of any Ray Pattern Generator, or it can be set more frequently, for example each time `getNextInterval()` is called. This is up to the implementation of the Ray Pattern Generator to decide, as long as the interval returned by `getNextInterval()` is within the currently set ray pattern array. By setting the ray pattern array only once, performance can be improved.

32.2.4. Ray Distortion Handler

The `agxSensor::LidarRayDistortionHandler` of a Lidar can be used to add distortions to the Lidar rays before they are emitted from the sensor.

32.2.4.1. Angle Gaussian Noise

An `agxSensor::LidarRayAngleGaussianNoise` instance can be added to the Lidar's distortion handler to apply Gaussian noise to the emission angles of the Lidar rays:

```
// Create a ray angle noise with 0.02 radians standard deviation
agxSensor::LidarRayAngleGaussianNoiseRef noise = new agxSensor::LidarRayAngleGaussianNoise();
noise.setStandardDeviation( 0.02f );

// Set perturbation vector to X (vertical noise)
noise.setAxis( agxSensor::LidarRayAngleGaussianNoise::Axis::AXIS_X );

// Add noise to the Lidar
lidar->getRayDistortionHandler()->add( noise) ;
```

The noise added to the ray angles through `agxSensor::LidarRayAngleGaussianNoise` will alter the angle of the Lidar rays without compensating the output, and thus the angle perturbations will be visible in the resulting Lidar output.

32.2.5. Output Handler

The `agxSensor::RtOutputHandler` of a Lidar is used to define what data will be generated during ray tracing. It also provides a convenient way of accessing this data.

The Output Handler can be accessed by calling:

```
lidar->getOutputHandler();
```

where `lidar` is an instance of `agxSensor::Lidar`.

The output of a Lidar is defined using one or several fields defined in `agxSensor::RtOutput::Field`. The field `XYZ_VEC3_F32` is used when the (local) locations of points are of interest, `INTENSITY_F32` is used when intensity should be included etc. These fields can be combined in any order to

configure the output of the Lidar. The type of the output is given by the user, and it must have memory alignment matching the fields used to define the output. Below is a minimal example of this, for a more detailed example, see the example and tutorial listed in Lidar:

```
lidar->getOutputHandler()->add<MyStructType, RtOutput::XYZ_VEC3_F32, RtOutput::INTENSITY_F32>();
```

Here, `MyStructType` must have a memory layout of four `float` contiguous in memory. Three `float` members where the x, y and z will be written to and a fourth `float` member to which the intensity will be written. It could for example also have been of type `agx::Vec4f`, or any other type fulfilling this requirement.

Accessing the output data of a Lidar can also be done using the Output Handler. This should be done in each **POST_STEP** of the `agxSDK::Simulation` time step. Below is a minimal example of this, for a more detailed example, see the example and tutorial listed in Lidar:

```
auto view = lidar->getOutputHandler()->view<MyStructType>();
for ( const auto& p : view )
  // Access p which is the MyStructType data (point) for this ray.
```

Note that calling `view` is blocking, i.e. it waits until the Lidar output is available, which is generated on the GPU.

32.2.5.1. Distance Gaussian Noise

Gaussian noise can be added to the distance, from the sensor, at which the points in Lidar data will appear by adding an `agxSensor::RtDistanceGaussianNoise` instance to the Lidar's output handler:

```
// Create a distance noise with a base standard deviation of 0.005 m...
agxSensor::RtDistanceGaussianNoiseRef noise = new agxSensor::RtDistanceGaussianNoise();
noise->setStdDevBase( 0.005f );

// ...growing with 0.001 m/m
noise->setStdDevSlope( 0.001f );

// Add noise to the Lidar output
lidar->getOutputHandler()->add( noise );
```

32.2.5.2. Sending output over ROS2

It is possible to use ROS2 to publish point cloud data from lidar sensors via the functions available in the `agxROS2` module. For more details see ROS2 and PointCloud2 usage with agxSensor.

32.3. Inertial Measurement Unit

Inertial Measurement Units (IMUs) can appear in multiple different configurations, both of sub-sensors and output formatting, depending on manufacturer and requirement specifications. `agxSensor::IMU` in the AGX Sensor module provides a way to model most standard IMUs, and provides a standardized way to collect signals from the IMU's sub-sensors, providing the readings through the common IMU interface. An `agxSensor::IMU` is attached to a frame, and in order to be active, it must be added to the `agxSensor::Environment`:

```
agxSensor::IMURef imu = new agxSensor::IMU( frame, model );
environment->add( imu );
```

Where `environment` is an instance of an `agxSensor::Environment`, `frame` is an optional `agx::Frame` and `model` is an `agxSensor::IMUModel`. The `frame` parameter can be set to, for example, attach the IMU to anything moving in the simulation, such as a Rigid Body. Meanwhile, the `model` determines the behavior and characteristics of the IMU.

Specifying which output an IMU should produce, and later accessing the resulting signal, is done using the IMU's `agxSensor::IMUOutputHandler`, see Output Handler for details.

32.3.1. IMU Model

An `agxSensor::IMUModel` is what determines the basic characteristics of an `agxSensor::IMU`. It is the `agxSensor::IMUModel` which is used to specify which sub-sensors are attached on the IMU, and in which order they are accessed when assembling the output signal. Currently, an `agxSensor::IMUModel` can hold attachments for the following sensors:

- Accelerometer, through an `agxSensor::IMUModelAccelerometerAttachment`
- Gyroscope, through an `agxSensor::IMUModelGyroscopeAttachment`
- Magnetometer, through an `agxSensor::IMUModelMagnetometerAttachment`

When constructing an `agxSensor::IMUModel`, the attachments are supplied through an `agxSensor::IMUModelSensorAttachmentRefVector`:

```
// Specify a model attachment vector
agxSensor::IMUModelSensorAttachmentRefVector attachments{
  new agxSensor::IMUModelAccelerometerAttachment( aDisplacement, accelerometerModel ),
  new agxSensor::IMUModelMagnetometerAttachment( mDisplacement, magnetometerModel )
};

// More attachments can naturally be added to the vector if needed
attachments.push_back(
  new agxSensor::IMUModelAccelerometerAttachment( aTwoDisplacement, accelerometerTwoModel )
);

// Create the IMU model
agxSensor::IMUModelRef model = new agxSensor::IMUModel( attachments );
```

Each attachment supplied to the IMU model can be specified with a local displacement of the attachment using an `agx::AffineMatrix4x4`. Multiple of the same sensor type, such as multiple Accelerometers, can be attached to the IMU model.

32.3.2. Output Handler

To specify the output of an `agxSensor::IMU` the `agxSensor::IMUOutputHandler` is used. To it, output instances of type of `agxSensor::IMUOutput` are added. These output instances specify which signals from the sub-sensors on the IMU that should be combined, and in which order:

```
agxSensor::IMUOutputRef output = new agxSensor::IMUOutput();
output->build<agxSensor::IMUOutput::ACCELEROMETER_Z_F64,
              agxSensor::IMUOutput::GYROSCOPE_X_F64,
              agxSensor::IMUOutput::GYROSCOPE_Y_F64>();
```

Each of the sub-sensor outputs can also be specified using the numerical order in which their attachments were specified in the attachment vector provided to the `agxSensor::IMUModel`. This may be useful in the cases where multiple of the same sensor type are present on the same IMU:

```
// An IMU model with two accelerometers
agxSensor::IMUModelRef model = new agxSensor::IMUModel(
  agxSensor::IMUModelSensorAttachmentRefVector{
    new agxSensor::IMUModelAccelerometerAttachment( aDisplacement, accelerometerModel ),
    new agxSensor::IMUModelAccelerometerAttachment( aTwoDisplacement, accelerometerTwoModel )
  }
);

// An output reading from sensor attachment #0 (first accelerometer) and #1 (second accelerometer)
agxSensor::IMUOutputRef output = new agxSensor::IMUOutput();
output->build<agxSensor::IMUOutput::SENSOR_0_Z_F64,
              agxSensor::IMUOutput::SENSOR_1_Z_F64>();
```

When specifying sub-sensor outputs using the sensor type specific fields, such as `agxSensor::IMUOutput::ACCELEROMETER_X_F64` or `agxSensor::IMUOutput::MAGNETOMETER_X_F64`, these fields will refer to the first instance of each sub-sensor of that type, regardless of whether that specific sub-sensor provides all three axes of output or not.

If outputs are specified for non-existent sensor, or non-existent axes, the output signal for those entries will be padded by a padding value. This value defaults to signaling NaN, but is possible to manually specify for an `agxSensor::IMUOutput` as:

```
// Specify custom missingno
output->setPaddingValue( -1e100 );
```

The output is then added to the IMU's output handler in the same vain as for all other sensors, and can be read from during the **POST_STEP** of the `agxSDK::Simulation` time step, using the unique identifier assigned to that output entry:

```
// Configure IMU output
agxSensor::IMUOutputRef output = new agxSensor::IMUOutput();
output->build<agxSensor::IMUOutput::ACCELEROMETER_Z_F64>();
imu->getOutputHandler()->add( 1, output );

// ... run simulation ...
```

```
// Access IMU readings in POST_STEP
auto view = imu->getOutputHandler()->view<agx::Real>( 1 );
for ( const auto& v : view )
  // Use IMU measurement v.
```

32.4. Accelerometer

`agxSensor::Accelerometer` is a sensor type in the AGX Sensor module. It provides a standardized way to measure the accelerations applied to a specified frame over time, with up to three degrees of freedom. In order to be active, the Accelerometer must be added to the `agxSensor::Environment`:

```
agxSensor::AccelerometerRef accelerometer = new agxSensor::Accelerometer( frame, model );
environment->add( accelerometer );
```

Where `environment` is an instance of an `agxSensor::Environment`, `frame` is an optional `agx::Frame` and `model` is an `agxSensor::AccelerometerModel`. The `frame` parameter can be set to, for example, attach the Accelerometer to anything moving in the simulation, such as a Rigid Body. Meanwhile, the `model` determines the behavior and characteristics of the Accelerometer.

Specifying which output an Accelerometer should produce, and later accessing the resulting signal, is done using the Accelerometer's `agxSensor::AccelerometerOutputHandler`, see Output Handler for details.

In accordance with the SI unit usage of the rest of AGX Dynamics, this Accelerometer provides its measurements in units of m/s² as default, instead of g. This can be adjusted by rescaling the signal using an `agxSensor::TriaxialSignalScaling` instance added to the output handler, see Signal Scaling for details.

32.4.1. Accelerometer Model

An `agxSensor::AccelerometerModel` is what determines the basic characteristics of an `agxSensor::Accelerometer`:

```
agxSensor::AccelerometerModelRef model =
  new agxSensor::AccelerometerModel( range,
                                     crossAxisSensitivity,
                                     zeroGBias,
                                     outputModifiers );
```

Where `range` is an `agxSensor::TriaxialRange` that specifies the Accelerometer's measurement range, while `zeroGBias` is an `agx::Vec3` specifying the measurement bias in each of the three axes at zero external acceleration. `crossAxisSensitivity` is an `agxSensor::TriaxialCrossSensitivity` that specifies how much an acceleration applied to one axis bleeds over into the signal of another axis. The optional `outputModifiers` parameter supplied to the model can be used to define modifiers applied to the output signal on a per-model basis. These modifiers are allowed to be the same as the ones that can be added to the Output Handler:

- Total Gaussian Noise
- Signal Scaling
- Gaussian Spectral Noise

32.4.2. Output Handler

The `agxSensor::AccelerometerOutputHandler` of an Accelerometer is used to specify the output signal of the Accelerometer, and which axes it should contain. The fields of `agxSensor::TriaxialOutput::Field` can be combined in any order to form an output signal containing the desired information. For example a single axis Accelerometer output can be added with the unique identifier of `1` to the Accelerometer as:

```
accelerometer->getOutputHandler()->add<agx::Real, agxSensor::TriaxialOutput::Z_VALUE_F64>( 1 );
```

The output data from the Accelerometer can then be accessed through the `agxSensor::AccelerometerOutputHandler`, in the **POST_STEP** of the `agxSDK::Simulation` time step, using the unique identifier assigned to that data:

```
auto view = accelerometer->getOutputHandler()->view<agx::Real>( 1 );
for ( const auto& v : view )
  // Use accelerometer measurement v.
```

32.4.2.1. Total Gaussian Noise

To model base level electrical noise in the measurement signal, an `agxSensor::TriaxialGaussianNoise` instance can be added to the `agxSensor::Accelerometer`'s output handler:

```
agxSensor::TriaxialGaussianNoiseRef noise =
  new agxSensor::TriaxialGaussianNoise( agx::Vec3( 1.456 * 1e-6 ) );

accelerometer->getOutputHandler()->add( noise );
```

The `agxSensor::TriaxialGaussianNoise` allows the noise RMS to be specified independently for each of the Accelerometer axes.

32.4.2.2. Signal Scaling

The Accelerometer output signal can be rescaled by adding an `agxSensor::TriaxialSignalScaling` instance to the output handler of the `agxSensor::Accelerometer`. This scaling can be used to conveniently convert from AGX Dynamics output in the SI unit of m/s² to the common Accelerometer reading unit of g:

```
agx::Vec3 perAxisScaling = agx::Vec3( TriaxialSignalScaling::CONVERT_METER_PER_SECOND_SQUARED_TO_G );

agxSensor::TriaxialSignalScalingRef scaling =
  new agxSensor::TriaxialSignalScaling( perAxisScaling );

accelerometer->getOutputHandler()->add( scaling );
```

The `agxSensor::TriaxialSignalScaling` allows the signal scaling to be specified independently for each of the Accelerometer axes.

32.4.2.3. Gaussian Spectral Noise

Commonly, Accelerometers come with a specified output noise spectral density, describing the expected noise at a given sample frequency. A Gaussian distributed noise using this density specification can be added to the output of an `agxSensor::Accelerometer` using an `agxSensor::TriaxialSpectralGaussianNoise`:

```
agxSensor::TriaxialSpectralGaussianNoiseRef noise =
  new agxSensor::TriaxialSpectralGaussianNoise( agx::Vec3( 2.943 * 1e-3 ) );

accelerometer->getOutputHandler()->add( noise );
```

The `agxSensor::TriaxialSpectralGaussianNoise` allows the noise density RMS to be specified independently for each of the Accelerometer axes.

32.5. Gyroscope

`agxSensor::Gyroscope` is a sensor type in the AGX Sensor module. It provides a standardized way to measure the angular rotation rate of a specified frame over time, with up to three degrees of freedom. In order to be active, the Gyroscope must be added to the `agxSensor::Environment`:

```
agxSensor::GyroscopeRef gyroscope = new agxSensor::Gyroscope( frame, model );
environment->add( gyroscope );
```

Where `environment` is an instance of an `agxSensor::Environment`, `frame` is an optional `agx::Frame` and `model` is an `agxSensor::GyroscopeModel`. The `frame` parameter can be set to, for example, attach the Gyroscope to anything moving in the simulation, such as a Rigid Body. Meanwhile, the `model` determines the behavior and characteristics of the Gyroscope.

Specifying which output a Gyroscope should produce, and later accessing the resulting signal, is done using the Gyroscope's `agxSensor::GyroscopeOutputHandler`, see Output Handler for details.

In accordance with the SI unit usage of the rest of AGX Dynamics, this Gyroscope provides its measurements in units of radians/s as default, instead of degrees/s. This can be adjusted by rescaling the signal using an `agxSensor::TriaxialSignalScaling` instance added to the output handler, see Signal Scaling for details.

32.5.1. Gyroscope Model

An `agxSensor::GyroscopeModel` is what determines the basic characteristics of an `agxSensor::Gyroscope`:

```
agxSensor::GyroscopeModelRef model =
  new agxSensor::GyroscopeModel( range,
                                 crossAxisSensitivity,
```

```
                                zeroRateBias,
                                outputModifiers );
```

Where `range` is an `agxSensor::TriaxialRange` that specifies the Gyroscope's measurement range, while `zeroRateBias` is an `agx::Vec3` specifying the measurement bias in each of the three axes at zero angular rate. `crossAxisSensitivity` is an `agxSensor::TriaxialCrossSensitivity` that specifies how much an angular rate applied to one axis bleeds over into the signal of another axis. The optional `outputModifiers` parameter supplied to the model can be used to define modifiers applied to the output signal on a per-model basis. These modifiers are allowed to be the same as the ones that can be added to the Output Handler:

- Linear Acceleration Effects
- Total Gaussian Noise
- Signal Scaling
- Gaussian Spectral Noise

32.5.2. Output Handler

The `agxSensor::GyroscopeOutputHandler` of a Gyroscope is used to specify the output signal of the Gyroscope, and which axes it should contain. The fields of `agxSensor::TriaxialOutput::Field` can be combined in any order to form an output signal containing the desired information. For example a single axis Gyroscope output can be added with the unique identifier of `1` to the Gyroscope as:

```
gyroscope->getOutputHandler()->add<agx::Real, agxSensor::TriaxialOutput::Z_VALUE_F64>( 1 );
```

The output data from the Gyroscope can then be accessed through the `agxSensor::GyroscopeOutputHandler`, in the **POST_STEP** of the `agxSDK::Simulation` time step, using the unique identifier assigned to that data:

```
auto view = gyroscope->getOutputHandler()->view<agx::Real>( 1 );
for ( const auto& v : view )
  // Use gyroscope measurement v.
```

32.5.2.1. Linear Acceleration Effects

The Gyroscope output may commonly be disturbed by accelerations applied to the Gyroscope. To achieve this behavior, an `agxSensor::GyroscopeLinearAccelerationEffects` instance can be added to the output handler of the Gyroscope:

```
agx::Vec3 perAxisAccelerationEffects = agx::Vec3( 2.62 * 1e-4 );

agxSensor::GyroscopeLinearAccelerationEffectsRef accelerationEffects =
  new agxSensor::GyroscopeLinearAccelerationEffects( perAxisAccelerationEffects );

accelerometer->getOutputHandler()->add( accelerationEffects );
```

The `agxSensor::GyroscopeLinearAccelerationEffects` allows the acceleration effects to be specified in general using an `agx::Matrix3x3` to model how accelerations along each input axis perturbs the readings at each output axis, or using an `agx::Vec3` to model only the more commonly specified case of linear acceleration effects along the measurement output axes of the Gyroscope.

32.5.2.2. Total Gaussian Noise

To model base level electrical noise in the measurement signal, an `agxSensor::TriaxialGaussianNoise` instance can be added to the `agxSensor::Gyroscope`'s output handler:

```
agxSensor::TriaxialGaussianNoiseRef noise =
  new agxSensor::TriaxialGaussianNoise( agx::Vec3( 5.67 * 1e-6 ) );

gyroscope->getOutputHandler()->add( noise );
```

The `agxSensor::TriaxialGaussianNoise` allows the noise RMS to be specified independently for each of the Gyroscope axes.

32.5.2.3. Signal Scaling

The Gyroscope output signal can be rescaled by adding an `agxSensor::TriaxialSignalScaling` instance to the output handler of the `agxSensor::Gyroscope`. This scaling can be used to conveniently convert from AGX Dynamics output in the SI unit of radians/s to the common Gyroscope reading unit of degrees/s:

```
agx::Vec3 perAxisScaling = agx::Vec3( TriaxialSignalScaling::CONVERT_RADIANS_PER_SECOND_TO_DEGREES_PER_SECOND );

agxSensor::TriaxialSignalScaingRef scaling =
  new agxSensor::TriaxialSignalScaling( perAxisScaling );

gyroscope->getOutputHandler()->add( scaling );
```

The `agxSensor::TriaxialSignalScaling` allows the signal scaling to be specified independently for each of the Gyroscope axes.

32.5.2.4. Gaussian Spectral Noise

Commonly, Gyroscopes come with a specified output noise spectral density, describing the expected noise at a given sample frequency. A Gaussian distributed noise using this density specification can be added to the output of an `agxSensor::Gyroscope` using an `agxSensor::TriaxialSpectralGaussianNoise`:

```
agxSensor::TriaxialSpectralGaussianNoiseRef noise =
  new agxSensor::TriaxialSpectralGaussianNoise( agx::Vec3( 1.75 * 1e-4 ) );

gyroscope->getOutputHandler()->add( noise );
```

The `agxSensor::TriaxialSpectralGaussianNoise` allows the noise density RMS to be specified independently for each of the Gyroscope axes.

32.6. Magnetometer

`agxSensor::Magnetometer` is a sensor type in the AGX Sensor module. It provides a standardized way to measure the magnetic field vector in a specified frame over time, with up to three degrees of freedom. In order to be active, the Magnetometer must be added to the `agxSensor::Environment`:

```
agxSensor::MagnetometerRef magnetometer = new agxSensor::Magnetometer( frame, model );
environment->add( magnetometer );
```

Where `environment` is an instance of an `agxSensor::Environment`, `frame` is an optional `agx::Frame` and `model` is an `agxSensor::MagnetometerModel`. The `frame` parameter can be set to, for example, attach the Magnetometer to anything moving in the simulation, such as a Rigid Body. Meanwhile, the `model` determines the behavior and characteristics of the Magnetometer.

Specifying which output an Magnetometer should produce, and later accessing the resulting signal, is done using the Magnetometer's `agxSensor::MagnetometerOutputHandler`, see Output Handler for details.

In accordance with the SI unit usage of the rest of AGX Dynamics, this Magnetometer provides its measurements in units of tesla as default, instead of gauss. This can be adjusted by rescaling the signal using an `agxSensor::TriaxialSignalScaling` instance added to the output handler, see Signal Scaling for details.

32.6.1. Magnetometer Model

An `agxSensor::MagnetometerModel` is what determines the basic characteristics of an `agxSensor::Magnetometer`:

```
agxSensor::MagnetometerModelRef model =
  new agxSensor::MagnetometerModel( range,
                                    crossAxisSensitivity,
                                    zeroFluxBias,
                                    outputModifiers );
```

Where `range` is an `agxSensor::TriaxialRange` that specifies the Magnetometer's measurement range, while `zeroFluxBias` is an `agx::Vec3` specifying the measurement bias in each of the three axes at zero externally applied magnetic field. `crossAxisSensitivity` is an `agxSensor::TriaxialCrossSensitivity` that specifies how much a magnetic field applied to one axis bleeds over into the signal of another axis. The optional `outputModifiers` parameter supplied to the model can be used to define modifiers applied to the output signal on a per-model basis. These modifiers are allowed to be the same as the ones that can be added to the Output Handler:

- Total Gaussian Noise
- Signal Scaling
- Gaussian Spectral Noise

32.6.2. Output Handler

The agxSensor::MagnetometerOutputHandler of an Magnetometer is used to specify the output signal of the Magnetometer, and which axes it should contain. The fields of agxSensor::TriaxialOutput::Field can be combined in any order to form an output signal containing the desired information. For example a single axis Magnetometer output can be added with the unique identifier of 1 to the Magnetometer as:

```
magnetometer->getOutputHandler()->add<agx::Real, agxSensor::TriaxialOutput::Z_VALUE_F64>( 1 );
```

The output data from the Magnetometer can then be accessed through the agxSensor::MagnetometerOutputHandler, in the **POST_STEP** of the agxSDK::Simulation time step, using the unique identifier assigned to that data:

```
auto view = magnetometer->getOutputHandler()->view<agx::Real>( 1 );
for ( const auto& v : view )
  // Use magnetometer measurement v.
```

32.6.2.1. Total Gaussian Noise

To model base level electrical noise in the measurement signal, an agxSensor::TriaxialGaussianNoise instance can be added to the agxSensor::Magnetometer's output handler:

```
agxSensor::TriaxialGaussianNoiseRef noise =
  new agxSensor::TriaxialGaussianNoise( agx::Vec3( 4.5e-7 ) );

magnetometer->getOutputHandler()->add( noise );
```

The agxSensor::TriaxialGaussianNoise allows the noise RMS to be specified independently for each of the Magnetometer axes.

32.6.2.2. Signal Scaling

The Magnetometer output signal can be rescaled by adding an agxSensor::TriaxialSignalScaling instance to the output handler of the agxSensor::Magnetometer. This scaling can be used to conveniently convert from AGX Dynamics output in the SI unit of tesla to the common Magnetometer reading unit of gauss:

```
agx::Vec3 perAxisScaling = agx::Vec3( TriaxialSignalScaling::CONVERT_TESLA_TO_GAUSS );

agxSensor::TriaxialSignalScaingRef scaling =
  new agxSensor::TriaxialSignalScaling( perAxisScaling );

magnetometer->getOutputHandler()->add( scaling );
```

The agxSensor::TriaxialSignalScaling allows the signal scaling to be specified independently for each of the Magnetometer axes.

32.6.2.3. Gaussian Spectral Noise

In some cases, Magnetometers come with a specified output noise spectral density, describing the expected noise at a given sample frequency. A Gaussian distributed noise using this density specification can be added to the output of an agxSensor::Magnetometer using an agxSensor::TriaxialSpectralGaussianNoise:

```
agxSensor::TriaxialSpectralGaussianNoiseRef noise =
  new agxSensor::TriaxialSpectralGaussianNoise( agx::Vec3( 4.2e-9 ) );

magnetometer->getOutputHandler()->add( noise );
```

The agxSensor::TriaxialSpectralGaussianNoise allows the noise density RMS to be specified independently for each of the Magnetometer axes.

32.7. Encoder

agxSensor::Encoder is a sensor type in the AGX Sensor module. It provides a standardized way to measure the position of an unconstrained degree of freedom of a constraint, acting both as a linear or rotary Encoder. In order to be active an Encoder must be added to the agxSensor::Environment:

```
agxSensor::EncoderRef encoder = new agxSensor::Encoder( hinge, model );
environment->add( encoder );
```

Where environment is an instance of an agxSensor::Environment, hinge is the constraint to sample measurements from, in this case an agx::Hinge, and model is an agxSensor::EncoderModel. The agxSensor::Encoder provides a number of constructors that allow the Encoder to

be attached to the following constraints:

- `agx::CylindricalJoint`: Linear or rotary Encoder, depending on the selected degree of freedom.
- `agx::Hinge`: Rotary Encoder.
- `agx::Prismatic`: Linear Encoder.
- `agx::SplineJoint`: Linear or rotary Encoder, depending on the selected degree of freedom.
- `agx::SlackCylindricalJoint`: Linear or rotary Encoder, depending on the selected degree of freedom.
- `agx::SlackHinge`: Rotary Encoder.
- `agx::SlackPrismatic`: Linear Encoder.
- `agxVehicle::WheelJoint`: Rotary wheel Encoder, or rotary steering Encoder, depending on the selected degree of freedom.

Meanwhile, the `model` determines the behavior and characteristics of the Encoder.

Specifying which output an Encoder should produce, and later accessing the resulting signal, is done using the Encoder's `agxSensor::EncoderOutputHandler`, see Output Handler for details. As of default, the Encoder will provide its readings as they would be provided after the microcontroller without resolution binning, and thus it does not provide the pure quadrature waveforms. To achieve a step-like resolution binning, an `agxSensor::MonoaxialSignalResolution` instance can be added to the output handler as can be seen under Signal Resolution, while a quadrature-like output can be achieved through combination of model parameters and resolution binning as described under Encoder Model.

In accordance with the SI unit usage of the rest of AGX Dynamics, this Encoder provides its measurements in units of radians or meter as default. This can be adjusted by rescaling the signal using an `agxSensor::MonoaxialSignalScaling` instance added to the output handler, see Signal Scaling for details.

32.7.1. Encoder Model

An `agxSensor::Encodermodel` is what determines the basic characteristics of an `agxSensor::Encoder`:

```
agxSensor::EncoderModelRef model =
  new agxSensor::EncoderModel( mode,
                               range,
                               outputModifiers );
```

Where `mode` is either `agxSensor::EncoderModel::INCREMENTAL` or `agxSensor::EncoderModel::ABSOLUTE` to distinguish between an incremental or absolute mode of operation and `range` denotes the range of one incremental bin or the full absolute range of the Encoder, depending on the selected mode of operation. The optional `outputModifiers` parameter supplied to the model can be used to define modifiers applied to the output signal on a per-model basis. These modifiers are allowed to be the same as the ones that can be added to the Output Handler:

- Total Gaussian Noise
- Signal Resolution
- Signal Scaling

To model a 360-degree incremental rotary Encoder one would construct the model as such:

```
agxSensor::EncoderModelRef model =
  new agxSensor::EncoderModel( agxSensor::EncoderModel::INCREMENTAL,
                               agx::RangeReal( 0.0, 2.0 * agx::PI ) );
```

Whereas, for an absolute linear Encoder, an appropriate model might be:

```
agxSensor::EncoderModelRef model =
  new agxSensor::EncoderModel( agxSensor::EncoderModel::ABSOLUTE,
                               agx::RangeReal(
                                 -std::numeric_limits<agx::Real>::infinity(),
                                 std::numeric_limits<agx::Real>::infinity(),
                               ) );
```

To achieve a single quadrature-like waveform as the output a combination of an incremental range and resolution binning can be employed:

```
agx::Real ppr = 1000;
agx::Real period = 2.0 * agx::PI / ppr;
```

```
agxSensor::IMonoaxialSignalSystemNodeRefVector modifiers{
  new agxSensor::MonoaxialSignalResolution( 0.5 * period )
};

agxSensor::EncoderModelRef model =
  new agxSensor::EncoderModel( agxSensor::EncoderModel::INCREMENTAL,
                               agx::RangeReal( 0.0, period ),
                               modifiers );
```

32.7.2. Output Handler

The `agxSensor::EncoderOutputHandler` of an Encoder is used to specify the output signal of the Encoder. The fields of `agxSensor::EncoderOutput::Field` can be combined in any order to form an output signal containing the desired information. For example a regular Encoder position output can be added with the unique identifier of `1` to the Encoder as:

```
encoder->getOutputHandler()->add<agx::Real, agxSensor::EncoderOutput::POSITION_F64>( 1 );
```

The output data from the Encoder can then be accessed through the `agxSensor::EncoderOutputHandler`, in the **POST_STEP** of the `agxSDK::Simulation` time step, using the unique identifier assigned to that data:

```
auto view = encoder->getOutputHandler()->view<agx::Real>( 1 );
for ( const auto& v : view )
  // Use encoder measurement v.
```

32.7.2.1. Total Gaussian Noise

To model base level noise in the measurement signal, an `agxSensor::MonoaxialGaussianNoise` instance can be added to the `agxSensor::Encoder`'s output handler:

```
agxSensor::MonoaxialGaussianNoiseRef noise =
  new agxSensor::MonoaxialGaussianNoise( 4.5e-7 );

encoder->getOutputHandler()->add( noise );
```

32.7.2.2. Signal Resolution

Due to the quadrature waveforms that Encoders natively provides, the accumulated output signal after the microcontroller will usually appear step-like, with resolution bins. This effect can be a achieved by adding an `agxSensor::MonoaxialSignalResolution` instance to the `agxSensor::Encoder`'s output handler:

```
agx::Real ppr = 1000;
agx::Real period = 2.0 * agx::PI / ppr;

agxSensor::MonoaxialSignalResolutionRef resolution =
  new agxSensor::MonoaxialSignalResolution( 0.5 * period );

encoder->getOutputHandler()->add( resoluiton );
```

32.7.2.3. Signal Scaling

The Encoder output signal can be rescaled by adding an `agxSensor::MonoaxialSignalScaling` instance to the output handler of the `agxSensor::Encoder`. This scaling can for example be used to conveniently convert a rotary Encoder output from an AGX Dynamics output in the SI unit of radians to an output in degrees:

```
agxSensor::MonoaxialSignalScalingRef scaling =
  new agxSensor::MonoaxialSignalScaling( agx::RAD_TO_DEG );

encoder->getOutputHandler()->add( scaling );
```

32.8. Odometer

`agxSensor::Odometer` is a sensor type in the AGX Sensor module, representing a wheel rotation-based Odometer. It provides a standardized way to measure the distance traveled due to the rotation around an unconstrained rotational degree of freedom of a constraint. In order to be active an Odometer

must be added to the agxSensor::Environment:

```
agxSensor::OdometerRef odometer = new agxSensor::Odometer( hinge, model );
environment->add( odometer );
```

Where environment is an instance of an agxSensor::Environment, hinge is the constraint to sample measurements from, in this case an agx::Hinge, and model is an agxSensor::OdometerModel. The agxSensor::Odometer provides a number of constructors that allow the Odometer to be attached to the following constraints with unconstrained rotational degrees of freedom:

- agx::CylindricalJoint: Rotational degree of freedom.
- agx::Hinge: Rotational degree of freedom.
- agx::SplineJoint: Rotational degree of freedom.
- agx::SlackCylindricalJoint: Rotational degree of freedom.
- agx::SlackHinge: Rotational degree of freedom.
- agxVehicle::WheelJoint: Wheel's rotational degree of freedom.

Meanwhile, the model determines the behavior and characteristics of the Odometer.

Specifying which output an Odometer should produce, and later accessing the resulting signal, is done using the Odometer's agxSensor::OdometerOutputHandler, see Output Handler for details.

32.8.1. Odometer Model

An agxSensor::Odometermodel is what determines the basic characteristics of an agxSensor::Odometer:

```
agxSensor::OdometerModelRef model =
  new agxSensor::OdometerModel( wheelRadius
                                outputModifiers );
```

Where wheelRadius is the wheel radius assumed by the odometer when calculating the distance traveled from rotation. The optional outputModifiers parameter supplied to the model can be used to define modifiers applied to the output signal on a per-model basis. These modifiers are allowed to be the same as the ones that can be added to the Output Handler:

- Total Gaussian Noise
- Signal Resolution
- Signal Scaling

32.8.2. Output Handler

The agxSensor::OdometerOutputHandler of an Odometer is used to specify the output signal of the Odometer. The fields of agxSensor::OdometerOutput::Field can be combined in any order to form an output signal containing the desired information. For example a regular Odometer distance output can be added with the unique identifier of 1 to the Odometer as:

```
odometer->getOutputHandler()->add<agx::Real, agxSensor::OdometerOutput::DISTANCE_F64>( 1 );
```

The output data from the Odometer can then be accessed through the agxSensor::OdometerOutputHandler, in the **POST_STEP** of the agxSDK::Simulation time step, using the unique identifier assigned to that data:

```
auto view = odometer->getOutputHandler()->view<agx::Real>( 1 );
for ( const auto& v : view )
  // Use odometer measurement v.
```

32.8.2.1. Total Gaussian Noise

To model base level noise in the measurement signal, an agxSensor::MonoaxialGaussianNoise instance can be added to the agxSensor::Odometer's output handler:

```
agxSensor::MonoaxialGaussianNoiseRef noise =
  new agxSensor::MonoaxialGaussianNoise( 4.5e-7 );
```

```
odometer->getOutputHandler()->add( noise );
```

32.8.2.2. Signal Resolution

Odometers may be based on underlying wheel Encoders, natively providing quadrature signals. Due to these quadrature signals that the underlying wheel Encoders provide, the accumulated output signal after the microcontroller and Odometer accumulation may appear step-like, with resolution bins. This effect can be a achieved by adding an `agxSensor::MonoaxialSignalResolution` instance to the `agxSensor::Odometer`'s output handler:

```
agx::Real ppr = 1000;
agx::Real period = 2.0 * agx::PI / ppr * odometer->getModel()->getWheelRadius();

agxSensor::MonoaxialSignalResolutionRef resolution =
  new agxSensor::MonoaxialSignalResolution( 0.5 * period );

odometer->getOutputHandler()->add( resoluiton );
```

32.8.2.3. Signal Scaling

The Odometer output signal can be rescaled by adding an `agxSensor::MonoaxialSignalScaling` instance to the output handler of the `agxSensor::Odometer`. This scaling can for example be used to conveniently convert an Odometer output from an AGX Dynamics output in the SI unit of meter to an output in kilometer:

```
agxSensor::MonoaxialSignalScalingRef scaling =
  new agxSensor::MonoaxialSignalScaling( 1.0e-3 );

odometer->getOutputHandler()->add( scaling );
```

32.9. Known Limitations

- Lidar simulation is currently only supported on computers with a CUDA enabled graphics card with late Kepler or newer architectures.
- Sensor environment materials can currently not be configured on systems without support for Lidar simulation.

33. Surface Velocity Conveyor Belt

The class *agx::SurfaceVelocityConveyorBelt* can be used in order to simulate a basic conveyor belt. No internal mechanics in the conveyor belt are simulated, only velocity changes in contacts between conveyor belt and transported goods. As the name indicates, the SurfaceVelocityConveyorBelt uses the surface velocity property inherent in contact points, which is described in chapter 11.7.6. For simple cases, if might suffice to use an *agxCollide::Geometry*'s surface velocity. However, the latter is constant over the whole surface of the Geometry. For cases like in a conveyor belt, one wants the velocity to follow around the surface of the geometry which might be more complex (going up and down, …).

A SurfaceVelocityConveyorBelt has different surface velocity directions for different parts of the belt, but the length - or rather "speed" - remains the same. The different directions are computed from a list of points along the surface of the belt in direction of its movement - the single directions are computed from one point to the next. For a given contact point, the closest line segment from one conveyor belt point to the next is chosen for the surface velocity. This might give wrong behavior in more advanced concave cases, which is a limitation of the model. For "typical" conveyor belt cases however, it should work fine.

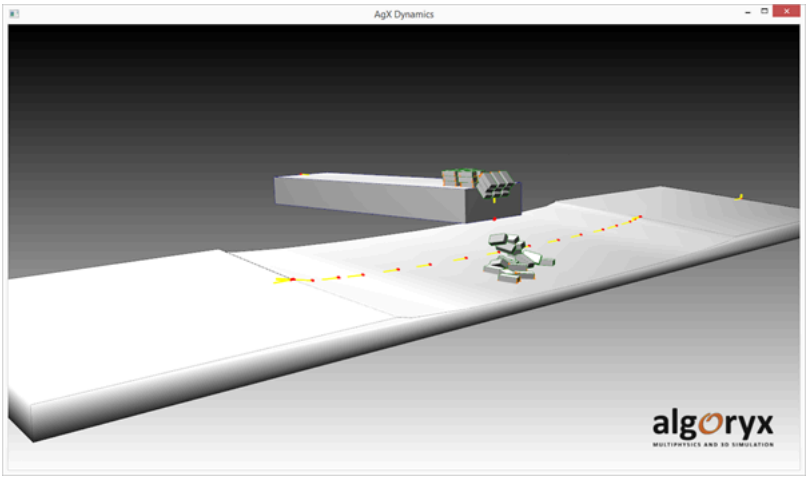


Fig. 33.1 Two conveyor belts. Ret dots are points, yellow lines indicate direction.

Implementationwise, the *agx::SurfaceVelocityConveyorBelt* inherits from *agxCollide::Geometry* and can be used as a geometry. It is constructed from a list of points, such as in the following example:

```
#include <agx/SurfaceVelocityConveyorBelt.h>

void createConveyorBelt(agxSDK::Simulation* sim)
{
  agx::Vec3Vector points;
  points.push_back(agx::Vec3(-1, 0, 0));
  points.push_back(agx::Vec3(0, 0, 1));
  points.push_back(agx::Vec3(1, 0, 0)); // Change in direction, from upwards to downwards.
  agx::SurfaceVelocityConveyorBeltRef belt = new agx::SurfaceVelocityConveyorBelt(points);
  sim->add(belt);
  // We should also add shapes to the conveyor belt, and they should map the points.
}
```

Given n points, the surface velocity conveyor belt will create n-1 directions. This allows for open connection. If a closed loop is desired, then the first point should be added another time as the last one, as in this example:

```
points.push_back(a);
points.push_back(b);
points.push_back(c);
points.push_back(d);
points.push_back(a); // Close the rectangle loop.
```

The speed of a conveyor belt can be set by a call to `void setSpeed(agx::Real speed)`, where negative speeds will indicate a change in direction.

Debug rendering can be activated for a conveyor belt by a call to `bool enableDebugRendering(agxSDK::Simulation* sim)` (and disabled with a call to `bool disableDebugRendering()`). It will look like the red dots and yellow lines in Fig. 33.1.

When creating an agx::SurfaceVelocityConveyorBelt for an agxCollide::Trimesh, one can cut the trimesh with a plane in order to obtain an ordered list of points. This list of points can be used as input to the constructor for the conveyor belt, instead of having to specify them all by hand. However, to close the loop, the points' first point should be added once more as its last, as in this example:

```
#include <agxCollide/Trimesh.h>
#include <agx/SurfaceVelocityConveyorBelt.h>

agx::SurfaceVelocityConveyorBelt* createConveyorBeltFromTrimesh(agxCollide::TrimeshRef trimesh)
{
  auto plane = agx::Plane(agx::Vec3(0, 1, 0), 0);
  // This will increase and decrease the trimesh's ref count - make sure you have a ref pointer to it.
  auto points = agxCollide::intersectTrimeshWithPlane(trimesh, plane, agx::AffineMatrix4x4::translate(trimesh-
>getCenter()));
  if (points.size() == 0)
    return nullptr;
  points.push_back(points.front()); // Close the loop.
  auto belt = new agx::SurfaceVelocityConveyorBelt(points);
  belt->add(trimesh);
  return belt;
}
```

34. AGX Control

The AGX Control module contains functionality related to controlling vehicles and machines. Please note that some of the functionality is contained in other C++ namespaces than `agxControl::`.

34.1. Inverse Kinematics and Inverse Dynamics

Functionality for inverse kinematics and inverse dynamics and some applications such as Computed-Torque-Control are described in the Robotics chapter. This lies in the agxModel namespace but is part of the agxControl module.

34.2. Pure Pursuit

Pure Pursuit is a geometric path tracking algorithm. Given a vehicle of some type, steering information is computed so that the vehicle should reach a target position on a path. As the vehicle moves forward, so does the target position.

The algorithm works in the following way:

- Given the vehicle's position relative to the path and a lookahead distance, find the target position further along the path
- Based on curvature, compute how the vehicle should steer to land on the target position.

How well the vehicle tracks the actual path depends on the path, is it smooth or have sharp turns? The choice of lookahead value also affects the tracking. This lookahead can be a fixed value or be velocity-dependent for the vehicle.

Different kinds of vehicles have different ways to turn. The Pure Pursuit implementation is made in a generic way so that it takes two inputs which provide abstractions for the required functionality: `agxControl::PurePursuit(agxControl::PurePursuit::SteeringInformation*, agxControl::PurePursuit::Path*);`.

The first argument is an abstraction for the vehicle type. A specialization here can, for example, be `agxControl::TrackedVehicleSteeringInformation` or `agxControl::CarSteeringInformation`. For a tracked vehicle with 2 tracks, two velocities are computed, and the difference between left- and right track speed causes the machine to turn. For a car, one value is computed which is the steering angle.

The second argument to `PurePursuit` is the path representation which has an API for computing the target position. A specialization here is the `agxControl::LinearSegmentPath` which consists of multiple linear segments.

Note

Pure Pursuit is one possible component for vehicle motion that handles path tracking. It does not control start/stop logic or vehicle speed. For that it has to be combined with additional components or logic.

An example of how this functionality can be used can be seen in `python/tutorials/tutorial_purepursuit.agxPy` which also shows how to create and provide the vehicle control class with the correct coordinate system.

35. Robotics

AGX dynamics offers flexibility in simulating robotics for various tasks and seamless integration with other software. Robots based on CAD drawings or URDF can be imported and configured in `Python` or `C++`. Once the model is in place, AGX supports motion planning, testing, developing, and training controllers through reinforcement learning. The controller can live directly in AGX or elsewhere and communicate with the simulation using middleware such as `ROS2`.

35.1. Serial chains

Although AGX dynamics supports modelling all types of robot mechanisms, the most convenient way to do conventional robotics is to use a `agxModel::SerialKinematicChain`. It offers functionality for planning and control, including forward, inverse, and differential *kinematics*, as well as forward and inverse *dynamics*.

A kinematic chain is a description of a mechanical system. The chain links are rigid bodies that are constrained to each other. Depending on its topology, a chain is classified as either serial, tree, or graph structure. Serial chains and trees are open without loops, whereas the graph can be closed. A serial kinematic chain describes many mechanisms, such as industrial robots, excavator arms, and cranes.

The class `agxModel::SerialKinematicChain` in AGX represents an open serial chain. There are two ways to initialize this class. The first is to provide a simulation along with the base and end bodies, in which case the implementation searches through the simulation breadth first among the bodies and constraints. The other way to create a chain is to provide a list of bodies and a list of constraints. In this case, the items' order in both lists is crucial to create a valid chain as it requires that `constraint[i]` connects `body[i]` and `body[i+1]`.

35.1.1. Forward kinematics

Forward kinematics is concerned with finding the position and orientation of the end-effector, given the robot's joint configuration. In the case of serial chains, joint configurations respecting limits result in valid end-effector poses in the task space.

AGX dynamics supports computing forward kinematics *relative to the base* from a `agxModel::SerialKinematicChain`. The method `computeForwardKinematics` computes an `AffineMatrix4x4` pose of the end-effector given a joint configuration, including the option to consider joint limits. Similarly, `computeForwardKinematicsAll` computes the pose of all links in a chain.

Examples of forward computations can be seen in `data/python/tutorials/RobotControl/tutorial_set_robot_pose.agxPy`.

35.1.2. Manipulator Jacobian

The manipulator Jacobian $J(q)$ is a component of differential kinematics and essential for controlling and planning the motion of robotic manipulators. It relates the joint velocities $\dot{q}$ with the velocity of the end-effector v according to

$$v = J(q)\dot{q}\,.$$

The method `computeManipulatorJacobian` evaluates the manipulator Jacobian in fixed frame coordinates relative to the base. It takes the joint positions q as input and outputs the Jacobian matrix stored *by row* in a vector with size $6 \times N_j$.

35.1.3. Inverse kinematics

Inverse kinematics (IK) deals with finding joint configurations that correspond to the desired position and orientation of the end-effector in task space. Unlike forward kinematics, the inverse problem can have zero, multiple, or infinitely many solutions. If the desired pose is unreachable by the manipulator, IK has zero solutions. When a robot has more joints than the dimension of the task space, it is called redundant. While multiple solutions may exist for nonredundant manipulators, e.g. in the case of elbow up or elbow down, only redundant robots can have infinitely many solutions.

A general approach to solve the IK problem for open chains is to rely on iterative numerical algorithms. These take advantage of the inverse of the manipulator Jacobian to find a solution when one exists.

The `agxModel:SerialKinematicChain` can compute IK via the method `computeInverseKinematics`. The input is a desired end-effector transform in task space relative to the base, current joint positions to help guide the algorithm toward a solution, and an optional solver tolerance.

We recommend checking the IK computations return status (enum) before using the output joint angles. Also, the best practice is to use the latest joint positions as an initial guess. This is to avoid cases where a slight change in end-effector transform causes an element in the output joint position to switch from π to $-\pi$.

35.1.4. Inverse dynamics

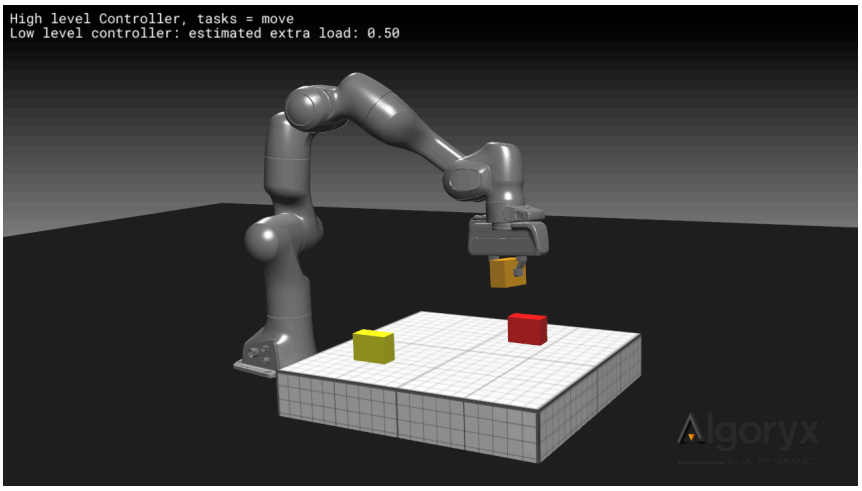
Dynamics is the study of motions and the forces and torques that cause them. AGX typically follows this process: Advancing the simulation based on current forces and torques acting on bodies and particles, resulting in a new state with updated positions and velocities.

The problem of inverse dynamics (ID) is to find the forces/torques that result in a certain state. This is the opposite of forward dynamics, where the forces and torques are known, but the new state is not.

Unlike kinematics computations, inverse dynamics computations in the `SerialKinematicChain` require knowledge of the simulation. Therefore, the chain must receive a simulation instance either through the constructor or by invoking `setSimulation` at a later stage.

With knowledge of the simulation, `computeInverseDynamics` performs the ID computations with an input corresponding to the desired joint accelerations, $\ddot{q}_{\text{desired}}$. The rest of the state, current q and $\dot{q}$ are available via the simulation. The computed output is a vector with forces/torques τ. One overload to `computeInverseDynamics` also accepts additional external forces acting on the chain's links.

Examples with ID including external forces are available both in python `data/python/tutorials/RobotControl/tutorial_pick_and_place.agxPy` and C++ `tutorials/agxOSG/tutorial_robot_pick_and_place.cpp`.



Ensure thread safety when using `computeInverseDynamics`. The method reads data from the chain's components and transforms constraint attachment frames. Therefore, calling this method from a separate thread while the solver runs in another may cause incorrect results or undefined behaviour. Instead, recommended use is between calls to `Simulation::stepForward` or within a `StepEventListener` callback method, such as `pre` or `preCollide`.

35.2. Robot control

How to control a robot to achieve the desired behaviour depends on the working environment and the task tries to solve. AGX Dynamics aims to provide functionality that makes it easy to test, develop, and train controllers in dynamic simulation environments for various applications.

35.2.1. PID control

The most basic controller is the Proportional-Integral-Derivative (PID) controller. As a PID lacks a dynamic model of the robot, control relies entirely on feedback. The following mathematical expression describes it:

$$u(t) = MV(t) = K_p\, e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de}{dt},$$

where u is the output signal, also known as the Manipulated Variable (MV), e is the error between the measured process variable and the set point, and K_p, K_i, and K_d are the tuning parameters, or proportional gain, integral gain, and derivative gain.

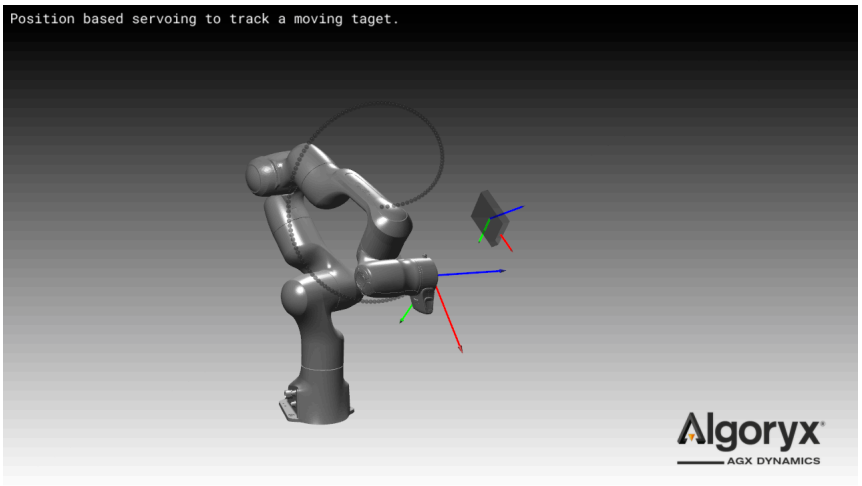
A PID controller can be used on its own or as part of a controller. For more details about the PID implementation in AGX and examples of PID control, see AGX Model: PID Controller.

35.2.2. Velocity control

Although most robots use actuators with force or torque inputs, these may be set to velocity mode for control. Velocity mode means that the velocities of the joints are controlled directly.

In AGX dynamics, joints are modelled as constraints with a single degree of freedom, i.e. either as a `agx::Hinge` or `agx::Prismatic`. These constraints can activate a `agx::Motor1D`, which allows setting a desired joint velocity through the `setSpeed` method.

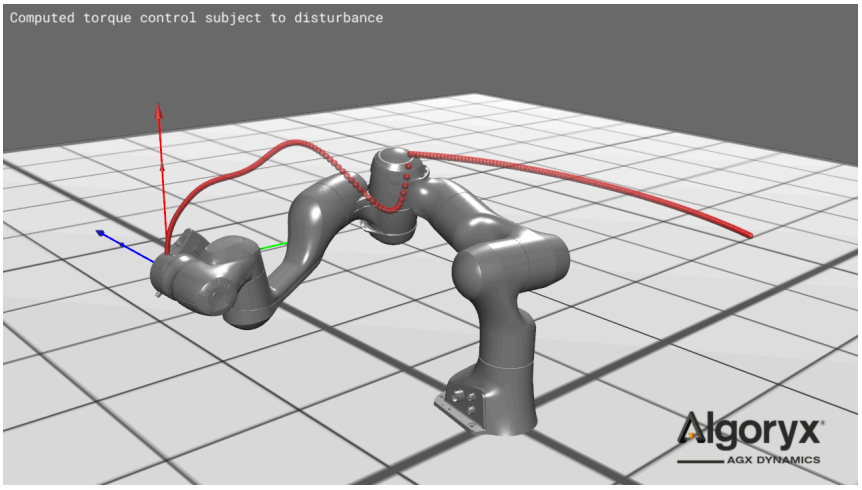
The tutorial `data/python/tutorials/RobotControl/tutorial_differential_kinematics.agxPy` demonstrates velocity control of the Franka Emika Panda when the task is to track a moving object. In the tutorial, we implement a control method called *Position-based servoing*, which relies on the pseudoinverse of the manipulator Jacobian to compute joint velocities.



35.2.3. Torque control

In contrast to calculating torques using a PID controller, precise torque control requires a dynamics model of the robot. When a dynamics model is at hand, inverse dynamics calculates the torques that achieve a desired motion. For instance, the tutorial `data/python/tutorials/RobotControl/tutorial_gravity_compensation.agxPy` demonstrates how to counteract gravity and maintain zero acceleration.

However, as no model is perfect, most practical controllers rely on feedback. A motion control algorithm that combines feed-forward with feedback is *computed torque control* as demonstrated in `data/python/tutorials/RobotControl/tutorial_computed_torque_control.agxPy`.



Regardless of where the desired torques come from, a simple way to torque-control a robot in AGX dynamics is to use a `agx::Motor1D` on each joint. An `agx::Motor1D` achieves the desired torque `tau` by setting the same value to the upper and lower limit through `setForceRange(tau, tau)`. For more accurate modelling of the actuator dynamics, refer to AGX DriveTrain. However, `agx::DriveTrain` is currently not supported by `computeInverseDynamics` but is expected to be so in the near future.

35.3. ROS2

The Robot Operating System (ROS) is a middleware and framework that is often used to work with robotics. It is not an Operating System even if the name implies so. One fundamental part in ROS2 is message-passing between different nodes, for example a controller that could be sending and receiving messages for a real or virtual robot.

ROS2 support in AGX Dynamics comes in two levels:

1. AGX Dynamics built in ROS2 support (see Built in ROS2 support for details).
    - Always available in AGX Dynamics on all supported operating systems except Ubuntu 18.04.
    - Publisher / Subscriber message passing support.
    - Supports all message types in `std_msgs`, `sensor_msgs`, `builtin_interfaces`, `rosgraph_msgs`, `geometry_msgs` and `agx_msgs`.

- Mechanism for serialization / deserialization of custom types (see Using custom types for details).
- QOS and domain ID support.
- No user-defined ROS2 package compilation support.

2. Pre-built ROS2 installation (see Pre-built ROS2 installation for details).

- Available for Windows only.
- Can be used together with AGX Dynamics in Python environments.
- User defined ROS2 packages can be compiled and used.
- Several built in ROS2 features that come with a typical ROS2 installation.

35.3.1. Built in ROS2 support

With AGX Dynamics it is possible to do Publisher / Subscribe message passing. It does not require a separate ROS2 installation or build in order to work.

All messages types in the following packages are supported:

`std_msgs`, `sensor_msgs`, `builtin_interfaces`, `rosgraph_msgs`, `geometry_msgs` and `agx_msgs`.

AGX Dynamics offers an API through the agxROS2 namespace that wraps some of the ROS2 functionality, namely Publisher and Subscriber message passing.

Specifying `QOS` such as reliability, durability, history type and depth as well as domain ID is also supported.

It targets the version `ROS2 Humble` and specifically the DDS implementation `Fast-DDS` which is the default DDS implementation in `ROS2 Humble`. ROS2 generally does not guarantee compatibility between ROS2 versions and DDS vendors, but it should work in practice, especially when matching the DDS vendor types. For more details regarding specifying the DDS vendor in a ROS2 environment, see Troubleshooting.

The built in ROS2 support does not support compiling and using user defined message types, but there is a mechanism available to enable sending and receiving custom data types, see Using custom types for details.

Below are two minimal examples of how to create a Publisher and Subscriber through the built in ROS2 support and passing a message between them. Note that the Publisher and Subscriber code normally are run on two separate applications and/or computers.

In C++:

```
#include <agxROS2/Publisher.h>
#include <agxROS2/Subscriber.h>
#include <iostream>

int main()
{
  // Create Publisher and Subscriber with specific Topic.
  agxROS2::stdMsgs::PublisherFloat32 pub("my_topic");
  agxROS2::stdMsgs::SubscriberFloat32 sub("my_topic");

  // Create a message and send it.
  agxROS2::stdMsgs::Float32 msgSend;
  msgSend.data = 3.141592f;
  pub.sendMessage(msgSend);

  // Try to receive the message.
  agxROS2::stdMsgs::Float32 msgReceive;
  if (sub.receiveMessage(msgReceive))
    std::cout << "Received message: " << msgReceive.data << ".\n";
  else
    std::cout << "Did not receive a message.\n";

  return 0;
}
```

In Python:

```
import agxROS2

def main():
    # Create Publisher and Subscriber with specific Topic.
```

```
    pub = agxROS2.PublisherStdMsgsFloat32("my_topic")
    sub = agxROS2.SubscriberStdMsgsFloat32("my_topic")

    # Create a message and send it.
    msgSend = agxROS2.StdMsgsFloat32()
    msgSend.data = 3.141592
    pub.sendMessage(msgSend)

    # Try to receive the message.
    msgReceive = agxROS2.StdMsgsFloat32()
    if sub.receiveMessage(msgReceive):
        print("Received message: {}".format(msgReceive.data))
    else:
        print("Did not receive a message.")

if __name__ == '__main__':
    main()
```

More advanced usage such as using a specific QOS or custom types can be seen in the tutorial `tutorials/agxOSG/tutorial_ros2.cpp`.

35.3.1.1. ros2_control interface

ros2_control is a commonly used framework to control robots using ROS2. You can find the documentation to ros2_control (here). We provide an easy way to communicate with ros2_control controllers using class `ROS2ControlInterface` which inherits from `agxSDK::StepEventListener`. It assumes that the joint commands and states is communicated on ROS2 topics using the `sensor_msgs/msg/JointState` message type. And that the robot loaded into ros2_control uses the (topic_based_ros2_control) hardware interface configured to use the same topics.

```
// Read your robot urdf file
agxSDK::AssemblyRef model = agxModel::UrdfReader::read(urdfFile, urdfPackage, initialPositions.get(), settings);

// Create the ROS2ControlInterface StepEventListener with set ROS2 topics for commands and states
auto ros2ControlInterface = agxROS2::ROS2ControlInterface("joint_commands", "joint_states");

// Add the 1DOF constraints from the robot you want to control and how you want to control it
ros2ControlInterface.addJoint(model->getConstraint1DOF("joint1"), agxROS2::ROS2ControlInterface::POSITION)
ros2ControlInterface.addJoint(model->getConstraint1DOF("joint2"), agxROS2::ROS2ControlInterface::POSITION)

// Add the listener to the simulation
simulation->add(ros2ControlInterface)
```

Or in python with

```
# Create an excavator
  excavator = Excavator()

  excavator.setRotation(terrain.getRotation())
  excavator.setPosition(0, 0, 0)
  simulation().add(excavator)

  joint_names = [
      'ArticulatedArm_Prismatic',
      'ArmPrismatic',
      'StickPrismatic',
      'TiltPrismatic',
      'BucketPrismatic',
      'BladePrismatic1',
      'BladePrismatic2',
      'CabinHinge',
      'LeftSprocketHinge',
      'RightSprocketHinge',
  ]

  excavator_ros2_interface = agxROS2.ROS2ControlInterface(
      "joint_commands",
      "joint_states",
  )
```

```
    for name in joint_names:
        excavator_ros2_interface.addJoint(excavator.getConstraint1DOF(name), agxROS2.ROS2ControlInterface.VELOCITY)
```

35.3.1.2. PointCloud2 usage with agxSensor

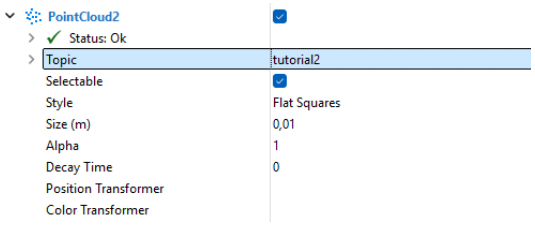
The `sensorMsgs/PointCloud2` message type (available using the built in support for ROS2 in AGX) defined in the `sensorMsgs` module for ROS2 provides a standardized type for point clouds but the output produced by the raytracing library used for our lidar solution (see Lidar) is not directly compatible with this message format. To simplify sending lidar output over ROS2 a utility function `agxROS2::convertLidarOutput` defined in `agxROS2/ROS2Util.h` is provided.

```
// Converts an output received from a lidar's output handler into a ROS2 message ready for publishing
agxROS2::sensorMsgs::PointCloud2 convertLidarOutput(agxSensor::RtOutput* lidarOutput, double timestamp, std::string
frameId, bool isDense, uint32_t height = 1)
```

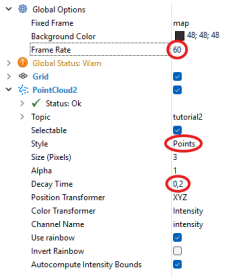
Tutorials in python and C++ are available which both provide minimal examples of both converting lidar output and using the built in support for ROS2 to publish the messages.

A common way of visualizing point clouds sent over ROS2 is via the visualization program `RViz` which comes with the most ROS2 installations, including the pre-built installation optionally provided for windows installations of AGX Dynamics (see Pre-built ROS2 installation for details). To visualize the published messages from scene 2 of the tutorial_pointcloud_over_ros2.agxPy tutorial file, the following steps can be used:

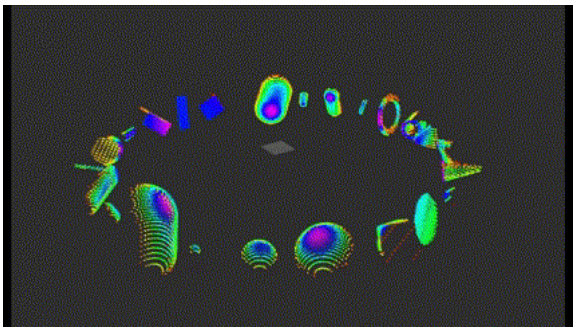
- Activate the ROS2 environment using `C:\opt\ros2-algoryx\install\setup.bat` in CMD
- Launch `RViz` using the command `rviz2`
- Press Ctrl+N and add a visualizer for the PointCloud2 message type
- Specify the topic of the visualizer as `tutorial2` to match the publisher in the tutorial



- Adjust visualization settings to make points easier to see (e.g. Frame Rate: 60, Style: Points, Decay time: 0,2)



- Now, if scene 2 of the tutorial is active and both programs are launched using the same ROS domain ID, the points produced by the lidar instance in agxViewer will be displayed in `RViz`.



35.3.1.3. Troubleshooting

If messages are missed, or the first couple of messages are missed, it may be of use to check the `QOS` settings used. By default the QOS settings are the same as in ROS2, i.e. RELIABILITY “reliable”, DURABILITY “volatile”, HISTORY “keep last” and history depth 10. More details on these settings can be seen (here).

If no communication between a built in agxROS2::Publisher and a built in agxROS2::Subscriber can be established it is possible that it is blocked by a firewall on the computer. Check the firewall settings and unblock the process in case it has been blocked.

If no communication between a built in agxROS2::Publisher/Subscriber and an external ROS2 node running on a complete ROS2 installation can be established, ensure that the same version of ROS2 is used. The built in ROS2 support in AGX Dynamics targets `ROS2 Humble`. If this does not solve the problem, ensure that the DDS implementation `Fast-DDS` is used in the ROS2 installation. This can be forced by setting the environment variable `RMW_IMPLEMENTATION=rmw_fastrtps_cpp`. See the (ROS2 documentation) for details. Also check the firewall settings if necessary.

If the creation of a `agxROS2::Publisher` or `agxROS2::Subscriber` hangs forever it is possible a seemingly rare bug in Fast-DDS is the culprit. This seems to happen when creating a DDS Participant (which is done automatically by AGX Dynamcis) on Windows on rare occations. The underlying reason why this can happen is not known, but a manual workaround is to do the following: 1. Turn off any process using ROS2. 2. Manually remove all files in `C:\ProgramData\eprosima\fastrtps_interprocess` (replace drive-letter if applicable).

35.3.1.4. Using custom types

There is some built in support for sending custom data types via ROS2. This is done using the `AnyMessageBuilder` and `AnyMessageParser`. The `AnyMessageBuilder` can serialize and deserialize several common built in primitive types to an `agxMsgs::Any` message. This message can then be sent on any Topic, received and deserialized using the `AnyMessageParser`.

The `agx_msgs::Any` message type as well as AnyMessageBuilder and AnyMessageParser are also available open-source on GitHub so that these can be used on machines that does not have AGX Dynamics or installed.

Below is a small example of how this can be used.

```
#include <agxROS2/AnyMessageBuilder.h>
#include <agxROS2/AnyMessageParser.h>
#include <agxROS2/Publisher.h>
#include <agxROS2/Subscriber.h>

struct MyStruct
{
  std::string myString;
  std::vector<int64_t> myInts;
  float myFloat{ 0.f };
};

int main()
{
  // Custom struct that we will send via ROS2.
  MyStruct sendStruct;
  sendStruct.myString = "We are all star stuff";
  sendStruct.myInts = { 42, 13, -999 };
  sendStruct.myFloat = 2.71828f;

  // Create Publisher and Subscriber with specific Topic.
  agxROS2::agxMsgs::PublisherAny pub("any_msg");
  agxROS2::agxMsgs::SubscriberAny sub("any_msg");

  // Serialize the custom struct type.
  agxROS2::AnyMessageBuilder builder;
  builder
    .beginMessage()
    .writeString(sendStruct.myString)
    .writeInt64Sequence(sendStruct.myInts)
    .writeFloat32(sendStruct.myFloat);

  // Send the message.
  pub.sendMessage(builder.getMessage());

  // Try to receive the message.
  agxROS2::agxMsgs::Any msgReceive;
  if (!sub.receiveMessage(msgReceive))
  {
    std::cout << "Dit not receive a message.\n";
```

```
    return 1;
  }

  // De-serialize the message and read back the custom struct that was received.
  MyStruct receiveStruct;
  agxROS2::AnyMessageParser parser;
  parser.beginParse();
  receiveStruct.myString = parser.readString(msgReceive);
  receiveStruct.myInts = parser.readInt64Sequence(msgReceive);
  receiveStruct.myFloat = parser.readFloat32(msgReceive);

  // receiveStruct now contains the data that was sent by the publisher.

  return 0;
}
```

Another example of this is available in the tutorial `tutorials/agxOSG/tutorial_ros2.cpp`.

35.3.2. Pre-built ROS2 installation

The pre-built ROS2 installation is available for Windows only. It can be installed through the AGX Dynamics installer. At the end of the AGX Dynamics installation, there is a checkbox to download and unzip a pre-built ROS2. It can also be downloaded manually from (here).

Un-zip it and place it in `C:\opt` such that the path to local_setup.bat is `C:\opt\ros2-algoryx\install\local_setup.bat`. The placement of the ROS2 installation must match exactly this.

The pre-built ROS2 installation is of version `ROS2 Humble`.

To be able to build custom ROS2 packages, `CMake` and `Visual Studio` must be installed manually on the computer.

There is no specific AGX Dynamics integration when using this alternative, but it is possible to use AGX Dynamics and a ROS2 installation in the same Python environment. Here, a simple ROS2 Publisher class and a ROS2 Subscriber class is added to an AGX Python script.

```
# Include the necessary ROS2 modules.
import rclpy
from rclpy.node import Node
from std_msgs.msg import Float32

# ROS2 publisher class.
class RosPublisher(Node):
  def __init__(self, node_name, topic_name):
    super().__init__(node_name)

    self.publisher = self.create_publisher(Float32, topic_name, 1)

  # Publish a ROS2 message.
  def send_data(self, data):
    msg = Float32()
    msg.data = data
    self.publisher.publish(msg)

# ROS2 subscriber class.
class RosSubscriber(Node):
  def __init__(self, node_name):
    super().__init__(node_name)

  # Note: the callback function must have an input parameter for the message.
  def register_callback(self, topic, callback_function):
    self.create_subscription(
      Float32,
      topic,
      callback_function,
      1)
```

35.4. URDF

AGX Dynamics provides a URDF reader that takes a URDF file and returns an AGX Dynamics representation of the model.

35.4.1. UrdfReader class

The `agxModel::UrdfReader` class has a static `read` method for reading urdf files and returning an `agxSDK::Assembly` object with all the robot parts (bodies, geometries and constraints). The behavior of the read method can be controlled using the `agxModel::UrdfReader::Settings` class. See the examples below for more information.

35.4.2. Examples

Below are two simple examples of how to read a URDF model and access one of its joints, both in C++ and Python. Also, instructions for running the `URDF Panda ROS2` Python example is described below.

35.4.2.1. C++ example

Read a URDF model and access one of its joints.

```
// Absolute path to the URDF file.
agx::String urdfFile = "absolute/path/model_package/description/model.urdf";

// (Optional) Absolute path to the URDF package path pointing to the 'package://'
// part of any filename attribute in the URDF file if existing.
agx::String urdfPackagePath = "absolute/path/model_package";

// (Optional) Initial joint positions. Must match in length the total number of
// revolute, continous and prismatic joints in the model.
agx::RealVector initJointPos {agx::PI/2.0, -agx::PI/4.0, 0.0, 0.0, 0.0, 0.0};

agxModel::UrdfReader::Settings settings;

// Determines if the base link of the URDF model should be attached to the
// world or not.
// Default is false.
settings.fixToWorld = true;

// If true the collision between linked/constrained bodies will be disabled.
// Default is false.
settings.disableLinkedBodies = true;

// If true, any link missing the <inertial></inertial> element will recieve
// a negnegligible mass and will be treated as a kinematic link
// and will be merged with its parent. According to http://wiki.ros.org/urdf/XML/link
// If false, the body will get its mass properties calculated from the shape/volume/density.
// Default is True.
settings.mergeKinematicLinks = true;

// Read the URDF.
agxSDK::AssemblyRef model =
  agxModel::UrdfReader::read(urdfFile,
                             urdfPackagePath,
                             &initJointPos,
                             settings);

// Add the model to the agxSDK::Simulation.
simulation->add(model);

// Access one of the joints in the model and set wanted speed for it. The name of the constraint
// corresponds to the name of the joint in the URDF file.
agx::Constraint1DOF* joint1 = model->getConstraint1DOF("joint_name");
joint1->getMotor1D()->setEnable(true);
joint1->getMotor1D()->setSpeed(0.5);
```

35.4.2.2. Python example

Read a URDF model and access one of its joints.

```
# Absolute path to the URDF file.
urdfFile = "absolute/path/model_package/description/model.urdf"

# (Optional) Absolute path to the URDF package path pointing to the 'package://'
```

```
# part of any filename attribute in the URDF file if existing.
urdfPackagePath = "absolute/path/model_package"

# (Optional) Initial joint positions. Must match in length the total number of
# revolute, continous and prismatic joints in the model.
initJointPos = agx.RealVector()
initJointPos.append(agx.PI/2.0)
initJointPos.append(-agx.PI/4.0)
for i in range(0, 4):
  initJointPos.append(0.0)

# Determines if the base link of the URDF model should be fixed to the
# world or not. Default is false.
fixToWorld = True

# If true the collision between linked/constrained bodies will be disabled.
# Default is false.
disableLinkedBodies = True

# If true, any link missing the <inertial></inertial> key will be treated
# as a kinematic link and will be merged with its parent. According to http://wiki.ros.org/urdf/XML/link
# If false, the body will get its mass properties calculated from the shape/volume/density.
# Default is True.
mergeKinematicLink = True

settings = agxModel.UrdfReaderSettings(fixToWorld, disableLinkedBodies, mergeKinematicLink)

# Read the URDF.
model = agxModel.UrdfReader.read(urdfFile, urdfPackagePath, initJointPos, settings)

# Add the model to the agxSDK.Simulation. Notice the .get() which is needed since model
# is an agxSDK.AssemblyRef.
sim.add(model.get())

# Access one of the joints in the model and set wanted speed for it. The name of the constraint
# corresponds to the name of the joint in the URDF file.
joint1 = model.getConstraint1DOF("joint_name")
joint1.getMotor1D().setEnable(True)
joint1.getMotor1D().setSpeed(0.5)
```

35.4.2.3. ROS2 URDF Panda example

Start the URDF Panda robot simulation

Simply double-click the `urdf_panda_ros2.agxPy` located in Algoryx/AGX-version/data/python/ROS2.

Start the URDF Panda ROS2 controller

1. Open a new “AGX Dynamics Command Line”.
2. In the AGX command prompt, cd to the AGX ROS2 examples directory, e.g:

```
cd Algoryx/AGX-version/data/python/ROS2
```

3. Run the URDF Panda ROS2 controller script by calling python, i.e:

```
python urdf_panda_ros2_controller.py
```

35.4.3. Supported features overview

Below is an overview of the URDF features. Those marked with ‘✓’ are supported by the URDF reader and those marked with ‘X’ are not.

```
robot ✓
|   name ✓
|
└──link ✓
|   |   name ✓
```

```
|   |
|   └──inertial ✓
|   |   |
|   |   └──origin ✓
|   |   |      xyz ✓
|   |   |      rpy ✓
|   |   |
|   |   └──mass ✓
|   |   |      value ✓
|   |   |
|   |   └──intertia ✓
|   |          ixx - izz ✓
|   |
|   └──visual ✓
|   |   |   name ✓
|   |   |
|   |   └──origin ✓
|   |   |      xyz ✓
|   |   |      rpy ✓
|   |   |
|   |   └──geometry ✓
|   |   |   |
|   |   |   └──box ✓
|   |   |   |      size ✓
|   |   |   |
|   |   |   └──cylinder ✓
|   |   |   |      radius ✓
|   |   |   |      length ✓
|   |   |   |
|   |   |   └──sphere ✓
|   |   |   |      radius ✓
|   |   |   |
|   |   |   └──mesh ✓
|   |   |          filename ✓
|   |   |          scale ✓
|   |   |
|   |   └──material ✓
|   |       |   name ✓
|   |       |
|   |       └──color ✓
|   |       |      rgba ✓
|   |       |
|   |       └──texture ✓
|   |
|   └──collision ✓
|       |   name ✓
|       |
|       └──origin ✓
|       |      xyz ✓
|       |      rpy ✓
|       |
|       └──geometry ✓
|              (see geometry under 'visual' above)
|
└──joint ✓
|   |   name ✓
|   |   type: revolute ✓
|   |   type: continuous ✓
|   |   type: prismatic ✓
|   |   type: fixed ✓
|   |   type: floating ✓
|   |   type: planar ✓
|   |
|   └──parent ✓
|   |      link ✓
|   |
|   └──child ✓
|   |      link ✓
|   |
```

```
|   └──axis ✓
|   |      xyz ✓
|   |
|   └──calibration X
|   |
|   └──dynamics ✓
|   |      damping X
|   |      friction ✓
|   |
|   └──limit ✓
|   |      lower ✓
|   |      upper ✓
|   |      effort ✓
|   |      velocity X
|   |
|   └──mimic ✓
|   |      multiplier ✓
|   |      offset ✓
|   |
|   └──safety_controller X
|
└──transmission X
|
└──gazebo X
|
└──sensor X
|
└──model_state X
```

35.4.4. AGX Dynamics representation

Below is a detailed description of the how the URDF features are represented by AGX Dynamics.

35.4.4.1. <robot>

The root element **robot** is represented by an `agxSDK::Assembly`. The name of the `agxSDK::Assembly` is set to the required *name* attribute.

35.4.4.2. <link>

Each **link** is represented by an `agx::RigidBody`. The name of the `agx::RigidBody` is set to the required *name* attribute.

35.4.4.2.1. <inertial>

The optional **inertial** element determines the mass and intertial properties of the `agx::RigidBody`. If no **inertial** element exist for a **link**, then AGX Dynamis will automatically calculate the mass properties based on all **visual** elements. The reason for using the visual elements is that it in most cases will have a more correct volumetric representation of the robot compared to the **collide** elements. The behaviour of the reader can be controlled using the `UrdfReader::Settings` class.

35.4.4.2.1.1. <origin>

The optional **origin** element determines the center of mass of the `agx::RigidBody`. If not specified (but **inertial** is), then identity position and orientation is used.

35.4.4.2.1.2. <mass>

The **mass** element determines the mass of the `agx::RigidBody`. This is a required attribute for any **inertial** element.

35.4.4.2.1.3. <inertia>

The **inertia** element determines the inertia of the `agx::RigidBody` with respect to its center of mass. This is a required attribute for any **inertial** element.

35.4.4.2.2. <visual>

Each **visual** element is represented by a `agxCollide::Geometry` with collision disabled. The name of the `agxCollide::Geometry` is set to the optional *name* attribute if it is specified.

35.4.4.2.2.1. <origin>

The optional **origin** element determines the position and orientation of the `agxCollide::Geometry` with respect to the owning `agx::RigidBody`. Identity position and orientation is used if not specified.

35.4.4.2.2.2. <geometry>

The required **geometry** element is represented by an `agxCollide::Shape` where **box**, **cylinder** and **sphere** are represented by the corresponding AGX Dynamics primitive shapes. Any **mesh** element is represented by a `agxCollide::Trimesh`.

35.4.4.2.2.3. <material>

The optional **material** element is represented by a `agxCollide::RenderMaterial`. Note that if a **texture** is specified, the texture information will be lost if a serialize/deserialize cycle of the scene is done and the underlying texture resouce (pointed to by the *filename* attribute) is moved or removed.

35.4.4.2.2.3. <collision>

Each **collision** element is handled similarly as a **visual** element but has collision enabled. Any `agxCollide::Geometry` representing a **collision** element is given empty `agxCollide::RenderData` meaning it will not be rendered to the screen if not debug rendering is active. Since it is not rendered, any **material** element inside a **collision** element is ignored. Note that a **collision** element does not affect the automatically calculated mass properties of a `agx::RigidBody` representing the parent **link** in the case where no **inertial** element exists. Only the **visual** elements are used in the calculation for that case.

35.4.4.2.4. <joint>

Each **joint** element is represented by an `agx::Constraint`. The name of the `agx::Constraint` is set to the required *name* attribute. Any *revolute* **joint** is represented by an `agx::Hinge` with the secondary constraint `agx::RangeController` active. Any *continuous* **joint** is represented by an `agx::Hinge`. Any *prismatic* **joint** is represented by an `agx::Prismatic` with the secondary constraint `agx::RangeController` active. Any *fixed* **joint** is represented by an `agx::LockJoint`. Any *floating* **joint** is ignored since free body movement is the default behaviour of any `agx::RigidBody`. Any *planar* **joint** is represented by an `agx::PlaneJoint`.

35.4.4.2.4.1. <origin>

The optional **origin** element specifies the position and orientation of the child `agx::RigidBody` representing the child **link** with respect to the parent `agx::RigidBody` representing the parent **link**. If not specified identity position and orientation is used.

35.4.4.2.4.2. <parent>

The required *link* attribute of the required **parent** element specifies the name of the parent `agx::RigidBody` that is attached to the `agx::Constraint` representing the **joint**.

35.4.4.2.4.3. <child>

The required *link* attribute of the required **child** element specifies the name of the child `agx::RigidBody` that is attached to the `agx::Constraint` representing the **joint**.

35.4.4.2.4.4. <axis>

The optional **axis** element determines the axis of rotation for any `agx::Hinge` representing a *revolute* or *continuous* **joint**, or the axis of translation for any `agx::Prismatic` representing a *prismatic* **joint**. If not specified, then [1, 0, 0] is used.

35.4.4.2.4.5. <calibration>

Not supported and will be ignored.

35.4.4.2.4.6. <dynamics>

The optional **dynamics** element determines the dynamic behaviour of the `agx::Constraint` representing a **joint**. If not specified the default AGX Dynamics parameters are used for the `agx::Constraint`.

35.4.4.2.4.6.1. damping

Not supported and will be ignored. Note that it is possible to manually set the damping for any DOF for a `agx::Constraint` representing a **joint**. This can be done by accessing the `agx::Constraint` from the returned `agxSDK::Assembly` followed by calling the function *agx::Constraint::setDamping(…)*.

35.4.4.2.4.6.2. friction

The optional *friction* attribute specifies the static friction of the joint in Newtons. This corresponds to a force range of the secondary constraint `agx::FrictionController` of the `agx::Constraint` and is applied as such. If the *friction* attribute is not specified, the default AGX Dynamics friction is used.

35.4.4.2.4.7. <limit>

The **limit** element specifies the upper and lower bounds of of the **joint** both in terms of rotational/translational freedom and in maximal force/torque. The **limit** element is supported only for *revolute* and *prismatic* **joint**'s and for those it is also a required element.

The *lower* and *upper* attributes are represented as the lower and upper range of the secondary constraint `agx::RangeController` and are applied as such. If not specified, then 0 is used for both attributes. The required *effort* attribute specifies the maximum force/torque of the `agx::Constraint`'s secondary constraints: `agx::RangeController`, `agx::LockController` and `agx::TargetSpeedController`. The *velocity* attribute is not supported and will be ignored.

35.4.4.2.4.8. <mimic>

The optional **mimic** element makes it possible to configure an `agx::Constraint` representing the **joint** in such a way that it will move in accordance with another `agx::Constraint`. Internally any **mimic** element is represented as a `agxPowerLine::PowerLine` with a number of `agxPowerLine::Connector`'s together with an `agxDriveTrain::Gear` configured to give this behaviour. The **mimic** feature is supported for *revolute*, *continuous* and *prismatic* **joint**'s. Having a *revolute* or *continuous* **joint** mimic a *prismatic* **joint** or vice versa is not supported, but having a *revolute* **joint** mimic a *continuous* **joint** or vice versa is.

35.4.4.2.4.8.1. multiplier

The optional *multiplier* attribute corresponds to a gear ratio of the `agxDriveTrain::Gear` that is part of the `agxPowerLine::PowerLine` representing the **mimic** element. If not specified the value 1 is used.

35.4.4.2.4.8.2. offset

The optional *offset* attribute will add an initial rotational or translational offset to the `agx::Constraint` representing the **joint** with the **mimic** element. If an initial joint value is set using the *joints* parameter when calling the `agxModel::UrdfReader::read()` function, the *offset* attribute will be added to this initial joint value.

35.4.4.2.4.9. <safety_controller>

Not supported and will be ignored.

35.4.4.3. <transmission>

Not supported and will be ignored.

35.4.4.4. <gazebo>

Not supported and will be ignored.

35.4.4.5. <sensor>

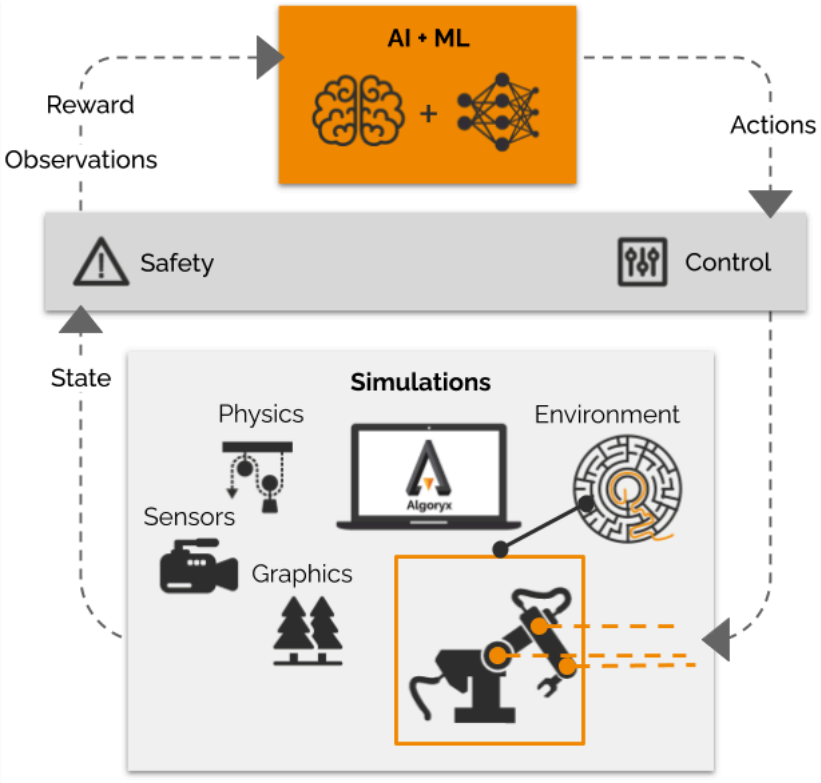
Not supported and will be ignored.

35.4.4.6. <model_state>

Not supported and will be ignored.

36. Reinforcement Learning

This section shows how to use AGX dynamics with Gymnasium-style environments to train reinforcement learning (RL) control policies.



You will learn how to:

- Wrap an AGX simulation as a Gymnasium environment
- Run and visualize example environments
- Train controllers using popular RL libraries

36.1. Installation

Install required Python packages:

```
pip install -r data/python/RL/requirements.txt
```

Note

We do not support platforms with python versions < 3.7.

36.2. AGX gym environment

An `AGXGymEnv` inherits from `Gymnasium.Env` and provides:

- Implementations of the API methods `step()` `render()`, `close()`, `reset()`
- Custom hooks for:
  - `_build_scene()` - Sets up the AGX dynamics simulation (environment).
  - `_set_action()` - Passes the action to the controller. Called in `step()`.
  - `_observe()` - Returns `(obs, reward, terminal, truncated, info)`. Called in `env.step()`.
- Optional camera rendering and graphics.

36.3. Training a control policy

These examples use an AGX implementation of the classic reinforcement learning problem called cart pole, where the goal is to balance a pole on a moving cart. As opposed to classic CartPole, our environment uses continuous actions and the observation space is either the standard state, or visual input from a virtual camera. We show how to train a control policy using stable-baselines3 and run trained control policies.

Run the CartPole environment with a random policy:

```
python data/python/RL/cartpole.py
```

Train a control policy from scratch:

```
python data/python/RL/cartpole.py --train
```

Train using visual observation:

```
python data/python/RL/cartpole.py --train --observation-space visual
```

Train a policy on 4 parallel environments:

```
python data/python/RL/cartpole.py --train --num-env 4 --observation-space visual
```

Load a control policy:

```
python data/python/RL/cartpole.py --load path/to/trained/policy
```

Load one of our pre-trained policies:

```
python data/python/RL/cartpole.py --observation-space visual --load
data/python/RL/policyModels/cartpole_visual_policy.zip
```

36.4. Environment gallery

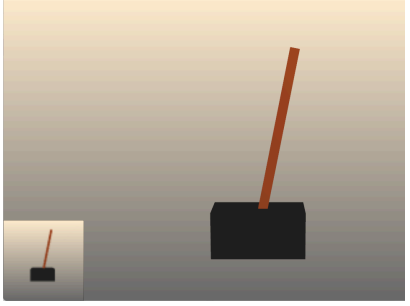
To list all environments:

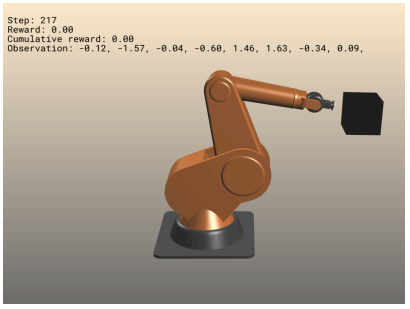
```
python data/python/RL/run_env.py -l
```

To run an environment using a random policy, pass the environment ID to `run_env.py`:

```
python data/python/RL/run_env.py --env agx-pushing-robot-v0
```

36.4.1. Classic and Robotics

Environment	ID
Classic Cartpole	agx-cartpole-v0
Balances a pole on a cart.	
Pushing robot	agx-pushing-robot-v0

Environment	ID
The task of the 2DOF robot is to find the box and push it as far away from as possible.	

36.4.2. Wheel loading

To quickly get started with more complex tasks using heavy vehicle and terrain iteraction, we have several example environments for wheel-loading that operate on AGX terrain. The environments use different vehicle models but share the same goal; to move soil and rocks using the bucket. Being example environments, these do not represent comprehensive environments with tuned reward functions expected to learn useful control policies. The wheel loaders can be controlled using keyboard, a random policy, or a heuristic pre-programmed policy.

Run an environment using the DL300 wheel loader with a random policy:

```
python data/python/RL/run_env.py --env agx-dl300-terrain-v0
```

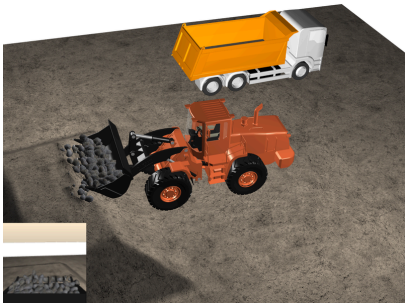
Run using the WA475 wheel loader on a terrain with rocks using keyboard control:

```
python data/python/RL/run_env.py --env agx-wa475-flat-terrain-and-rocks-v0 --policy keyboard
```

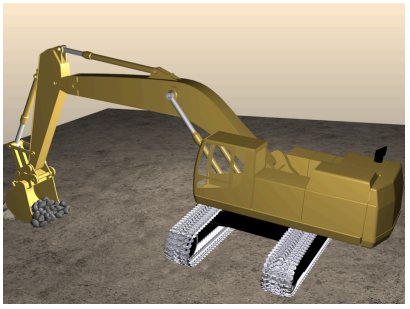
Run using the Algoryx wheel loader with a pre-programmed heuristic policy:

```
python data/python/RL/run_env.py --env agx-algoryxwheelloader-terrain-v0 --policy heuristic
```

Vehicle model	ID
WA475 wheel loader	agx-wa475-terrain-v0
Move soil from a pile using the WA475 wheel loader.	
WA475 wheel loader with rocks	agx-wa475-flat-terrain-and-rocks-v0
Use the WA475 wheel loader to move big rocks placed on an AGX terrain.	

Vehicle model	ID
DL300 wheel loader	agx-dl300-terrain-v0
Move soil from a pile using the DL300 wheel loader.	
L70 wheel loader	agx-l70-terrain-v0
Move soil from a pile using the L70 wheel loader.	
Algoryx wheel loader	agx-algoryxwheelloader-terrain-v0
Move soil from a pile using the Algoryx wheel loader.	

36.4.3. Excavation

Environment	ID
Excavator 365	agx-365-terrain-v0
Move soil on a flat terrain using the 365 excavator.	

37. Strong interaction

Strong interaction is a concept in AGX Dynamics enabling the possibility to write data in the mass matrix part of the system matrix.

The equation system (symmetric version):

$$\begin{bmatrix} \tilde{M} & G^T \\ G & -\varepsilon \end{bmatrix} \begin{bmatrix} v \\ -\lambda \end{bmatrix} = \begin{bmatrix} Mv + hF_{ext} \\ b(M, v, g(q)) \end{bmatrix}$$

The general strong interaction interface enables the user to write the following equation system:

$$\begin{bmatrix} \tilde{M} + A & G^T \\ G & -\varepsilon \end{bmatrix} \begin{bmatrix} v \\ -\lambda \end{bmatrix} = \begin{bmatrix} Mv + hF_{ext} + a \\ b(M, v, g(q)) \end{bmatrix}$$

Where A is symmetric and a is any value.

37.1. Strong interaction constraint

To validate this feature, it is possible to create an instance of `agx::StrongInteractionConstraint` based on any of our ordinary (hinge, prismatic, lock joint etc.) constraints. This will mimic the functionality of the ordinary constraint. The strong interaction constraint will write the constraint data in A and a instead of G. The system for the direct solver to solve then becomes:

$$\left[\tilde{M} - \tfrac{1}{\epsilon} G^T G \right] V_{k+1} = Mv_k + hF_{ext} - \tfrac{1}{\gamma} G^T G v_k - cG^T g(q)$$

Where, for two rigid bodies, $G^T G$ is a 12x12 (symmetric) matrix. I.e., two 6x6 blocks (for rigid body 1 and 2) in the diagonal and one additional 6x6 block, off diagonal.

The code snipped below illustrates the setup:

```
auto frame1 = new agx::Frame();
auto frame2 = new agx::Frame();
auto lock = new agx::LockJoint(body1, frame1, body2, frame2);
auto lockStrong = new agx::StrongInteractionConstraint( lock );
simulation->add( lockStrong );
```

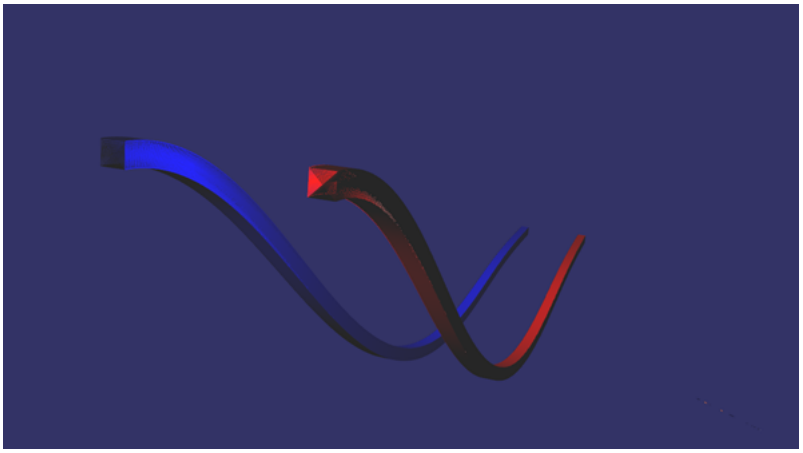


Fig. 37.1 Segmented beams. Blue using lock joints and red the strong interaction constraint concept. The trajectories are close to identical.

37.2. Added mass interaction

The added mass interaction, `agx::AddedMassInteraction`, is the strong interaction defined between one or two rigid bodies. The object contains three matrix blocks (each 6x6) – one for the first rigid body, another one for the second rigid body and a third which is the off diagonal block.

37.2.1. Example

Below is an example that creates an interaction between ship1 and ship2, assigns the data and verifies so that the constructor arguments are one to one with the API:

```
// Create added mass interaction between ship 1 and ship 2.
agx::AddedMassInteractionRef addedMassInteraction = new agx::AddedMassInteraction( ship1, ship2 );

// Enable/disable (enabled by default).
addedMassInteraction->setEnable( false );
addedMassInteraction->setEnable( true );

// Add to a simulation.
simulation->add( addedMassInteraction );

auto blockAssign = []( const agx::Real* data, agx::AddedMassInteraction::Matrix6x6& block )
{
  for ( agx::UInt i = 0; i < 6; ++i )
    for ( agx::UInt j = 0; j < 6; ++j )
      block[ i ][ j ] = data[ 6 * i + j ];
};

// Assign diagonal block for ship1.
blockAssign( ship1AddedMassData, addedMassInteraction->getRigidBody1Storage()->getBlock() );
// Diagonal block for ship2.
blockAssign( ship2AddedMassData, addedMassInteraction->getRigidBody2Storage()->getBlock() );
// Off diagonal block, default assumed to be given in ship1 center of gravity frame.
blockAssign( offDiagonalData, addedMassInteraction->getOffDiagonalBlock() );

agxAssert( addedMassInteraction->getRigidBody1() == ship1 );
agxAssert( addedMassInteraction->getRigidBody2() == ship2 );
agxAssert( addedMassInteraction->getRigidBody1() ==
          addedMassInteraction->getRigidBody1Storage()->getRigidBody() );
agxAssert( addedMassInteraction->getRigidBody2() ==
          addedMassInteraction->getRigidBody2Storage()->getRigidBody() );
```

37.2.2. Coordinate systems

Coordinate systems of the three blocks are currently assumed to be:

1. Rigid body 1 block in rigid body 1 local center of gravity frame
2. Rigid body 2 block in rigid body 2 local center of gravity frame
3. Off-diagonal block in rigid body 1 local center of gravity frame

This is assumptions by the default implementation. The user may change this behavior by implementing the transformations themselves (see MatrixTransformer).

What is well defined for this interaction to be part of a whole system is that the **output** of the added mass interaction must be of the form such that **the velocities are given in the world coordinate frame**.

37.2.3. General transformations

In AGX Dynamics, rotation matrices of a rigid body transforms from a local to the world coordinate system. The transpose of that rotation matrix transforms from world to local.

```
const Matrix3x3 R  = Matrix3x3( rb->getCmRotation() );
const Matrix3x3 RT = R.transpose();
```

Consider an equation system with mass matrices and velocities given in local center of gravity frame:

$$\begin{bmatrix} M_1' & \\ & M_2' \end{bmatrix} \begin{bmatrix} v_1' \\ v_2' \end{bmatrix}$$

From the direct solver point of view, the velocities has to be given in world coordinate frame. In order to get the velocities in world and to transform M' to the world coordinate system we do the following (considering first block row):

$$M_1' v_1' = M_1' (R_1 \underbrace{R_1^T}_{=\mathrm{I}}) v_1'$$

We transform it to the world coordinate system by left multiplying with $R_1^{T'}$

The corresponding matrix form with velocities in world then becomes:

$$\begin{bmatrix} R_1^T M_1' R_1 & \\ & R_2^T M_2' R_2 \end{bmatrix}$$

This is how we transform our center of gravity mass matrices in the solver.

37.2.4. Off diagonal block

One very important aspect of the handling of these strong interaction mass blocks, is that the ordering of the global mass matrix is out of the control for any added mass interaction. This side effect is handled by the internal implementation but will become an issue whenever a custom Matrix transformer is used and hence need to be handled explicitly.

The mass matrix and velocities given N bodies where body *i* and body *j* interacts and has an off diagonal block allocated:

$$\begin{bmatrix} M_1' & & & & \\ & \ddots & & & \\ & & M_i' & & \\ & & \vdots & \ddots & \\ & & A_j' i & \cdots & M_j' \\ & & & & \ddots \end{bmatrix}$$

Following the same procedure as in General transformations we get:

$$\begin{bmatrix} R_1^T M_1' R_1 & & & & \\ & \ddots & & & \\ & & R_i^T M_i' R_i & & \\ & & \vdots & \ddots & \\ & & R_x^T A_j' i R_i & \cdots & R_j^T M_j' R_j \\ & & & & \ddots \end{bmatrix} \begin{bmatrix} R_1^T v_1' = v_1 \\ \vdots \\ R_i^T v_i' = v_i \\ \vdots \\ R_j^T v_j' = v_j \\ \vdots \end{bmatrix}$$

Where R_x^T takes $A_{ji}^{'}$‘$fromitsgivencoordinatesystemand: math:$‘$R_i$ on the right because the off diagonal block is located at column index i (and will be multiplied with v_i).

The default implementation assumes $R_x^T = R_{rb1}^T$, where $rb1$ is the first rigid body passed to the `agx::AddedMassInteraction` object.

The tricky part is that we can’t assume body i and body j in the matrix maps rb_1 and rb_2 in the added mass interaction. The order can change between time steps! I.e., one time step body i is rb_1 but the next time step body i could be rb_2.

Given from the solver:

1. Body i.
2. Block row index of body i, i.e., i as a number.
3. Body j.
4. Block row index of body j, i.e., j as a number.

Hence the added mass interaction has to manually map its bodies to ones sent from the solver.

37.2.5. Matrix transformer

Because of the confusions around transforming the matrices, it’s possible to override the default behavior by implementing the transformation part on the user side.

```
addedMassInteraction->setMatrixTransformer( new MyMatrixTransformer() );
```

Where `MyMatrixTransformer` implements the method transform:

```
void MyMatrixTransformer::transform( const agx::AddedMassInteraction::RigidBodyStorage* rb1Storage,
                                     const agx::UInt rb1BlockRow,
                                     const agx::AddedMassInteraction::RigidBodyStorage* rb2Storage,
                                     const agx::UInt rb2BlockRow,
                                     const agx::AddedMassInteraction::Matrix6x6& offDiagonalBlock,
```

```
                                    agx::AddedMassInteraction::MatrixTransformer::Result& result ) const
{
  // From world to center of gravity frame.
  const agx::Matrix3x3 rb1R  = rb1Storage->getRigidBody() != 0L ?
                               agx::Matrix3x3( rb1Storage->getRigidBody()->getCmRotation() ) :
                               agx::Matrix3x3();
  const agx::Matrix3x3 rb2R  = rb2Storage->getRigidBody() != 0L ?
                               agx::Matrix3x3( rb2Storage->getRigidBody()->getCmRotation() ) :
                               agx::Matrix3x3();
  // From center of gravity frame to world.
  const agx::Matrix3x3 rb1RT = rb1R.transpose();
  const agx::Matrix3x3 rb2RT = rb2R.transpose();

  // Given A_ij are 3x3 blocks, transform( RT, R ) performs:
  //   RT A_00 R     RT A_01 R
  //   RT A_10 R     RT A_11 R
  result.rb1DiagonalBlock = rb1Storage->getBlock().transform( rb1RT, rb1R );
  result.rb2DiagonalBlock = rb2Storage->getBlock().transform( rb2RT, rb2R );

  // Assuming offDiagonalBlock is given in rb1Storage coordinate system!
  // <-> first argument to transform is rb1RT.

  // rb1 block is above rb2 in the matrix, transform from right should be to rb1.
  if ( rb1BlockRow < rb2BlockRow )
    result.offDiagonalBlock = offDiagonalBlock.transform( rb1RT, rb1R );
  // rb2 block is above rb2 in the matrix, transform from right should be to rb2.
  else
    result.offDiagonalBlock = offDiagonalBlock.transform( rb1RT, rb2R );
}
```

Worth noticing is the logic round *rb1BlockRow* and *rb2BlockRow* to choose which matrix to multiply from the right.

If `MyMatrixTransformer` was used in the first example, we could assume that *rb1Storage* contains the data for ship1 and *rb2Storage* the data for ship2. I.e., these aren't changed when the solver changes the order – it's handled by the implementation that calls transform! Overriding transform enables the possibility to have the three blocks in the added mass interaction in ANY coordinate system. The important part to follow is the solver assumption that the velocities are given in the world coordinate system.

38. Performance

This section will describe different methods and techniques that can be applied to simulations in AGX to increase the runtime performance.

Some methods are described in other sections:

- Use multiple threads, see Section 38.6
- Use agxSDK::MergeSplitHandler (AMOR), see Section 39.2
- Use agx::MergedBody (Section 39.1)

38.1. Timeline visualization

Profiling data can be used to identify performance bottlenecks in simulations and guide optimization efforts.

There are two main ways to extract profiling data:

- **Command-line arguments** – In ExampleApplication-based programs (e.g., agxViewer), command-line options (see Table 44.2) can be used to control timeline generation.
- **AGX Dynamics API** – The `agxSDK::Simulation` class provides the methods `enableThreadTimeline` and `disableThreadTimeline` to control when profiling data is gathered.

The collected timelines can be exported in two formats:

- **SVG images** – One image per timestep, stored as separate `.svg` files.
- **Chrome Tracing format** (`ChromeTracing.json`) – Viewable in a Chrome-based browser at `chrome://tracing/`.

38.2. Contact viscosity

If you are using the DIRECT solver for contact friction (see Section 11.16.4) you can sometimes achieve improved performance by increasing the viscosity in the contact material.

38.3. Contact Warmstarting

In Section 11, friction models were explained. Which friction model(s) that are in use can affect the runtime performance in a large way. Scale-box-friction for example provides a more realistic model than box-friction at the expense of a higher runtime cost due to more numeric work in the solver.

The cost of scale-box-friction can in some scenarios be reduced by enabling *contact warmstarting* via

```
agx::DynamicsSystem::setEnableContactWarmstarting( bool enable )
```

This feature will map internal solver states from the previous timestep to the current one by matching the previous contact point positions and normals to the current ones. The solver is then warmstarted and have the possibility to reduce the amount of work needed to solve the system.

When the default friction model is used and no special handling is setup via contact materials, enabling contact warmstarting will just add overhead since extra steps are enabled that checks if data should be cached and if contact points should be matched from timestep to timestep.

Attention

The internal states used for warmstarting contacts are not part of the data that is serialized. Creating a simulation and simulating for e.g. 10 seconds can have slightly different trajectories compared to a different simulation with the same initial state that was simulated for 5 seconds, stored, restored and then simulated an additional 5 seconds.

38.4. Using Iterative contact solver

In Section 3.4 the different solvers available in AGX Dynamics where described. If you have performance challenges and you are willing to sacrifice precision for performance, the Iterative solver can be used for solving contact friction. As described in Section 11.16.4 we can control solve type for each ContactMaterial by creating a Friction model.

Assume for example you have a scenario where you are using a machine (wheel loader, excavator) and you want to interact with many (hundreds) of rocks, the performance might be an issue when using the default (SPLIT) contact model.

Assume you have a Material for the rocks and you want to use the ITERATIVE solve model for anytime two rocks interact:

```
// Create a material for the rocks
// Then (not shown) you need to assign each "rock" geometry the new material
agx::MaterialRef rockMaterial = new agx::Material("Rock")
```

```
// Get (or create if not already created) the ContactMaterial for the Rock/Rock material interaction
agx::ContactMaterialRef rock_rock_cm = simulation->getMaterialManager()->getOrCreateContactMaterial(rockMaterial,
rockMaterial);

// Create a new FrictionModel, we will use the friction model which is default in AGX:
agx::FrictionModelRef fm = new agx::IterativeProjectedConeFriction();
fm->setSolveType( agx::FrictionModel::ITERATIVE );

// Assign the friction model to the contact material
rock_rock_cm->setFrictionModel(fm);
```

By using the code above, AGX will now use the fast ITERATIVE solver for any contacts between two geometries using the “Rock” material.

Section 9.8 show an example in how to combine the ITERATIVE and the DIRECT solver in the same scene.

38.5. AGX Sabre

The overview in Table 3.1 mentioned that agxSabre contains a sparse matrix solver. The sparse matrix is blocked so that BLAS-like kernels can be used to achieve better performance.

By default AGX will select which kernels to use and this default value should be left alone for best performance. But if the user really wants to, the setting can also be changed by the function

```
bool agxSabre::SabreKernels::setPreferredVectorInstructionSet( bitmask )
```

AGX Sabre will check which hardware features that are supported during library initialization. Unsupported features in the input value to setPreferredVectorInstructionSet will then be masked away.

The bit in the instruction set bitmask which has the largest effect is AVX2_BIT which enables fused-multiply-add instructions.

Which kernels to use can also be configured via the environment variable `AGXSABRE_KERNELS` and that variable accepts the same values as `setPreferredVectorInstructionSet`. For verification, the environment variable `AGXSABRE_VERBOSE_KERNELS` can be set to recieve a printout on `stdout` such as `[agxSabre] Detected "GenuineIntel" CPU with HW features 7, preferred features set to 7` to see how the hardware was identified. Additionally, if `AGXSABRE_KERNELS` is also set, two lines will be displayed, e.g.:

```
[agxSabre] Environment set preferred vector instruction set to 15
[agxSabre] Detected "AuthenticAMD" CPU with HW features 15, preferred features set to 15
```

38.6. Parallelization

In Fig. 10.2 we could see that some of the items were marked with an asterisk (*). This indicates that the subtask is a potential target for parallelization. In this section the various tasks will be dissected in more detail.

AGX is built upon the notion of *tasks*. A task can depend on other tasks and can also have sub-tasks. A Task which does not depend on other tasks can run in parallel with other tasks.

For example the class DynamicsSystem is built upon several tasks and subtasks. Such as “UpdateWorldMassAndInertia”, “IntegrateVelocity”, etc. Many of these tasks will split up the work and execute in parallel. Another important feature of AGX is that all of the critical data for rigid bodies, shapes etc. are stored in *buffers*. These buffers are memory allocation blocks, aligned in memory appropriately for SSE optimization. Also, the memory can be made available for other implementation platforms such as OpenCL or CUDA. The executional part of a task is called a *kernel*. It can have several implementations, SSE, non-SSE, OpenCL, OpenGL etc. Which type of implementation that is needed, is selected when the task/kernel is initialized. A kernel is a small executional unit which operates on indata and supplies the result as out-data. It operates directly on the buffers for the fastest possible data access. This schema makes it possible to have some kernels executing on the graphics cards (for example OpenCL kernels), and some on the CPU.

38.6.1. Threads

In AGX there is a pool of threads used for all threaded jobs. The size of this pool is controlled with the call:

agx::setNumThreads(int n);

A negative value for n means that one thread is created per Core/CPU. The default value is 1. All parallelizable tasks are job-oriented. All jobs are put into a queue and scheduled for execution.

38.6.1.1. Executing AGX in threads

AGX creates and manages its own threads in a thread pool. However, to use the AGX API you need to be aware of a few things:

- Any thread that calls the AGX API need to be registered as an AGX thread. This is because various resources need to be available for each thread. To promote a thread to be an AGX Thread call:

  ```
  agx::Thread::registerAsAgxThread();
  ```

  To unregister a thread, call:

  ```
  agx::Thread::unregisterAsAgxThread();
  ```

- Callbacks from AGX such as contact events (10.4.3) or from event listeners (10.4.6) will always be done to the *main thread* for an agxSDK::Simulation. This is the thread that created the agxSDK::Simulation object. To make a thread the main thread for a simulation call (after a call to registerAsAgxThread()):

  ```
  simulation->setMainWorkThread( agx::Thread::getCurrentThread() );
  ```

For an example of this, see tutorial_threads.cpp

38.6.2. Parallel tasks

Table 20 below; show some of the tasks which are parallelizable. These tasks will however not run in parallel to each other, as they depend on data from the previous task. Compare to the time line for the call to Simulation::stepForward() (Fig. 10.2).

Table 38.1 Parallelizable tasks

Name	Description	Belongs to
NarrowPhase	Calculates contact data between two overlapping Geometries.	agxCollide::Space
Update bounding volumes	Update bounding volumes for geometries.	agxCollide::Space
ApplyGravity	Add gravitational force to bodies.	agx::DynamicsSystem
Solver	Solve the constraint system.	agx::DynamicsSystem
IntegratePositions	Integrate transformation and calculate acceleration.	agx::DynamicsSystem
IntegrateVelocity	Integrate Velocity	agx::DynamicsSystem
UpdateWorldMassAndInertia	Will calculate various items needed for the solver.	agx::DynamicsSystem

The Solver stage is only parallelizable if the partitioner can create disjoint groups of bodies which can be solved independently from each other.

38.6.2.1. NarrowPhase

This task will be given a list of overlapping bounding volumes (from the broad phase) calculate the exact contact data for two possibly overlapping geometries. This is a trivially parallelizable task, as there are no data dependency between the different overlapping geometries.

38.6.2.2. Update bounding volumes

This task will update the current bounding volume for a geometry (including its shape transformation and size). Depending on the frame hierarchy, this task can occupy quite some time for a large number of geometries. There are no data dependencies between the geometries. It can therefore be parallelized.

38.6.2.3. Partitioner

Based upon the connectivity in the whole dynamics system, the partitioner can split the system into independent sub-systems.

Two bodies are connected if there is a constraint between them. Constraints including contacts can create large trees of interconnected bodies.

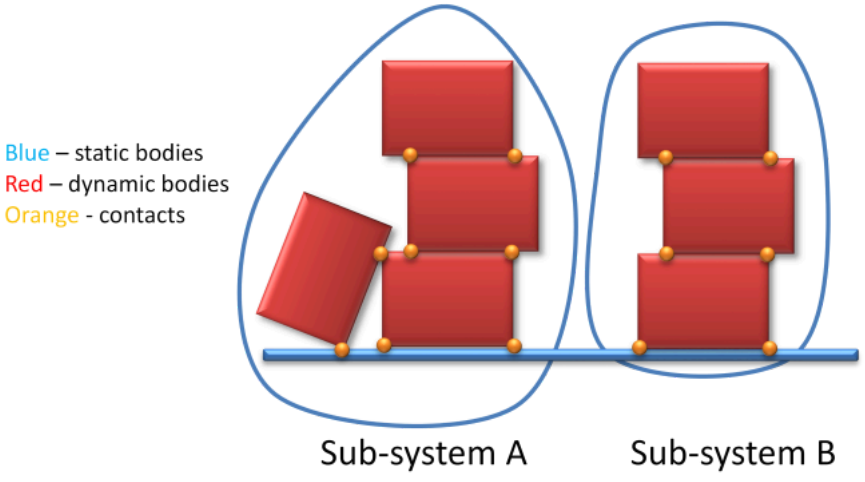


Fig. 38.1 Two separate subsystems.

In the figure above, two separate systems can be identified which can be solved independently from each other, hence the above system would gain if the solver would run the two systems in one thread each. As soon as there is a connection (through a constraint/contact) between two systems, they are merged. *Kinematic bodies* will analogous to static bodies split a system into two parts allowing for parallelization.

38.7. Contact reduction

By using contact reduction, the number of contact points later submitted to the solver as contact constraint can be heavily reduced, hence improving performance.

38.7.1. Motivation

Assuming that a rigid body has one or several geometries associated, when these geometries collide with the geometries from another rigid body, contact points will be generated. Given a complicated geometry or shape, or many overlapping geometries, potentially a large number of contact points will be created. Each contact point will later lead to a contact constraint which must be handled by the solver. Since frictional contacts are relatively expensive to solve, a large number of contact points will have a negative effect on performance. Also, in some cases too many contacts can lead to an over-determined system and cause the simulation to deliver bad results.

For example: a box (primitive) on a plane only needs 3-4 contacts to stand stable whereas in a different scenario where the box is a tesselated mesh resting on a tesselated plane we can easily get hundreds of contact points.

38.7.2. Technical details

The reduction is done by the class agxCollide::ContactReducer, where contact points are reduced by a 6-dimensional binning algorithm in [nxp, n]-space. Here, n is the contact normal and p the contact point relative to a reference point (e.g. one body's center of mass, or the world origin).

In the overall physics pipeline, contacts can get reduced in two separate stages:

- Directly in the narrow phase stage, contact reduction is done on each geometry-geometry interaction (can be controlled, see below). Contact points that are reduced get removed immediately.
- *After* collision detection and *before* stepping the Dynamics system (i.e. PRE in a StepEventListener), contact reduction is done on some rigidbody-rigidbody interactions (can be controlled, see below).

The contact reduction within the physics pipeline can be controlled in three ways:

1. By setting the reduction mode: *ContactReductionMode*. Either no reduction, or per geometry overlap, or per geometry overlap AND per rigid body overlap.
2. By setting the bin resolution b, *number of bins per dimension*. Given a bin resolution of b, this could lead to b^6 contacts being left after reduction in worst case. Average case is about b^2. The bin resolution can be set differently for geometry overlaps and rigid body overlaps. A high value will keep more contacts, lower will result in more aggressive reduction. Commonly a value of 2-3 will give good results. Default is 3. Values from 1 to 10 are valid.
3. By setting the threshold for doing contact reduction. In order to not waste time on doing contact reduction for contacts with small numbers of contact points, contact reduction is only done on contacts which exceed a certain number of contact points. This threshold can be set separately for geometry-geometry-overlaps and rigidbody-rididbody-overlaps.

It is possible to call `agxCollide::ContactReducer::reduce` manually. One motivation for this might be if you want to analyze *all* contact points before sending them to the solver. This can be done by:

- setting the reduction mode to REDUCE_NONE for a contact material,
- and then in a ContactEventListener:
    - analyzing the contacts as desired
    - making a manual call to `agxCollide::ContactReducer::reduce` to reduce the number of contacts that reaches the solver

Note

Modifying contacts might interfere with other ContactEventListeners listening to the same contact! Currently, the only way to modify contact points that should reach the solver is via a ContactEventListener.

38.7.3. ReductionMode

AGX has two steps where contacts are reduced:

- For a pair of overlapping geometries (with potentially several shapes)
- For a pair of overlapping rigid bodies (with potentially several geometries).

By default, AGX does one contact reduction pass between each pair of overlapping geometries.

This behavior can be changed to one of the three modes:

```
/// Specifies the mode for contact reduction
enum ContactReductionMode {
  REDUCE_NONE,               /**< No contact reduction enabled */
  REDUCE_GEOMETRY,           /**< Default: Reduce contacts between geometries */
  REDUCE_ALL                 /**< Two step reduction: first between geometries, and then between rigid bodies */
};
```

The contact reduction mode can be set per contact material. An example:

```
// We want to reduce contacts not only between each geometry, but between body and other body

// Assume we have a body and five geometries, each with e.g. one or more trimesh shapes.
rigidBody->add( geometry1 );
rigidBody->add( geometry2 );
rigidBody->add( geometry3 );
rigidBody->add( geometry4 );
rigidBody->add( geometry5 );

// Assign a material to the geometries
geometry1->setMaterial( material1 );
geometry2->setmaterial( material1 );

// Assume we also have another body which we also assign a material to
otherBody->add( otherGeometry );
otherGeometry->setMaterial( material2 );

// Create a new contact material
agx::ContactMaterialRef cm = simulation->getMaterialManager()->getOrCreateContactMaterial(material1, material2);

// Set contact reduction mode to REDUCE_ALL - now, contacts between rigidBody and otherBody
// get reduced together, not only each contact by itself.
cm->setContactReductionMode( agx::ContactMaterial::REDUCE_ALL );
```

Note that REDUCE_ALL is not the default for a reason:

Often, bodies are assigned different geometries because one would like to set different materials on these geometries, resulting in different physical behavior (e.g. vary friction, Young's modulus, …).

Contact reduction with REDUCE_ALL does not take these differences into account, but reduces over all contact points. However, each contact point will remain in the GeometryContact it belongs to, and only GeometryContacts with a contact material with REDUCE_ALL set will be reduced upon.

All geometries without a rigid body will be treated as belonging to the same one when doing contact reduction between rigid bodies. If you want to avoid geometry contacts from several geometries without rigid bodies to end up in the same reduction, move these geometries into separate (static) rigid bodies.

38.7.3.1. Bin resolution

The bin resolution gives the number of bins per dimension. Given a bin resolution of n, this could lead to n^6 contacts in worst case, where the average case is about c*n^2 (common contact region, like in convex overlaps).

The bin resolution can be set differently for geometry overlaps and rigid body overlaps, as well as for manual contact reduction.

In the general case, a bin resolution of between 1 and 10 is allowed, where typical values are 2 or 3.

Below is a more detailed description.

38.7.3.1.1. Geometry overlaps

The bin resolution for contact reduction for rigid body overlaps can only be given for the whole simulation by setting the value like in this example:

```
// Set contact reduction bin size to 3 for simulation (rigid body overlaps).
sim->setContactReductionBinResolution( 3 );
```

The bin resolution for contact reduction for geometry overlaps can be set per contact material:

```
// Create a new contact material
agx::ContactMaterialRef cm = simulation->getMaterialManager()->getOrCreateContactMaterial(material1, material2);
```

```
// Set contact reduction bin size to 3 for contact material (all geometry overlaps with this contact material).
cm->setContactReductionBinResolution( 3 );
```

If the bin resolution is set to zero (not allowed in other contexts, see above) in the contact material or if no contact material is used, a fallback value from agxCollide::Space is used which can be specified like in this example:

```
// Set contact reduction bin size to 3 for space (fallback value for geometry overlaps).
agxCollide:;SpaceRef = sim->getSpace();
space->setContactReductionBinResolution( 3 );
```

38.7.3.1.2. Rigid body overlaps

The bin resolution for contact reduction for rigid body overlaps can only be given for the whole simulation by setting the value like in this example:

```
// Set contact reduction bin size to 3 for simulation (rigid body overlaps).
sim->setContactReductionBinResolution( 3 );
```

38.7.3.1.3. Manual contact reduction

The bin resolution for manual contact reduction can be given in the reduce-call to contact reducer (a static method):

```
// Set contact reduction bin size to 3 for simulation (rigid body overlaps).
size_t binResolution = 3;
agxCollide::ContactReducer::reduce(contactPoints, binResolution);
```

38.7.3.2. ContactReductionThreshold

The contact reduction between overlapping geometries is enabled by default (otherwise dependent on the material's reduction mode, see chapter above) and will be activated when the number of contact points between two geometries is larger than a specified threshold. This threshold is set globally in agxCollide::Space (since space is responsible for geometries):

```
agxCollide::SpaceRef space = new agxCollide::Space();
// More than 10 contact points will trigger contact reduction
space-> setContactReductionThreshold( 10 );
```

The additional contact reduction between overlapping rigid bodies is disabled by default (otherwise dependent on the material's reduction mode, see ReductionMode) and will be activated when the number of contact points between two rigid bodies is larger than a specified threshold. This threshold is set globally in agxSDK::Simulation:

```
agxSDK::SimulationRef sim = new agxSDK::Simulation();
// More than 10 contact points will trigger contact reduction
sim-> setContactReductionThreshold( 10 );
```

39. Reducing model complexity

To improve performance, AGX Dynamics offers a numbers of approaches to reduce the number of bodies active in a simulation but a the same time retain as much dynamic behaviour as possible. Either the merging of bodies can be done manually using MergedBody or via the Adaptive Model Order Reduction system (AMOR).

- MergedBody – many bodies simulated as one
- Adaptive Model Order Reduction AMOR

39.1. MergedBody – many bodies simulated as one

A merged body is can consist of several agx::RigidBody objects merged together, handled as one single agx::RigidBody in the solver. This functionality can be used to *dramatically reduce the system size*, as seen from the solver, but preserving the mass properties/inertia of a merged system. It can be seen as an extension of a “auto sleep” functionality, but with the huge benefit, that it support merging/sleeping on moving/dynamics bodies.

The functionality of agx::MergedBody can be used in two ways:

- Explicitly/manual: By explicitly creating EdgeInteractions (Section 39.1.3) between bodies which are added to an agx::MergedBody the user has full control over how bodies are merged and split.
- Automatic: Section 39.2 describes functionality for automatic merge/split based on relative velocities, contacts and force measurements.

```
// Create a new merged body.
agx::MergedBodyRef mergedBody = new agx::MergedBody();
// Active the merged body by adding it to the simulation.
simulation->add( mergedBody );
// Deactivating the merged body by removing it from the simulation.
simulation->remove( mergedBody );
```

39.1.1. Applications

Let’s have a look at same sample cases. A good start is to have a distinct parent body. This parent body can for example be a ship, the ground, a harbor, a flatbed, etc.

39.1.1.1. Hydraulic cranes on an offshore vessel

A set of hydraulic cranes on an offshore vessel. Each crane may consist of around 15 bodies and 15-20 constraints. The weight of the crane is significant, affecting the motion of the ship, so when the crane isn’t in use, it cannot just be removed (e.g., only rendered instead).

15 bodies and 15-20 constraints are in general fast to solve but imagine three cranes per vessel and 10 vessels. It’s now 450 extra rigid bodies and about 500 constraints that aren’t explicitly in use.

Now, merging the bodies in the cranes to the ship, the constraints will become inactive (since no relative motion), and the solver will only have to solve for ONE rigid body instead of 45.

39.1.1.2. Logs or rocks on a flatbed

A few hundred logs or rocks on a moving flatbed can in general take a significant amount of time in the solver. For this to be a reasonable scenario, the vehicle pulling the flatbed has to feel the weight and inertia of the load, so the load cannot simply be removed to reduce the time in the solver.

If the stacked logs or rocks can be considered steady, i.e., not moving relative to each other, it’s possible to merge the objects to the flatbed. Instead of a few hundred objects + thousands of contacts to solve, the solver only has one single body, attached to the vehicle to solve for.

39.1.2. The dynamics of merged bodies

The motion control of the final merged body is determined by the motion controls of the set of rigid bodies merged, with the following priority: 1. (if one is) KINEMATICS results in KINEMATICS 2. (if at least one is) STATIC and none KINEMATICS, the result is STATIC 3. (if all are) DYNAMICS and none STATIC or KINEMATICS, the result is DYNAMICS

Note

Only one rigid body with KINEMATICS motion control may occur in a merged body at the time! Having more than one is considered undefined since it’s not possible to determine the final velocity of the bodies involved.

If the final motion control is DYNAMICS, the total mass becomes

$$m^{tot} = \sum_i m_i$$

the center of mass

$$p^{cm} = \frac{1}{m^{tot}} \sum_i m_i p_i^{cm}$$

and finally the inertia

$$I^{tot} = \sum_i I_i - m_i [p^{cm} - p_i^{cm}]^2$$

where

$$[r] = \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}$$

If the motion control **is different from** DYNAMICS, the calculations of center of mass, mass and inertia are ignored.

In the case of KINEMATICS, the linear- and angular velocity will be the one of the kinematic body. In the DYNAMICS case, the final linear- and angular velocity becomes:

$$v^{tot} = \frac{1}{m^{tot}} \sum_i m_i v_i$$

$$\omega^{tot} = \frac{1}{m^{tot}} \sum_i m_i \omega_i$$

39.1.2.1. Advanced mass properties

agx::MergedBody supports rigid bodies with advanced mass properties features. Examples of advanced features are damping (Section 11.2), added mass (Section 21.2.2).

39.1.3. Edge interactions – merging objects together

The internal structure of an agx::MergedBody is a graph where the nodes in the graph is an agx::RigidBody and the edges, connecting the nodes, are so called edge interactions.

An edge interaction defines how the merged rigid bodies interacts and what happens when an edge is removed.

39.1.3.1. Empty edge interaction

An empty edge interaction is a pure logical connection between two rigid bodies.

```
// Merge rb1 <-> rb2 given an empty edge interaction.
mergedBody->add( new agx::MergedBody::EmptyEdgeInteraction( rb1, rb2 ) );
```

39.1.3.2. Contact generator edge interaction

A contact generator edge interaction is an edge that, depending on when the edge is removed, can perform collision detection between the geometries in the first and the second rigid body.

If the contact generator edge is removed before the collision detection is executed, the edge does nothing. I.e., it acts as an empty edge interaction, assuming the objects will participate in the collision detection given the usual update.

If the contact generator edge is removed *after* the collision detection pass has been run and *before* the solver, the edge will check for overlaps between the two bodies and add agxCollide::GeometryContact's to the system. This prevents the two objects to "fall into" each other when the solver solves the system.

```
// Merge rb1 <-> rb2 and generate geometry contacts
// between them if we split in a PRE_STEP callback.
mergedBody->add( new agx::MergedBody::ContactGeneratorEdgeInteraction( rb1, rb2 ) );
```

39.1.3.3. Geometry contact edge interaction

An edge interaction, constructed given an agxCollide::GeometryContact, taking advantage of the contact information, normal and friction forces.

To gain full advantage of this edge, the solver should have seen the geometry contact, i.e., normal and friction forces are available in the contact.

```
// Post-step: If the objects in the geometry contact
// are at rest, merge them.
for ( auto geometryContact : simulation->getSpace()->getGeometryContacts() ) {
  if ( isRestingContact( geometryContact ) )
    mergedBody->add( new agx::MergedBody::GeometryContactEdgeInteraction( *geometryContact ) );
}
```

Similar to agx::MergedBody::ContactGeneratorEdgeInteraction, the contact points and normal are added back, as a new agxCollide::GeometryContact, if the edge is removed after collision detection but before the solver executes.

39.1.3.4. Binary constraint edge interaction

A binary constraint edge interaction is an edge constructed given a one or two (hence binary) body constraint.

This edge does nothing when removed from an agx::MergedBody.

```
// If the hinge motor isn't enabled - merge.
if ( !hinge->getMotor1D()->getEnable() )
  mergedBody->add( new agx::MergedBody::BinaryConstraintEdgeInteraction( hinge ) );
```

39.1.4. Data and state of the merged rigid bodies

If an agx::RigidBody is merged, it's possible to access the agx::MergedBody it belongs to by doing:

```
// Check whether rb belongs to a merged body.
agx::MergedBody* mergedBody = agx::MergedBody::get( rb );
if ( mergedBody != nullptr )
  std::cout << rb->getName() << " is merged." << std::endl;
```

Having access to the agx::MergedBody object, it's possible to traverse/inspect/collect the nodes and edges:

```
// Collect edges connected to 'rb'.
agx::MergedBody::EdgeInteractionRefContainer rbEdges;
agx::MergedBody::EdgeInteractionVisitor visitor = [ &rbEdges ]( agx::MergedBody::EdgeInteraction* edge )
{
  rbEdges.push_back( edge );
};
// 'Visit all edges associated to rb'.
mergedBody->traverse( rb, visitor );

// Print the name of the bodies 'rb' is merged to.
for ( auto edge : rbEdges ) {
  const agx::RigidBody* otherRb = edge->getRigidBody1() == rb ? edge->getRigidBody2() :
                                                                edge->getRigidBody1();
  std::cout << "rb is merged to: " << otherRb->getName() << std::endl;
}
```

A similar example, visiting all neighboring nodes instead of collecting all edges (there may be an arbitrarily number of edges between two bodies):

```
agx::RigidBodyPtrVector otherBodies;
agx::MergedBody::RigidBodyVisitor visitor = [ &otherBodies ]( agx::RigidBody* otherRb )
{
  otherBodies.push_back( otherRb );
};
// 'Visit all neigboring nodes to rb'.
mergedBody->traverse( rb, visitor );

// Print the name of the bodies 'rb' is merged to.
for ( auto otherRb : otherBodies )
  std::cout << "rb is merged to: " << otherRb->getName() << std::endl;
```

Or one can simply do:

```
const auto* otherBodies = mergedBody->getNeighbors( rb );
if ( otherBodies == nullptr )
  std::cout << "rb is not merged in mergedBody" << std::endl;
else
  for ( auto otherRb : *otherBodies )
    std::cout << "rb is merged to: " << otherRb->getName() << std::endl;
```

39.1.4.1. Listeners

It's possible to add listeners to an agx::MergedBody. These listeners are passive, i.e., it's not valid to change the state and not possible to for example prevent a merge, from within a listener.

Callbacks: - **clone()** Create a clone of your listener. It's valid to return the same instance. The clone method is called when the merged body, the listener belongs to, is being split into several agx::MergedBody instances (see: agx::MergedBody::splitIslands). - **onAdd(agx::RigidBody* rb, const agx::MergedBody* mb)** Called when a rigid body is added to the merged body. - **onRemove(agx::RigidBody* rb, const agx::MergedBody* mb)** Called when a rigid body is being removed from the merged body. - **onAdded(EdgeInteraction* edge, const agx::MergedBody* mb)** Called when an edge has been added to the merged body. - **onRemoved(const EdgeInteractionRefContainer& edges, const agx::MergedBody* mb)** Called when a set of edges has been removed from the merged body. - **onMovedFromTo(const EdgeInteractionRefContainer& edges, const agx::MergedBody* fromMergedBody, const agx::MergedBody* toMergedBody)** Called when a set of edges has been moved from one merged body to another.

39.1.5. Splitting merged rigid bodies

If a rigid body is merged, there are several ways to split it. The most convenient way is to do:

```
// Split method finds the merged body associated to 'rb'
// and removes 'rb' from it.
const agx::Bool success = agx::MergedBody::split( rb );
if ( success )
  std::cout << rb->getName() << " successfully removed from its merged body." << std::endl;
else
  std::cout << rb->getName() << " failed to split. Was it merged in the first place?" << std::endl;
```

For more control, use the *remove* method:

```
// Find if 'rb' is merged.
agx::MergedBody* mergedBody = agx::MergedBody::get( rb );
// If merged, remove it (split).
if ( mergedBody != nullptr )
  mergedBody->remove( rb );
```

The *remove* method guarantees to remove any references to the agx::MergedBody as long as *rb* is present in *mergedBody*.

Note

that when you're managing the agx::MergedBody objects, an instance may become empty when removing a rigid body. E.g., two bodies are merged, remove one of them will result in a single body without any edges being the only one left – so it will be removed as well. Leaving the agx::MergedBody object empty. An update to the above example:

```
// Find if 'rb' is merged.
agx::MergedBodyRef mergedBody = agx::MergedBody::get( rb );
// If merged, remove it (split).
if ( mergedBody != nullptr ) {
  mergedBody->remove( rb );
  // If the remove of 'rb' results in an empty 'mergedBody',
  // remove 'mergedBody' from the simulation.
  if ( mergedBody->isEmpty() )
    simulation->remove( mergedBody );
  mergedBody = nullptr;
}
```

It's also possible to split a rigid body by removing all the edges to its neighboring nodes/bodies:

```
// Collect edges connected to 'rb'.
agx::MergedBody::EdgeInteractionRefContainer rbEdges;
agx::MergedBody::EdgeInteractionVisitor visitor = [ &rbEdges ]( agx::MergedBody::EdgeInteraction* edge )
{
```

```
  rbEdges.push_back( edge );
};
// 'Visit all edges associated to rb'.
mergedBody->traverse( rb, visitor );

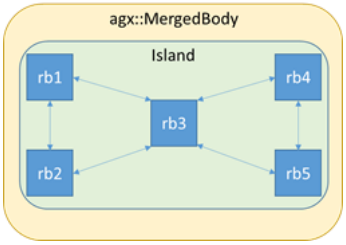
// Remove all the edges associated to our 'rb'.
for ( auto edge : rbEdges )
  mergedBody->remove( edge );

// When all edges has been removed, 'rb' hasn't got
// any connections left in 'mergedBody' so it will
// be removed.
agxAssert( agx::MergedBody::get( rb ) == nullptr );
```

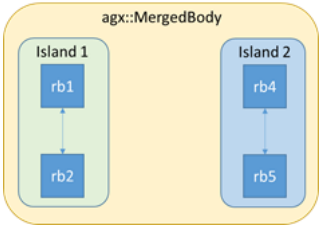
39.1.6. Separate islands within an agx::MergedBody

The agx::MergedBody doesn't assume that all rigid bodies/nodes are connected. E.g, if you have four bodies *rb1, rb2, rb3* and *rb4*, and add edge interactions *rb1* <-> *rb2* and *rb3* <-> *rb4* to the agx::MergedBody – it's a valid, but maybe not a desired state.

A more natural example. Say we have five rigid bodies, merged like this:



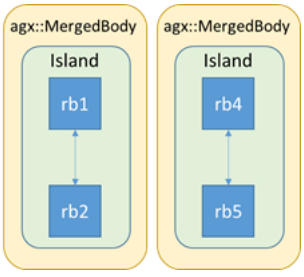
Removing *rb3* will result in the following merged state:



where there aren't any interactions connecting *island 1* to *island 2*. It's not possible to query the merged body how many islands it consists of since it needs a full graph traversal to determine this. Still, if the desired behavior is there shouldn't be separate islands in my merged bodies' it's possible to call agx::MergedBody::splitIslands. This method will determine separate islands, create new merged bodies and add them to the simulation.

```
// Checks for separate, non-interacting, islands in mergedBody.
// NOTE: mergedBody has to be in the simulation and all new
//       agx::MergedBody objects will be added to the same
//       simulation.
agx::MergedBodyRefVector newIslands = mergedBody->splitIslands();
std::cout << "Number of new islands: " << newIslands.size() << std::endl;
```

Resulting in:



39.1.7. Merging two already merged objects

For obvious reasons (it's rigid, it cannot move in different directions at the same time!), a rigid body may only belong to one agx::MergedBody. As a result of this constraint, an edge will be rejected by a merged body if one (or both) of the bodies included is merged to another merged body. This state has to be

explicitly handled by the user.

Given two rigid bodies *rb1* and *rb2*, merged with other objects in *mergedBody1* and *mergedBody2*, it's possible to merge the two agx::MergedBody objects together given an edge between *rb1* and *rb2*:

```
agx::MergedBody* mergedBody1 = agx::MergedBody::get( rb1 );
  agx::MergedBody* mergedBody2 = agx::MergedBody::get( rb2 );
  // Copying all data from mergedBody2 into mergedBody1 and
  // adding the empty edge interaction to mergedBody1.
  agx::MergedBody::EmptyEdgeInteractionRef edge = new agx::MergedBody::EmptyEdgeInteraction( rb1, rb2 );
  const agx::Bool success = mergedBody1->merge( mergedBody2, edge );
  // If merge is successful, remove the empty mergedBody2
  // from the simulation.
  if ( success )
    simulation->remove( mergedBody2 );
```

39.1.8. Active and inactive

Consider the example scenario described in Hydraulic cranes on an offshore vessel. The crane is not in use so we have it merged. The state of the agx::MergedBody containing the objects, is *active*:

```
// Adding 'craneMergedBody' will active it and the containing
// rigid bodies will be merged.
simulation->add( craneMergedBody );
```

When the crane is ready for use, e.g., moving the boom, we want to inactivate the agx::MergedBody. This can of course be done in many different ways, but one convenient thing of the agx::MergedBody objects is that it will preserve its internal structure when removed from the simulation. The objects merged will simply not be merged anymore.

Let a prismatic be the constraint we control the boom hydraulic piston/cylinder with. When the prismatic motor is active we expect the objects to be able to move relative each other, and when the motor is inactive – we may merge the crane objects again. This can be achieved by adding and removing the *craneMergedBody*:

```
if ( prismatic->getMotor1D()->getSpeed() != 0.0 ) {
    // Motor is active, split all objects in 'craneMergedBody'
    // by removing it from the simulation.
    if ( craneMergedBody->isInSimulation() )
      simulation->remove( craneMergedBody );
  }
  else {
    // Motor isn't active, merge all objects in 'craneMergedBody'
    // by adding it to the simulation.
    if ( !craneMergedBody->isInSimulation() )
      simulation->add( craneMergedBody );
  }
```

This means that a rigid body can be associated to an agx::MergedBody but it's still not actively merged. It's possible to check whether an objects is actively merged (i.e., not seen by the solver) or passively merged (i.e., part of a merged body, but that merged body is not in a simulation):

```
// agx::MergedBody::get returns the merged body 'craneBoom' is part of.
const agx::Bool isPartOfAMergedBody = agx::MergedBody::get( craneBoom ) != nullptr;
// agx::MergedBody::getActive returns the merged body 'craneBoom' is part
// of IF that merged body is a part of the simulation.
const agx::Bool isActivelyMerged = agx::MergedBody::getActive( craneBoom ) != nullptr;
```

During normal usage there's no need to use the *getActive* method.

39.2. Adaptive Model Order Reduction AMOR

AGX Adaptive Model Order Reduction (AMOR) are algorithms where a simulated system adapts to a size where the dynamics is presumed to be preserved. It aims to reduce the number of degrees of freedoms given conditions of steady states. I.e., if the relative motion between two objects is zero, it's possible to consider these two objects as one, as long as the local system isn't affected by external interactions that may change their relative motion.

The purpose of AMOR is solely to reduce the workload of the dynamics solver and collision detection (broad phase excluded).

In the previous section we covered how to explicitly merge and split rigid bodies to/from each other, using the agx::MergedBody object. This section is about an *automatic version of merging and splitting rigid bodies*, where the adding, removing and managing of the agx::MergedBody objects is handled by an `agxSDK::MergeSplitHandler` object.

39.2.1. Enabling and disabling the agxSDK::MergeSplitHandler

Every instance of an agxSDK::Simulation has an instance of agxSDK::MergeSplitHandler. To enable or disable the merge split handler, simply:

```
simulation->getMergeSplitHandler()->setEnable( true );
simulation->getMergeSplitHandler()->setEnable( false );
```

Disabling the handler will split all objects merged by it.

39.2.2. agxSDK::MergeSplitProperties

The agxSDK::MergeSplitProperties defines if and how an object may merge and/or split to and from other objects. By default, the objects doesn't carry an instance of the agxSDK::MergeSplitProperties, and this is interpreted as 'merge and split of this object is disabled'.

To get an already created or to create a new instance of MergeSplitProperties for an object (RigidBody, Geometry, Constraint or Wire):

```
agxSDK::MergeSplitProperties* properties = agxSDK::MergeSplitHandler::getOrCreateProperties( obj );
agxAssert( obj == nullptr || properties != nullptr );
```

Given *obj != nullptr* then *properties != nullptr*.

With the merge split properties it's possible to:

- **Enable and disable merge (disabled by default).** When *merge* is enabled, it means that this object may merge to other objects.
- **Enable and disable split (disabled by default).** When *split* is enabled, it means that this object (if merged), may split from the object it's merged to.
- **Enable and disable merge and split (disabled by default).** Convenience method to enable/disable both merge and split.
- **Manage thresholds used by the AMOR algorithms.** Get, create and set thresholds used by contact, constraint and wire AMOR algorithms.
- **Manage merge ignore groups.** Merge ignore groups are used to prevent merging between bodies that would otherwise have merged.

Objects that can carry an agxSDK::MergeSplitProperties instance are; *agx::RigidBody, agxCollide::Geometry, agx::Constraint* and *agxWire::Wire*:

```
agxSDK::MergeSplitProperties* rbProperties         = agxSDK::MergeSplitHandler::getOrCreateProperties( rb );
agxSDK::MergeSplitProperties* geometryProperties   = agxSDK::MergeSplitHandler::getOrCreateProperties( geometry );
agxSDK::MergeSplitProperties* constraintProperties = agxSDK::MergeSplitHandler::getOrCreateProperties( constraint );
agxSDK::MergeSplitProperties* wireProperties       = agxSDK::MergeSplitHandler::getOrCreateProperties( wire );
```

To check if an objects has properties or to inspect the current state:

```
const agxSDK::MergeSplitProperties* rbProperties         = agxSDK::MergeSplitHandler::getProperties( rb );
const agxSDK::MergeSplitProperties* geometryProperties   = agxSDK::MergeSplitHandler::getProperties( geometry );
const agxSDK::MergeSplitProperties* constraintProperties = agxSDK::MergeSplitHandler::getProperties( constraint );
const agxSDK::MergeSplitProperties* wireProperties       = agxSDK::MergeSplitHandler::getProperties( wire );
```

39.2.2.1. Properties carried by agxCollide::Geometry or its parent – agx::RigidBody

Since the context of AMOR is 'bodies', the properties for agxCollide::Geometry behaves a bit different. It is, as shown above, possible to create and change the merge split properties of a geometry object making it possible to, for example, have merge enabled/disabled in different parts of a rigid body.

A geometry inherits the merge split properties of its parent rigid body. Assume we've created merge split properties for the rigid body and enabled merge:

```
agx::RigidBodyRef rb = new agx::RigidBody();
agxCollide::GeometryRef geometry = new agxCollide::Geometry( new agxCollide::Sphere( 0.5 ) );
rb->add( geometry );
...
// Create merge split properties and enable merge for 'rb'.
agxSDK::MergeSplitProperties* rbProperties = agxSDK::MergeSplitHandler::getOrCreateProperties( rb );
rbProperties->setEnableMerge( true );
```

Now, for geometries, the semantics of the *getProperties* method differs from the rigid body, constraint and wire versions in two ways:

1. There's a mutable version.
2. If the geometry hasn't got any merge split properties (i.e., *getOrCreateMergeSplitProperties* hasn't been called), the merge split properties of the parent rigid body will be returned (if created).

```
// 1.
agxSDK::MergeSplitProperties* geometryProperties = agxSDK::MergeSplitHandler::getProperties( geometry );
// 2.
agxAssert( geometryProperties == agxSDK::MergeSplitHandler::getProperties( rb ) );
```

Also, in the call to *getOrCreateProperties*, given a geometry, the merge split properties (if created) of the parent rigid body (if present) will be cloned. Continuing the example from above (remember *merge* is set to be enabled for rb):

```
agxSDK::MergeSplitProperties* newProperties = agxSDK::MergeSplitHandler::getOrCreateProperties( geometry );
// We should get a new instance of the merge split properties.
agxAssert( newProperties != geometryProperties );
// The new properties should be a clone of 'rbProperties'.
agxAssert( newProperties->getEnableMerge() == rbProperties->getEnableMerge() );
```

39.2.3. Merge Split Thresholds

Similar to *agxSDK::MergeSplitProperties*, each object may carry a set of parameters/thresholds that controls the behavior when merging and splitting the object. The global (simulation specific) merge split thresholds are used by default.

39.2.3.1. Global (simulation specific) merge split thresholds

The *agxSDK::MergeSplitHandler* has the default thresholds used for objects without explicitly created thresholds:

```
auto globalContactThresholds = simulation->getMergeSplitHandler()->getGlobalContactThresholds();
// Set global contact thresholds for all objects without explicitly created thresholds.
configureGlobalContactThresholds( globalContactThresholds );
```

39.2.4. Object specific merge split thresholds

All objects that may have *agxSDK::MergeSplitProperties* may have a list of thresholds. E.g., *agx::RigidBody*, *agxCollide::Geometry*, *agxWire::Wire* and *agx::Constraint*. The thresholds may be accessed, created and removed via the merge split properties API.

There are two ways to assign object specific thresholds:

```
auto rbProperties = agxSDK::MergeSplitHandler::getOrCreateProperties( rb );
// By default the thresholds should be null meaning global thresholds are used.
agxAssert( rbProperties->getContactThresholds() == nullptr );
auto explicitContactThresholds = new agxSDK::GeometryContactMergeSplitThresholds();
rbProperties->setContactThresholds( explicitContactThresholds );
```

```
auto rbProperties = agxSDK::MergeSplitHandler::getOrCreateProperties( rb );
// By default the thresholds should be null meaning global thresholds are used.
agxAssert( rbProperties->getContactThresholds() == nullptr );
auto explicitContactThresholds = rbProperties->getOrCreateContactThresholds();
```

To remove the thresholds and go back to the global default, simply assign null:

```
rbProperties->setContactThresholds( nullptr );
```

The explicit thresholds instance may be shared arbitrarily:

```
auto explicitContactThresholds = new agxSDK::GeometryContactMergeSplitThresholds();
auto rbProperties = agxSDK::MergeSplitHandler::getOrCreateProperties( rb );
auto wireProperties = agxSDK::MergeSplitHandler::getOrCreateProperties( wire );
auto geometryProperties = agxSDK::MergeSplitHandler::getOrCreateProperties( geometry );

rbProperties->setContactThresholds( explicitContactThresholds );
```

```
wireProperties->setContactThresholds( explicitContactThresholds );
geometryProperties->setContactThresholds( explicitContactThresholds );
```

39.2.5. Wire merge split thresholds

The wire merge split properties and thresholds are shared with all objects created/owned by the wire instance. There are currently two occasions with predefined behaviors:

1. **Cutting a wire** - When cutting a wire the properties and thresholds for that wire are cloned, i.e., new instances of the properties and thresholds are created. The newly created wire will have identical values as before but when one, e.g., changes thresholds of the original wire it won't affect the part that has been cut from the original one.
2. **Merging wires** - When merging two *wires wire1->merge(wire2)* the properties and thresholds of *wire1* will be used for the whole wire.

39.2.6. Merge conditions

Two objects may merge when the objects involved has merge split properties and property merge enabled.

```
agxSDK::MergeSplitHandler::getOrCreateProperties( rb1 )->setEnableMerge( true );
agxSDK::MergeSplitHandler::getOrCreateProperties( rb2 )->setEnableMerge( true );
```

39.2.6.1. Resting contact

The contacts points carries three directions (an orthonormal basis), namely the normal N, the tangent U and the tangent V directions. U is the primary friction direction and V the secondary friction direction. Given two rigid bodies

$rb_1 = p_1^{cm}, v_1, \omega_1,$

$rb_2 = p_2^{cm}, v_2, \omega_2,$

where p_i^{cm} is center of mass position, v_i the linear velocity and ω_i the angular velocity. The speed s_d along a given direction d at point p_d is given by

$s_d = d * [v_1 + \omega_1 \times (p_d - p_1^{cm}) - [v_2 + \omega_2 \times (p_d - p_2^{cm})]]$.

Given the speeds s_N, s_U and s_V along each direction of the contact points p_i, the two objects may merge if:

$$\max_{p_i} \mid s_N \mid \leq \epsilon_N$$

$$\max_{p_i} \mid s_U \mid \leq \epsilon_{UV}$$

$$\max_{p_i} \mid s_V \mid \leq \epsilon_{UV}$$

Where ϵ_N is called MAX_RELATIVE_NORMAL_SPEED and ϵ_{UV} MAX_RELATIVE_TANGENT_SPEED. Since the conditions above recovers the rolling condition there's a third condition regarding rolling,

$$\| \omega_1 - \omega_2 \| \leq \epsilon_\omega$$

and ϵ_ω is named MAX_ROLLING_SPEED.

```
auto contactThresholds = objProperties->getOrCreateContactThresholds();

// Assign maximum speed along a contact normal for a contact
// to be considered resting. Default: 0.01.
contactThresholds->setMaxRelativeNormalSpeed( nS );

// Assign maximum (sliding) speed along a contact tangent
// for a contact to be considered resting. Default 0.01.
contactThresholds->setMaxRelativeTangentSpeed( tS );

// Assign maximum (rolling) speed for a contact to be considered
// resting. Default 0.01.
contactThresholds->setMaxRollingSpeed( rS );
```

When two objects merge due to a resting contact state, the current contact data (such as contact point, normal, normal- and friction forces) is stored and is used when the objects splits and to determine if they should split due to external interactions.

39.2.6.2. Constraint in a steady state

Similar to contact points, constraints keeps track of the speeds at the anchor point of the constraint. Since constraints (like Hinge, Prismatic, LockJoint etc.) are persistent on a different level compared to contacts, it's possible to include 'time' dependent data. It's not time in seconds, rather "this constraint has

been under constant load for some time - merge!". The 'time' concept is introduced by using an Exponential Moving Average (EMA) operation on the relative speeds.

Given a statistic S and an i'th observation O_i the i'th EMA statistic is given by

$S_i = \alpha O_i + (1 - \alpha) S_{i-1}$

where $0 \leq \alpha \leq 1$ is a smoothing factor, controlling the sensitivity of the new observations. When α is small, the relative speeds has to be small for a (much) longer time than when α is close to one.

α isn't exposed to be manipulated by the user. Instead the more intuitive threshold of *maximum relative speed* may be changed to make the constrained system merge more or less aggressively.

```
auto constraintThresholds = objProperties->getOrCreateContactThresholds();

// Assign maximum relative speed between the constrained objects
// for the system to be considered at rest. I.e., when the relative
// motion between the objects is less than this threshold, the
// objects may merge. Default: 0.005.
constraintThresholds->setMaxRelativeSpeed( maxRelSpeed );
```

39.2.6.3. Splitting Wires

Wires contains bodies, geometries and constraints so when a wire object is merging, the specific objects state and properties are checked. E.g., when a mass node is in contact with another object, the contact thresholds and algorithms are used to determine if the node should be merged. Similar for the wire attachments - the constraint thresholds and algorithms are used.

```
auto wireProperties = agxSDK::MergeSplitHandler::getOrCreateProperties( wire );

// Thresholds controlling the wire attachments.
auto wireAttachmentsThresholds = wireProperties->getOrCreateConstraintThresholds();

// Thresholds controlling the merging and splitting of
// mass/lumped nodes relative other objects.
auto wireContactThresholds = wireProperties->getOrCreateContactThresholds();
```

The wires has its own thresholds as well but they are currently classified as split thresholds.

39.2.6.4. Merge ignore filters

A merge ignore filter is used to prevent merges between bodies that would otherwise have merged. The filtering is done based on groups which are added to `MergeSplitProperties`. A group is specified by either a group ID, which is an `agx::UInt32`, or a group name:

```
agx::UInt32 groupById = agx::UInt32(4);
agx::Name groupByName = "group";
properties->addGroup(groupById);
properties->addGroup(groupByName);
```

A filter is a pair of such groups and is registered with the `MergeSplitHandler`:

```
simulation->getMergeSplitHandler()->setEnableMergePair(groupById, groupByName, false);
```

After this the `MergeSplitHandler` will not automatically merge any pair of bodies where one body contains the group `groupById` and the other contains the group `groupByName`. Constraints with merge split enabled is a special case. In this case the constraint's `MergeSplitProperties` takes precedence and its set of groups is used when comparing against the registered filters.

The filters are checked only for the groups of the two bodies that are currently being considered for merging. If either body has already been merged with something else then there may be other merge ignore group within that merged body, but those are not considered. This means that two bodies for which there is a merge ignore filter may still become part of the same merged body if they both merge with some third body against which neither of the two has a merge ignore filter.

Filters cannot be used to enable merging between pairs of bodies, it is an ignore filter only. Passing `true` to `setEnableMergePair` will only remove the ignore filter for the given pair and restore the default behavior.

Named groups are given an ID as well. In order to separate user specified group IDs from IDs generated from group names, a user specified group ID must be less than `MergeIgnoreFilter::MAX_UNAMED_GROUP_ID`. The collection of IDs return by `MergeSplitProperties::getGroupIds` will include both user specified group IDs and the group IDs generated from the named groups added to the `MergeSplitProperties`. The ID for a named group can

be added only by adding the name, not by adding the ID directly. However, removal based on named group ID is allowed and results in both the ID and the name being removed.

The relationship between merge ignore groups on geometries and the owning rigid body is the same as for any `MergeSplitProperties` property: the one in the geometry, if any, takes precedence over the one in the rigid body.

39.2.7. Split conditions

An object may split, i.e., leave a merged state, if it has merge split properties and property *split* is enabled.

```
agxSDK::MergeSplitHandler::getOrCreateProperties( rb )->setEnableSplit( true );
```

39.2.7.1. Impacting contact

There are two different concepts to choose from:

- Physical impact
- Logical impact

The logical impact is determined by the state of the contact. I.e., if the state were “no contact” and the geometries are overlapping, the state becomes “impact”. It’s at this state the agxSDK::ContactEventListener::impact callbacks are executed.

The physical impact condition is tied to a speed threshold, preventing objects from splitting too often.

The logical impact concept is disabled by default. If enabled, the merged object will split when the state of the contact is “impact”. If the state is different from “impact” the physical impact approach is testing the contact

The impact approach uses the relative speed s_N (from the Resting contact section), considering the sign of the speed as well:

$$\max_{p_i} s_N \leq -\epsilon_{impact}$$

Note that s_N is positive when the objects are separating and negative when they’re approaching each other. The latter is the one we’re catching and $\epsilon_{impact} \geq 0$ is a threshold called MAX_IMPACT_SPEED.

```
auto contactThresholds = objProperties->getOrCreateContactThresholds();

// Assign maximum impact speed (along a contact normal) a merged
// object can resist without being split. Default 0.01.
contactThresholds->setMaxImpactSpeed( value );

// True to split when geometry contact state is agxCollide::GeometryContact::IMPACT_STATE,
// i.e., the first time the objects collide. Default is false and "max impact speed" will
// be used instead.
contactThresholds->setSplitOnLogicalImpact( flag );
// Disable physical impacts completely when using logical impacts.
// This is optional and it's valid to combine the two.
if ( flag )
  contactThresholds->setMaxImpactSpeed( agx::Infinity );
```

39.2.7.2. Constraints

39.2.7.2.1. Changing state

Constraints can split bodies when the internal state of the constraint has been changed in a way that the dynamics may be affected. Currently, this functionality is focused to the secondary constraints/controllers and the constraint will split the bodies involved when:

- The motor is enabled with speed different from zero.
- The position of the lock has been changed (i.e., by calling constraint->getLock1D()->setPosition(newPosition)).
- The range of the range controller has been changed.
- When a controller state goes from inactive to active, e.g., when a range is hit or a lock is enabled.

It’s, unfortunately, often not enough to just split the two bodies in the constraint. Consider an articulated, hydraulic crane where the whole structure is merged. To rise the boom the prismatic motor is activated, but nothing will move since it’s only the hydraulic cylinder and piston that are free to move.

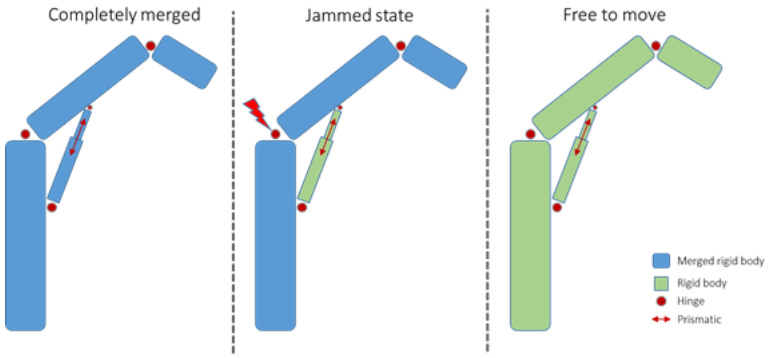


Fig. 39.1 Articulated crane where a prismatic motor is used to rise the boom.

What happens is that the split algorithm first splits the two bodies directly involved in the constraint. The structure is now in a “jammed state” where it still cannot move. The split algorithm continues to traverse the merged structure, splitting all constrained bodies it finds. The split-traversal stops when it reaches a body that may not be split or when the edges connecting two bodies aren’t related to constraints (i.e., different from agx::MergedBody::BinaryConstraintEdgeInteraction).

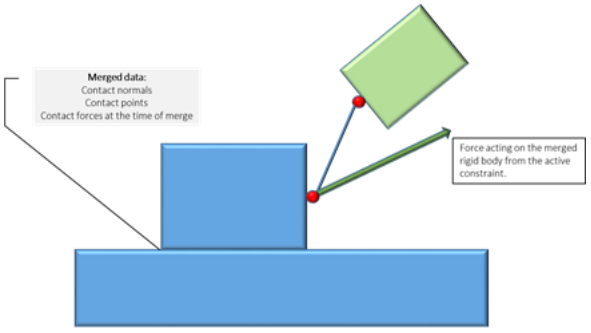
If, for example, hundreds of these cranes were to stand on a static ground or a dynamic ship, it’s important that the ground or the ship doesn’t have merge split property *split* enabled. In general there aren’t any reasons to have that property enabled for “parent” objects.

39.2.7.2.2. Applying forces

When a constraint is applying forces on a merged object, it may split that object if that object has merge split property *split* enabled and is merged due to resting contacts.

Mentioned in the Resting contact section – contact and interaction data is saved when two objects merge. This data can be used to approximately determine “how much” external force is needed for the object to start to move, and we can call it *the strength of a contact edge*.

When the force exceeds the contact edge strength, the edge can be considered removed and if all edges are removed, the objects splits.



There are some thresholds in the contact thresholds that controls the strength of a merged contact. In general contacts splits due to impacting objects but when a constraint is applying forces it’s possible to add some adhesiveness to the merged objects - preventing them from splitting at the default moment.

The merged contact adhesive forces are given in unit of force:

```
auto contactThresholds = objProperties->getOrCreateContactThresholds();

// Adhesive force in the normal directions preventing the object to split (if > 0)
// when the object is subject to external interactions (e.g., constraints).
// Default: 0.0
contactThresholds->setNormalAdhesion( value );

// Adhesive force in the tangential directions preventing the object to split (if > 0)
// when the object is subject to external interactions (e.g., constraints).
// Default: 0.0
contactThresholds->setTangentialAdhesion( value );
```

39.2.7.2.3. Split when applying external forces

As mentioned in the previous section, only external forces from constraints may split merged objects. I.e., *rb->addForce(largeForce);* will by default not split *rb* given no constraints involved. Disabled by default, but possible to enable, is a flag called *may split in gravity field*, meaning monitor this objects added external forces and interpret them in the same way as if a constraint was interacting with it.

```
auto contactThresholds = objProperties->getOrCreateContactThresholds();

// Check split given external forces for all objects merged (i.e., rb->getForce()
// the sum of rb->addForce(), including the gravity force). Performance warning,
// disabled by default.
contactThresholds->setMaySplitInGravityField( flag );
```

Note

Use this feature with as few objects as possible (if needed to begin with), since merged objects with this feature enabled are going through split tests each time step.

39.2.7.3. Merging Wires

Wire nodes merge when the contact between the wire node and another object is considered resting (thresholds in *agxSDK::GeometryContactMergeSplitThresholds*). When the contact is in a resting state we store a value of the current tension round the node, giving a measure of how the wire is looking before the merge, then we merge the node. Due to state changes in the simulation, the tension may change, resulting in a split of the wire node.

39.2.7.3.1. Merge tension scale

The *merge tension scale* threshold (default: 1.0) says something about how much the tension must change before the wire node will split. It's a scale of the stored merge tension meaning value < 1 it's more likely, and > 1 less likely for the wire node to split. For system with (locally) low tension but large objects interacting with them, this value could (for example) be round 100. I.e., when the wire/chain should split, the tension is likely to be 100 times larger than when it merged.

```
auto wireThresholds = wireProperties->getOrCreateWireThresholds();

// When a node is merged the tension is stored and monitored
// to perform split. This threshold scales the merge tension
// making split more likely when < 1 and less likely > 1. When
// this scale is 0 the only force keeping the node merged is
// from the contact - which in most cases isn't enough.
// Default: 1.0
wireThresholds->setMergeTensionScale( value );
```

39.2.7.3.2. Split propagation decay scale

The *split propagation decay scale* threshold (default: 1.0) says something about how far splits, due to external forces, propagates along a merged wire segment.

When a node splits, x amount of energy is consumed, and this threshold scales this energy consumption. So if this threshold is 0.0, all nodes will feel the same interaction forces as the first/last node did (probably splitting the whole wire when one splits). When this threshold is 1.0, the external interactions are receiving a measure of how heavy the merged wire is to pull/push, but not much more. When you're experiencing undesirable splits back and forth along the wire, increase this value slightly (1.0 - 20.0). To completely disable split propagation, set it to some value >> 1 (1.0E4 on the safe side and infinity to completely disable).

```
auto wireThresholds = wireProperties->getOrCreateWireThresholds();

// When external forces are acting on a partially merged wire,
// the force will propagate and split several nodes at once.
// This threshold controls the amout of force (of the total
// external force) that is used to split each node. If this
// value is high (> 1), the force will not propagate 'too'
// long, keeping the wire merged. Default: 1.0.
wireThresholds->setForcePropagationDecayScale( value );
```

39.2.7.4. Split on separation

There are cases where bodies remain merged after a contact that held the bodies in place disappears. Some of those cases can be handled by enabling splitting on separation events.

```
simulation->getMergeSplitHandler()->setEnableSplitOnSeparation(true);
```

The effect of split on separation is that when two bodies separate then they are split from their respective agx::MergedBody, if any. A side effect of this more aggressing splitting is that it becomes more difficult to form stable piles of merged bodies.

39.2.8. agx::MergedBody with agxSDK::MergeSplitHandler

As mentioned earlier, the agxSDK::MergeSplitHandler manages agx::MergedBody objects. If the agxSDK::MergeSplitHandler encounter a merged rigid body, but the rigid body wasn't merged by the handler – the rigid body is ignored.

This enables the possibility for the user to manage their own merged bodies. It can, for example, be used to initialize big scenes in a merged state.

For example – create a pile of boxes, enable merge and split and merge them in an agx::MergedBody:

```
// Create and add a merged body.
agx::MergedBodyRef mergedBody = new agx::MergedBody();
simulation->add( mergedBody );

agx::RigidBodyRef prevBox = nullptr;
for ( agx::UInt i = 0; i < numBoxes; ++i ) {
  // Create, position and add the box to the simulation.
  agx::RigidBodyRef box = createBoxInPile( i, simulation );
  // Enable merge+split for the box.
  agxSDK::MergeSplitHandler::getOrCreateProperties( box )->setEnableMergeSplit( true );
  // Merge with previous box.
  if ( prevBox != nullptr )
    mergedBody->add( new agx::MergedBody::ContactGeneratorEdgeInteraction( prevBox, box ) );
  prevBox = box;
}
```

Running this simulation, given a ground object with property merge enabled, the set of merged boxes will interact with the ground object, but not merge nor split.

For the agxSDK::MergeSplitHandler to be able to merge and/or split these explicitly merged objects, their agx::MergedBody has to be registered:

```
// Register 'mergedBody' to the handler. The objects continues to
// be merged after this call.
simulation->getMergeSplitHandler()->registerMergedBody( mergedBody );
```

Note: After an agx::MergedBody has been registered to the agxSDK::MergeSplitHandler, it's undefined behavior to add/remove edge interactions/bodies from the agx::MergedBody object.

40. Energy calculations

The energy of rigid bodies, constraints and motors can be reached using the energy manager of a simulation. Tutorials of using energy calculations with rigid bodies, constraints or drive train motors are located in **tutorials/agxOSG/tutorial_energyManager.cpp**.

```
agxSDK::EnergyManager* energyManager = simulation->getEnergyManager();
```

40.1. Rigid bodies

To be able to read the energy change of a rigid body it first has to be added to the **agxSDK::EnergyManager**.

```
energyManager->add( rigidBody );
```

Once added the energy manager will perform calculations to get

- The gravity potential change
- The dissipated energy from linear and angular velocity damping and external forces

Note

All external forces will be considered dissipative.

The energy changes during the last time step can be reached with the following code:

```
agx::Real potentialChange = energyManager->getPotentialChange( rigidBody );
agx::Real dissipation = energyManager->Dissipation( rigidBody );
```

It is possible to get both the change in kinetic energy and the total kinetic energy of a rigid body. The total kinetic energy can be reached with a static method and thus the rigid body does not have to be added to the energy manager for this.

```
// Get the kinetic energy change during the last time step.
agx::Real energy = energyManager->getKineticEnergyChange( rigidBody );
agx::Real energy = energyManager->getTranslationalEnergyChange( rigidBody );
agx::Real energy = energyManager->getRotationalEnergyChange( rigidBody );

// Get The total kinetic energy of a rigid body. The rigid body does not have to be added to the energy manager.
agx::Real energy = agxSDK::EnergyManager::getKineticEnergy( rigidBody );
agx::Real energy = agxSDK::EnergyManager::getTranslationalEnergy( rigidBody );
agx::Real energy = agxSDK::EnergyManager::getRotationalEnergy( rigidBody );
```

Attention

`getKineticEnergyChange`, `getPotentialChange` and `getDissipation` will return a change of energy during the last time step while `getKineticEnergy` will return the current energy.

40.2. Constraints

For constraints we get the change in potential energy and dissipation. The constraint first has to be added to the energy manager.

```
energyManager->add( constraint );
```

The energy change from the previous time step can be reached by the following calls.

```
agx::Real potentialChange = energyManager->getPotentialChange( constraint );
agx::Real dissipation = energyManager->getDissipation( constraint );
```

It is also possible to get dissipation, potential change and power from the secondary constraints. The `getPower` method is mainly used to get energy consumption from constraint motors such as **Motor1D** or **ElectricMotorController**.

```
energyManager->getPotentialChange( secondaryConstraint );
energyManager->getDissipation( secondaryConstraint );
```

```
energyManager->getPower( secondaryConstraint );
```

40.3. DriveTrain motors

For a motor it is possible to look att the power and the work done. A motor does not store any potential energy so all the work done by the motor is dissipative.

Energy calculations for drive train motors are supported for

- agxDriveTrain::ElectricMotor
- agxDriveTrain::Engine
- agxDriveTrain::FixedVelocityEngine
- agxDriveTrain::CombustionEngine

Add them to the **agx::DriveTrain::EnergyManager**.

```
agxDriveTrain::EnergyManager::add( motor );
```

The motor will be added to the energy manager of the simulation that the power line is added to. If the power line is not added to a simulation or if the motor has not been added to a power line it is also necessary to specify which simulation you want to use.

```
agxDriveTrain::EnergyManager::add( motor, simulation );
```

The power and work done can be accessed with the following methods.

```
agx::Real work = agxDriveTrain::EnergyManager::getWorkDone( motor );
agx::Real power = agxDriveTrain::EnergyManager::getPower( motor );
```

40.4. Known limitations

- All external forces on a rigid body are considered to be dissipative.
- The separation of conservative och dissipative work done by constraints works for
  - Soft elastic constraints
  - Stiff constraints with consistent initial conditions (non-violated and with a velocities tangent to the constraint)
- An added force on a drive train motor’s rigid body will also add to the work done and the power of that motor.

41. Statistics

AGX has several methods for measuring time and obtain statistics about timings in different parts of the simulation.

41.1. Timer

The agx::Timer can be used to measure the duration of the execution of compute-intensive code. The timer is based on hard-ware, using either the hardware's Time Stamp Counter or High Precision Event Timer (platform-dependent, please look at *include/agx/Timer.h* for details).

Timers can be started, stopped, (re-)started, or reset. They can be queried for their current time, which is given in milliseconds. The precision is between 100s of nanoseconds and microseconds (depending on hardware and operating system).

An example of the usage:

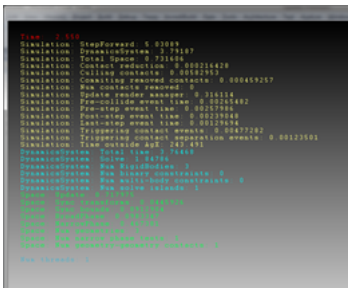
```
// Create a timer. It will started in the stopped (= "paused") state by default.
agx::Timer t;
//...do some thing else...
t.start();
// We want to measure this part of the code, call it section 1.
t.stop();
// We do NOT want to measure this part of the code, call it section 2.
t.start();
// Again, we want to measure this part of the code, call it section 3.
t.stop();
std::cout << "Section 1 and 3 together took " << t.getTime() << "ms.\n";
t.reset();
// Now, the timer is set to 0 again, and stopped. It can be used for other purposes now.
```

41.2. agx::Statistics

AGX has internal functionality for registering and reporting runtime statistics of a simulation. agx::Statistics is a singleton and can be accessed through a call to:

```
agx::Statistics::instance()
```

Various classes register into the statistics singleton and report both timing and other data related to statistics. The collection of data can be accessed in various ways. Through the debug rendering as overlay in the graphics of your application:



The statistics information presented in the graphics window is a subset from the complete set of data points stored. What is displayed in the graphics window can be changed in the file *settings.cfg* which should be located in <agx-dir>/data/cfg/settings.cfg.

To enable statistics rendering in the graphics window do the following calls:

```
agxSDK::SimulationRef sim = new agxSDK::Simulation;
agx::Statistics::instance()->setEnable( true );
sim->getRenderManager()->enableFlags(agxRender::RENDER_STATISTICS);
```

Or slightly more compressed:

```
sim->setEnableStatisticsRendering ( true );
```

If you want to get all the statistics information into a file, you can specify that through the simulation class:

```
agxSDK::SimulationRef sim = new agxSDK::Simulation;
sim->setStatisticsInterval( 1 ); // report every second

// Specifies a path to a file that will contain statistics info
sim->setStatisticsPath( "myStat.txt" );

sim->setEnableStatistics( true ); // Enable statistics reporting to file
```

This behavior can also be controller through environment variables read by the Simulation class when it is instanced. For more information see Section 47.

41.3. Statistics API

It is possible to readout the current statistics value after a call to Simulation::stepForward(). This is done through the singleton agx::Statistics::instance()

The C++ tutorial *tutorial_statistics.cpp* demonstrate how to access all the various statistics data that is available.

```
// Get the total time spent in AGX (and listeners)
agx::TimingInfo info =
  agx::Statistics::instance()->getTimingInfo("Simulation", "Step forward time");

// The last reported value
std::cerr << info.current;

// The accumulated value since start of the simulation
std::cerr << info.accumulated;
```

Statistics also collect other data, such as number of rigid bodies etc.:

```
agx::TimingInfo bodies =
  agx::Statistics::instance()->getTimingInfo("DynamicsSystem", "Num enabled rigid bodies"");
```

The available data can be queried by enabling statistics, running an application with AGX Dynamics and look at the generated statistics file (agx_statistics.txt):

```
cmd> set AGX_STATISTICS_ENABLE=1
cmd> set AGX_STATISTICS_INTERVAL=1
cmd> tutorial_constraints
```

Below is a snippet from that file. Red marks the name of the *module*, and Blue marks the name of the *data*. These strings can be used in the code listed above to receive the statistics data from a running simulation.

*** Time step 3.05 ***
Root
- Children:
Simulation
- Timings:
Inter-step time: 14.5213 (2608.82)
Step forward time: 2.3789 (351.901)
Dynamics-system time: 1.42312 (198.722)
Collision-detection time: 0.787439 (116.886)
Pre-collide event time: 0.0135979 (12.3506)
Pre-step event time: 0.0120962 (1.77996)
Triggering contact events: 0.0148825 (1.57511)
Post-step event time: 0.0105585 (1.14688)
Committing removed contacts: 0.00640043 (0.921192)
Culling contacts: 0.00576196 (0.619406)
Triggering contact separation events: 0.000594434 (0.115472)
Last-step event time: 0 (0.0924153)
Contact reduction: 0.000174818 (0.0256758)
- Data:
Num contacts removed: 0 (0)

42. Serialization

AGX Dynamics comes with a serialization system. This allows for a simulation to be stored on disk for later retrieval. Together with a callback system at restore, a stored simulation can be recreated together with custom references. This is very handy when debugging a system for example. If a problem occurs, a serialization of the simulation might be informative during debugging to locate any potential problems in the setup of the simulation. We recommend that a serialization is attached to reported tickets so that Algoryx can use that when analyzing the problem.

The serialization system will collect any references that a serialized object has. Meaning that serializing a rigid body will also store any geometries, shapes, materials that is associated to this rigid body. So it is not in general possible to do a partial store.

Serialization can be done to an ASCII (XML) file ".aagx", or a binary file ".agx". The ASCII format is not suitable for very large scenarios with huge height fields for example due to the size of the file. The binary file format is much faster and more compact and can in many case be written during runtime without a major effect on performance.

42.1. Universally unique identifier (UUID)

Each object derived from the base class **agxStream::Serializable** will inherit the property of a uuid. This is true for RigidBody, Geometry, Constraint, agxWire::Wire and many other classes.

When a simulation is stored in the serialization system, this uuid will be stored together will all the serialized objects. During restore, this uuid can be retrieved and used to identify objects. For an example, see 35.5.

42.2. Storing a simulation

To store an existing simulation there exists several options depending if you want to store it to memory or a file on disk. The simplest way is to use the writeFile function:

```
#include <agxIO/ReaderWriter.h>
// Create a simulation
agxSDK::SimulationRef simulation = new agxSDK::Simulation;
// Populate the simulation with bodies, geometries etc.
populateSimulation( simulation );
// Write to a file
if (!agxIO::writeFile("simulation.agx", simulation))
  error(...)
```

The file on disk will contain everything related to the simulation:

- Rigid bodies
- Geometries
- Shapes
- Constraints
- Wires
- GravityField
- Time step
- Current simulation time
- Collision groups
- …

The serialization will contain the current state of the whole simulation including velocities, tension, constraint violations etc. This means that restoring the simulation from the file will lead to an identical state as when it was stored.

Notice that restoring multiple serializations to a simulation *might* give you an unpredicted result. For example if one simulation disables two collision groups and another enables them, it will be the last restored simulation that dictates the result. The same goes for time step, gravity fields etc…

It is possible to filter out some of the data when performing a restore with the agxSDK::Simulation::ReadSelectionMask enum:

```
enum ReadSelectionMask {
  READ_NONE = 0x0,      ///<! Select to read none of the items below
  READ_TIMESTEP = 0x1,  ///<! Select to read and restore the TimeStep of the Simulation.
  READ_TIMESTAMP = 0x2, ///<! Select to read the TimeStamp (current time of the simulation)
  READ_GRAVITY = 0x4,   ///<! Select to read and restore the Gravity model of the simulation.
```

```
  READ_ALL = READ_TIMESTEP + READ_TIMESTAMP + READ_GRAVITY,
  READ_DEFAULT = READ_TIMESTEP /* + READ_TIMESTAMP */ + READ_GRAVITY
};
```

```
// Do not restore timestep, timestamp nor gravity.
agxIO::readFile("mystoredsimulation.agx", simulation, 0L, agxSDK::Simulation::READ_NONE);
```

Notice that the TIMESTAMP (current time) is not restored by default. So if you are restoring to an existing simulation which has been simulated for a while, the current time will not be affected.

42.3. Serialization to/from memory

To serialize a simulation to memory, first create a stream and then use the **Simulation::write method**:

```
// Create a memory stream, make sure it is in binary format to avoid NL/CR problems
std::stringstream os(std::ios_base::out | std::ios_base::binary);

// Now write the simulation to the stream. The second boolean argument determines if the
// serialization format should be binary or ASCII. Use ASCII for faster
// process and smaller memory footprint.
bool binaryFormat = true;
simulation->write(os, binaryFormat); // Serialization data is stored into the stream
```

To restore from memory, use an input stream and the Simulation::read method. In this case we copy data from the output stream to a new input stream, but it could just as easily been read from disk or any other location:

```
// Restore the simulation from memory
os.seekg(0);

// We need to get the data into a istream to be able to read from it
std::stringstream is(std::ios_base::in | std::ios_base::binary);
is.str(os.str());

// Now read from the binary stream.
simulation->read(is, binaryFormat);
```

42.4. Continuous serialization to disk

It is also possible to continuously serialize a simulation to disk during runtime. This might be helpful when debugging a problematic simulation. This will however have a substantial effect on performance, depending on the size of the simulation. For example, a large height field contains a lot of data per frame, hence it will severely slow down your simulation.

To enable this frame by frame simulation, you can use the API:

```
// Enable the serializer
simulation->getSerializer()->setEnable(true);
// Specify an interval for writing a frame
simulation->getSerializer()->setInterval(1);
simulation->getSerializer()->setFilename("frames.agx"); // 00001_frames.agx, 00002_frames.agx
```

Or change the **Enabled** value in the configuration file <agx-dir>/data/cfg/settings.cfg::

```
SimulationSerializer
{
  Enabled 1 // If ==1 then for each frame a file will be dumped with complete simulation content
  Mode 1 // When to write data: 0-PRE_COLLIDE, 1-PRE_STEP, 2-POST_STEP
  Interval 0.03333 // interval in seconds
  Filename "agx.agx" //.aagx for ascii, .agx for binary (faster/smaller)
}
```

With the settings above, a file named 00001_agx.agx, 00002_agx.agx will be written to disk with an interval of 33ms (30 Hz). These files can then be packed and set to Algoryx for further analysis regarding any issue that might have surfaced during the simulation.

42.5. Listening to restore events

It is possible to listen to a restore process and get callbacks when an object is being restored. This allows for custom connection between rigid bodies, geometries, constraints wires etc. and your own system.

In **tutorial_io.cpp** there is an example of how to associate rendering information for each restored geometry. Notice also that each stored object will have a uuid which is also restored. This can be used to store a reference in form of a text string (uuid, usually in the form of `"8d276495-19b6-44d8-84ba-5c1830ee97a4"`) which later during the restore can be used to uniquely identify a restored object.

So if one has a GUI system with representations of a rigid body, the GUI system could store the uuid of the rigid body in its own file storage system. When an AGX simulation is serialized to disk, this uuid is also stored together with the rigid body.

Later at restore, the GUI representation of a rigid body can be created and the reference to the rigid body can be established based on this uuid.

43. File formats

43.1. AGX Dynamics specific formats

This chapter explains the various file formats available for AGX Dynamics. All of the below listed file formats can be loaded with the application agxViewer.

Table 43.1 File formats

Class	Description
.agx	Binary serialization of an AGX Dynamics simulation
.aagx	XML serialization of an AGX Dynamics simulation
.agxPy	Python script using the AGX Dynamics Python scripting API
.agxPyz	Compressed archive including a .agxPy script file

43.1.1. .agx, .aagx

Chapter 42 Serialization describe the procedure for storing an existing AGX simulation to disk and how to later restore it. The file formats .agx and .aagx are binary and xml versions of these serializations.

43.1.2. .agxPy

When AGX is installed on Windows, the file format .agxPy (Python script) are associated with agxViewer.exe. In these file a function named buildScene() will be executed to build up the scene. For more information see chapter about Python scripting.

43.1.3. .agxPyz

The file format .agxPyz are zipped archives containing a Python script of the same name as the name of the archive. It can also contain textures/models and other data files required for the script to run. They can be loaded with the agxViewer application.

For example, the file ship.agxPyz *must* contain a script named: ship.agxPy in the root directory of the archive. This script will be executed by agxViewer when loaded:

```
> agxViewer ship.agxPy
```

Any data accessed by the script must be available in the archive (ship.agxPyz).

A standard zip command or the application agxArchive (see Section 44.2) can be used to create these archives.

43.1.4. .mpy

This file format is an Algoryx Momentum Python scripting file. The API contains a version controlled slightly modified subset of the full AGX Dynamics Python API. .mpy files are supposed to be *attached* to a simulation, as the Momentup scripting API does not allow for creating or modifying the structure of any AGX simulation. With this scripting interface you are limited to only modify constraint state (enabled, force-range, motors, locks etc.), change the sate of rigid bodies (such as moving, changing velocity etc.) and read simulation data such as constraint and contact forces/velocity/position. The API follows the semantics of the Momentum API, so many classes/methods have slightly different names compared to the full AGX Dynamics Python API.

An example of using a .mpy file:

```
import v1.Momentum

sim = v1.getSimulation()
hinge = sim.getHingeJoint("AHinge")

def OnStart(time):
    hinge.getMotor().setEnabled(True)
    hinge.getMotor().setTargetSpeed(0.1)

def OnStep(time):
    print(hinge.getMotor().getCurrentTorque())

def OnStop(time):
    pass
```

To use the above script, we save it to a file named `enableHinge.mpy` and attach it to an existing stored simulation (which should contain a hinge named ‘AHinge’):

```
> agxViewer saved_simulation.agx --attachScript enableHinge.mpy
```

43.1.5. .mpyz

Similar to .agxPyz, .mpyz is a compressed (zip) archive containing at least two files: a serialization (.agx) file with the simulation model, and one or more momentum Python scripting file (.mpy). The .agx file should have the same filename as the .mpyz file. This archive can contain more than one .mpy file, and they will all be attached to the simulation as in the example with the .mpy file.

```
> agxViewer mySimulation.mpyz
```

A standard zip command or the application agxArchive (see Section 44.2) can be used to create these archives.

43.2. 3D Mesh formats

AGX Dynamics is capable of reading a number of 3D mesh file formats. The table mesh file formats lists the supported file types.

44. 3D Viewer and default rendering

The tutorials and also the application **agxViewer** are all based on the class `agxOSG::ExampleApplication`. It is a class that is responsible for parsing command line arguments, reading files, creating a 3D window, stepping simulation and rendering the simulation. The 3D rendering is based on OpenSceneGraph.

44.1. agxViewer

agxViewer is an application for reading various files (.agx, .aagx, .agxScene, …). It is the default program for opening: .agx, .aagx, .agxPy, and agxPyz … files.

If you want to use agxViewer (and any other example/tutorial) from the command line/terminal, make sure you execute the **setup_env.bat** file in the installed agx directory from the cmd/terminal-instance. Otherwise it will not be able to find all the required dll-files and configuration files.

Help can be obtained with the argument:

cmd> **agxViewer –help**

For a list of available arguments, see Table 44.2 Arguments to agxViewer.

Sample use of agxViewer:

```
agxViewer --sceneFile template_scene1.agxScene --startPaused --stopAfter 3.1 --capture 1 --captureFPS 30 --saveScene
store.agx --saveAfter 3
```

The above example will:

- Load the scene file template_scene1.agxScene
- Start paused (press ‘e’ to continue)
- Exit the application after 3.1 seconds of simulation time
- Enable the capture of each frame as a .bmp file (agx__0001.bmp…)
- Capture 30 images/second (in simulation time to achieve real-time)
- 3.0 seconds into the simulation, the scene will be stored to disk with the name ‘**store.agx**’

For a list of available key bindings of the agxViewer application, see Table 44.1: Keyboard bindings.

44.1.1. 3D Rendering in Linux

To create an OpenGL context in Linux, GLX functions has traditionally been used to communicate with a X-server and create the context. This is also what OpenSceneGraph does when built to use OpenGL. To perform headless rendering, possible in the cloud or on a cluster, the requirement of having a X-server present to be able to use GLX can be unwanted.

It does not matter if it is a virtual framebuffer X-server that will not provide hardware acceleration or a real X-server configured to use one or more system GPUs and allow for not having a physical display connected. Neither of these two options is ideal for headless rendering.

Modern Linux distributions which have OpenGL libraries that support Vendor-Neutral-Dispatch (glvnd) usually also support EGL. With EGL it is possible to create an OpenGL Context without the requirement for having a X-server running. In such a setup, all of the libraries libGL, libEGL, libGLX, libOpenGL links to libGLdispatch which handles that there can be multiple drivers and maps API calls to the right driver instead of early days of OpenGL on Linux where drivers fought over which one should provide libGL.

EGL is supported in AGX packades for Ubuntu-18.04 and newer and CentOS 7.4 and newer where the base distribution have the required libraries and they are built in the right way.

To make use of headless rendering, the `--osgWindow 0` flag can be used and seamlessly `agxOSG::ExampleApplication` will use EGL in Linux to create a graphics context and provide OpenSceneGraph with said context. If EGL is not configured correctly on the machine or the EGL context creation does not succeed, OpenSceneGraph will try to create a context instead via GLX as fallback.

44.1.1.1. Possible EGL troubleshooting in Linux

1. Make sure your distribution has glvnd support.
2. Test that system provided utils can use EGL, e.g. `eglinfo`.
3. Check that EGL Installable Client Driver (ICD) files are placed correctly, normally in `/usr/share/glvnd/egl_vendor.d` and that the correct vendor EGL libraries are referenced. Normally done automatically but can be needed when using Docker.
4. Make sure Linux kernel graphics driver and corresponding vendor OpenGL libraries have matching version number.

44.1.2. ImageCapture

The agxViewer example above demonstrates how to capture images to file. The class used for this is `agxOSG::ImageCapture`.

```
agxOSG::ImageCapture* myCapture = application->getImageCapture();
```

ImageCapture has methods for

- changing the file format (setPostfix)
- changing the image logging frequency(setFps)
- turning logging on or off
- changing the image destination(setDirectoryPath)

The last one is important if the executable (agxViewer, your tutorial, …) is stored in a folder where the default user lacks writing access – in this case, logging the images would fail and a different location for the images should be given.

44.1.3. VideoCapture

ExampleApplication can capture video from the simulation using FFMPEG if a graphics window is present. To start video capturing from the agxViewer command line, the following, arguments can be used:

- `--captureVideo 1` - Starts the video capture when the simulation starts.
- `--captureFPS <float>` - Decides the number of frames that should be captured per second.
- `--videoFPS <float>` - Decides the FPS of the created video.
- `--videoName <str>` - Name of file, or path and name of capture video file.

The video will be saved as `agx_mov.mp4` in the working directory unless `--videoName <str>` is used.

Video capture can also be started using the API:

```
app->setupVideoCaptureRenderTotexture();
app->setAllowWindowResizing(False);
auto* vc = app.getVideoServerCapture();
vc->getEnableSyncWithSimulation();  // Synchronize the capture with the simulation time
vc->startCapture();
```

where `app` is an instance of the `agxOSG::ExampleApplication`. The video capture can be paused and resumed using

```
vc->stopCapture();  // Pause
// Do something
vc->startCapture();  // Resume
```

The video capture process (FFMPEG) can be stopped, allowing a new video file to be created:

```
vc->stopProcess();
vc->setFilename("video_file_2");  // Change the file name to avoid overwriting the previous file
vc->startProcess();  // Start recording a new video
```

The capture and video FPS can be changed as well:

```
vc->setImageCaptureFPS(60);
vc->setVideoFPS(20);
```

The capture FPS specifies the number of frames that will be captured each second, and the video FPS decides the playback speed of the video. A real time factor can be calculated by dividing the capture FPS with the video FPS, where a number less than 1 yields a slow motion video, and a number greater than 1 produces a timelapse.

See the AGX Dynamics API Documentation for the `agxOSG::VideoFFMPEGPipeCapture` class for more information on configuring the video capture.

44.1.4. Key bindings

Table 44.1 Keyboard bindings

Key	Description
Numeric keys, function keys	Select scene
a	not used (reserved for use in specific scenes)
b	Toggle statistics reporting to screen (debug rendering)
c	contact reduction (trimesh) circles between binning3, off, binning2
d	contact reduction circles between off, binning2, binning3
e	Toggle simulation run/pause
f	Toggle full screen
g	Toggle debug rendering
h	move camera to the left
i + left mouse	information about selected body
j	move camera backwards
k	move camera to the right
l	move camera downwards
m + left mouse	camera follows selected body
m	reserved for memory debugging
n	Toggle statistics reporting to screen (debug rendering) and console
o	move camera upwards
p	Toggle merge/split functionality
q	Toggle real time synchronization for image capture
r	In paused simulation, take a single time step
s	Toggle OSG statistics
t	Toggle synchronization of image capture with wall/simulation time
u	Move camera upwards
v	gravity points towards screen down
w	circle between osg shading modes (polygon, wire frame, points)
x + left mouse	remove selected body
left alt + x	move all non-static rigid bodies to the left
y	Toggle update debug render cache for simulation in pause mode
z	not used (reserved for use in specific scenes)
A	Toggle debug rendering of AABB trees
B	Toggle debug rendering of bounding volumes (spheres or boxes depending on broad phase algorithm)
C	Print camera data to console
D	Delete all objects from simulation
E	Circle through solvers
F	Toggle text debug rendering
G	Toggle OSG rendering
H	Print help screen to console
I	restore simulation from scene.agx
J	Update collision detection
K	Toggle debug coloring mode
L	Toggle osg lightning
M	reserved for memory debugger

Key	Description
N	Reload scene
O	save simulation to scene.agx
P	run parallel performance test
Q	Toggle wait in calling update (real-time sync)
R	Re-read specified configuration file
S	Toggle osg on screen stats
T	Toggle enable scene decorator
U	dump screenshot for povray
V	gravity points in original direction
W	save simulation to saved_scene.scene
X	Toggle simulation dump
left alt + X	move all non-static rigid bodies to the right
Y	Toggle auto sleep of resting objects
Z	Dump scene to osg-file
+	Select next scene
–	Select previous scene
return	Shoot sphere in camera direction
drag left mouse	rotate camera
drag mid mouse (press wheel)	pan camera
mouse wheel up/down	adjust camera movement speed
left ctrl + drag left mouse	add temporary ball joint to selected body
left ctrl + drag mid mouse	add temporary lock joint to selected body
left ctrl + drag right mouse	add temporary lock joint resetting rotation to selected body
left alt + drag left mouse	move selected body/geometry (no dynamics)
left alt + drag right mouse	rotate selected body/geometry (no dynamics)
space	Reset camera to view all objects
esc	Quit
Left	Turn batch debug rendering off
Right	Turn batch debug rendering on
Up	Increase debug rendering scale
Down	Decrease debug rendering scale
.	Toggle shader state
F10	Toggle video capture on/off
F12	Toggle image capture on/off

44.1.5. Arguments

Table 44.2 Arguments to agxViewer

Argument	Explanation
`--addBundlePath <path>`	list of paths to append to OpenPLX bundle dependencies if any
`--agxOnly  (shorthand: -a)`	Only step AGX, no graphics
`--allowExternal`	Enable external remote command connections
`--attachScript <filename>`	Specified a Python file that should be executed after a scene is loaded. Can be specified multiple times for multiple scripts

Argument	Explanation
--broadPhase <grid | sweepAndPrune>	Specifies the broad phase algorithm. Default is sweep and prune
--calculateResiduals	Switch on calculation of residuals during solver iterations
--capture <1 | 0>	Enable | disable capturing of screens to disk
--captureDirectory <directory>	Directory where captured frames are stored
--captureFPS <float>	Specify the desired capture rate for image capture
--captureFiletype <fileType>	File type of captured frames, eg 'png' (default is 'bmp')
--captureVideo <1 | 0>	Enable | disable ffmpeg capture of video to disk
--cfg <cfg-file>	Read in a configuration file
--clearColor r g b	Specifies the clear color for OpenGL
--clientQueue <1..n>	Specifies the maximum queue of simulation frames that the remote client will queue up before purging the queue. Default is 1
--coSimulationInputFile <filename>	File name for co simulation input data
--coSimulationOutputFile <filename>	File name for co simulation output data
--coSimulationServer	Start as a controllable co-simulation server
--connect <hostname>(:port)	Connect to a remote server
--contactReducer <0,1,2,3>	Specify the default contact reduction bin size between bodies, 0-disabled, 1-3 binResolution
--cycleThroughScenes	Opens the next scene once the old one has been closed
--disableArchiveDate	Do not log current execution date output archive, but rather AGX build date. Mostly for unittests
--disableUuid	Do not generate unique uuid for each new Serializable object. Mostly for unittests
--dumpEnable	Enable dumping of simulation to disk
--dumpInterval	Specify the interval (in seconds) between each dump of simulation to disk
--exportHdf5 <filename>	Export to a hdf5 file.
--exportSolverData	Export solver data to HDF5 archive
--forceSilent	Force redirect stdout/stderr to null device
--granularContactMode <2 | 3 | 4>	different granular contact modes (for internal usage).
--gravity x y z	Specify the gravity vector
--gravityEnabled <1 | 0>	0 will disable gravity, 1 will enable it (default)
--groundPlane <heightOverZero>	Create a plane with default material at specified height in Z
--help (shorthand: -h)	Prints help screen with command line arguments; exits
--interactiveRemoteClient	Use when interactiveRemoteClient will connect to AGX
--journal32bit	Force 32bit floating point data in journal
--journalAppend	Appends the recording in a specfied journal if the application opens an .agxJournal file.
--journalConfiguration <filename>	When recording a journal session, this is the file listing the buffers and values to record
--journalDisableSceneLoading	Load the journal without initializing the simulation
--journalFormat <HDF5 | CUSTOM>	The journal format
--journalFrequency <frequency>	The frequency of journal frames
--journalIncrementalStructure	Enables incremental journal recording. Important when simulation structure is changed during journal recording
--journalPlayback <filename>	Initiates playback from a previously recorded journal. See –sessionName
--journalPlaybackEofMode <LOOP | PAUSE | SIMULATE | RECORD>	What do do when journal EOF is reached. LOOP: Loop from start point, PAUSE: Pause playback, SIMULATE: Switch to simulation from current state, RECORD: Same as SIMULATE but also record new simulation frames to current journal session

Argument	Explanation
`--journalPlaybackStartAt <time | END>`	Where to start journal playback
`--journalPlaybackStopAt <time | END>`	Where to stop journal playback
`--journalRecord`	Initiates recording to a journal. See –sessionName and –journalRecordPath
`--journalRecordPath <filename>`	When recording a journal session, this is the journal in which the session will be stored
`--journalRecordStartAt <time>`	When recording a journal session, this is the time when the recording will start
`--journalStep <timeStep>`	Inverse frequency of journal frames
`--library <library name>`	Pre-load an AGX Dynamics module library(.dll / .so) to enable deserialization of features not part of agxCore / agxPhysics
`--listSessions <filename>`	List the sessions stored in the given journal
`--mcp <BLOCK | KELLER | HYBRID>`	Specify which MCP algorithm to use
`--modelName <modelname>`	the name of the model to load when loading a .openplx file.
`--multiSamples <0..n>`	Number of multi samples for anti aliasing (0 for off; default: 4)
`--nlSolveDataH5 <filename>`	Export non-linear solve data to HDF5 archive
`--numImpactIterations <1..n>`	Number of iterations during impact stage 1
`--numIterations <1..n>`	Number of iterations during normal solve (i.e., no impacts) **This will override file any previous read value specified by file or script**
`--numIterationsForIterateLast <1..n>`	Number of iterations performed on special constraints,
`--numLCPIterations <1..n>`	Max number of iterations for LCP solver
`--numPPGSIterations <1..n>`	Number of iterations during the Parallel-Projected-Gauss-Seidel solve, if enabled **This will override file any previous read value specified by file or script**
`--numPostImpactIterations <1..n>`	Number of iterations during impact stage 2 (can be less than for impact stage 1)
`--numThreads <n>`	Specify number of threads available for AGX
`--openplxDebugRenderFrames`	enable debug rendering of frames when loading a .openplx file.
`--enableOpenPlxDebugLogs`	Enable OpenPLX debug logs.
`--osgWindow <1 | 0>`	Render graphics, but optionally hidden window, useful for offline render to texture targets
`--particleRenderMode <0 | 1 | 2>`	set particle render mode < SPRITES(0) | QUAD_SPRITES(1) | TRIANGLES(2) >
`--pauseAfter <float>`	Will pause running simulation after this time stamp
`--portFile <filename>`	Print open ports to a file
`--preIntegratePositions <1 | 0>`	0 will integrate positions at end of timestep, 1 will integrate at beginning of timestep.
`--profileFrequency <Hz>`	Frequency of profiling, both thread timeline and task timings
`--quickProfiling`	Does a quick profiling (to analyze performance questions)
`--quickProfilingFile <filename>`	Send quick profiling output to this file instead of standard output
`--quickProfilingNumTasks <integer>`	The number of tasks to include in the quick profiling
`--rcs`	Start remote command server (default port 5656, modify with –rcsPort)
`--rcsPort <n>`	Choose port for remote command server, default 5656, 0 will automatically find an available port
`--realTime <1 | 0>`	Sync stepping simulation to real clock time (1) or run as fast as possible (0). Default: 1
`--renderDebug <1 | 0>`	Specify whether rendering using agx Debug renderer is enabled or not
`--renderGrid`	Turns rendering of (hierarchical) grid cells on
`--renderOsg <1 | 0>`	Specify whether rendering using OSG primitives is enabled or not
`--renderProxyShadows <1 | 0>`	If 1 geometries rendered by RenderProxys cast and recieves shadows. Default is 0
`--resetTimersAtTime <time>`	Reset task profiling accumulation timers at the given time
`--save <filename>`	Create scene and write to specified filename

Argument	Explanation
`--saveAfter <X seconds>`	Step forward system until we read X seconds in simulation time then the content of the scene will be stored to disk
`--saveScene <filename>`	When save scene button is pressed, this is the file where the scene will be stored
`--scene <n>`	In a simulation with several scenes, load the n-th one (1…n)
`--sceneFile <path>`	Path to a .scene file that will be read
`--server`	Start a remote debugger server
`--serverCompress <0/1>`	Specify whether compression of data for server will be used or not.
`--serverPort <port>`	Specify a socket port which will be used for listening for connection from a remote client.
`--serverTimeStep <timeStep>`	Specify the rate by which data is sent to the remote listener. Should be a multiple of the simulation time step.
`--serverWait <0 | 1>`	Specify whether the server should wait for the client to ask for more data before sending a frame
`--sessionName`	When recording or playing a journal, this is the session that will be written to or read from
`--shadowMethod <0,1,2>`	Select shadow method
`--solverDataGroup <group name>`	Group in the HDF5 archive to store solver data into. A counter will be added to separate different runs with the same group name
`--solverDataPath <filename>`	HDF5 archive to store soler data into
`--solver <MULTI | ITERATIVE>`	Specify which solver to use
`--startPaused (shorthand: -p)`	Start the simulation in paused mode
`--stopAfter <X seconds> (shorthand: -s <X>)`	Step forward system X seconds in simulation time, relative to the initial simulation time
`--stopAfterFrame <X steps>`	Step forward system until we reach X simulation steps
`--stopAt <X seconds>`	Step forward system until we read X seconds in simulation time
`--synchronize <1 | 0>`	Synchronize image capture with Simulation time
`--targetFPS <float>`	Specify number of graphic window updates per second. Defaults to 60.
`--taskProfile`	Enable export of task timing data
`--terrainJournalRecord`	Enables journal recording of agxTerrain height data. NOTE: compaction is not recorded.
`--threadTimeline <format>`	Enable thread timeline profile. Format can be either ‘svg’ or ‘chrome’
`--timeStep <float>`	Specify the time step (dt) used for stepping the simulation
`--translate x y z`	Translate the debug rendering of the scene
`--twoStagesImpacts <1 | 0>`	Enable two stages impact (might be slow) default: 0
`--unittest (shorthand: -u)`	Run all unit tests in file
`--unittestPassOnExceptions`	Unit tests will not catch exceptions, but pass them on
`--use32bitGranularBodySolver <0 | 1>`	Specify if the Real32 granular body solver should be used
`--useClick`	Use click with OpenPLX
`--useClickAddr <address>`	The address to use, default is tcp://*:5555
`--useContactWarmstarting <0 | 1>`	Toggles if warmstarting should be used for contacts with direction friction. Default 0
`--useGranularWarmStarting <0 | 1>`	Specify if warm starting for the granular GS solvers should be used
`--useIndexSetWarmStarting <0 | 1>`	Toggles if indexset should be warmstarted. Default 1
`--useParallelPGS <0 | 1>`	Specify if the ParallelPGS solver should be used
`--useShaders`	Use the shader based render state
`--useShadows`	Enable shadowing
`--vSync <1 | 0>`	Enable | Disable vSync
`--version (shorthand: -v)`	Prints version number, exits

Argument	Explanation
`--videoFPS <float>`	Specify the desired fps for videos created using ffmpeg.
`--videoName`	set video name, default is 'agx_mov.mp4'
`--window width height`	Specifies the size of the rendering window in pixels
`--windowTitle title`	Specifies the title of the rendering window. Default title is 'AGX Dynamics'

In order to get an overview over all command line arguments, use the *–help* switch: `agxViewer --help`

44.2. agxArchive

agxArchive is an utility that can be used to convert between .agx and .aagx file, investigate a serialization file and creating a script archive.

44.2.1. Command line arguments

Table 44.3 Arguments to agxArchive

Argument	Explanation
`--help/-h`	Show help
`--disableUuid`	Disable UUID generation for deterministic output
`--generate/-g`	Generate a random UUID string
`--library <library name>`	Pre-load an AGX Dynamics module library (.dll/.so) to enable deserialization of features not part of agxCore/agxPhysics
`--pack/p`	Create a compressed script archive
`--unpack/-u`	Decompress a script archive to disk

44.3. Usage

Investigate a serialization

The general use of agxArchive is to investigate a serialized file:

```
> agxArchive submarine.agx

Archive information
-------------------
Header               Algoryx:AgX Library 64Bit
AGX version          : 2.12.0.0-23170
Serialization version: 37
BuildDate            : Mar 30 2015
BuildTime            : 17:54:35
File format          : binary
File size            : 216601 bytes
Serialization date   :
Build flags
     USE_PARTICLE_SYSTEM=1
     USE_OSG=1
     USE_OPENCL=0
     SABRE_USE_SSE3=1
     UNITTEST_ENABLED=0
     SABRE_USE_PADDING=1
     USE_SSE=1
     USE_64BIT_ARCHITECTURE=1
     USE_REAL_FLOAT=0
```

44.3.1. Convert files

By supplying two files (.agx/.aagx) one can convert between these two file formats (ascii/binary) for serialization.

```
> agxArchive submarine.agx output.aagx
Reading input file: submarine.agx
```

```
Writing output file: output.aagx
```

44.3.2. Pack a script archive

If you have a Python script file together with some data, you can use agxArchive to pack these files into a compressed archive. Directories will recursively be compressed into the archive.

Usage: agxArchive <archive-name> files and directories

```
> agxarchive --pack submarine.agxPyz submarine.agxPy submarine.agx
submarine.agxPy
submarine.agx

Successfully archived 2 files to archive 'submarine.agxPyz'
```

44.3.3. Unpack a script archive

Usage: agxArchive <archive-name> <target directory>

```
> agxArchive --unpack submarine.agxPyz myScripts
Successfully decompressed archive 'submarine.agxPyz'' into directory 'myScripts'
```

44.3.4. Pre-loading a library

agxArchive is not linked against all AGX Dynamics runtime libraries. Therefore it can be necessary to explicitly load some libraries to enable de-serialization of specific features.

For example, objects in the namespace `agxDrivetrain::` are part of the agxModel library. Reading an archive containing such objects will fail because the agxModel libary will not be loaded automatically. In some cases, the correct library can be derived from the namespace. For example, an object in the agxModel namespace will automatically be searched for in the agxModel dynamic library. To explicitly load a module, the –library argument can be used:

Usage: agxArchive <archive-name> –library name1 –library name2

```
> agxArchive myModel.agx --library agxModel --library agxTerrain
```

44.3.5. Increasing performance for high frequency simulations with graphics

By default the number of graphics window updates per second is set to 60 frames per second (fps) regardless of the simulation timestep. This will throttle graphic updates to 60 frames per second (fps), while the simulation still runs at full speed. When using high frequency simulations, ie > 500Hz (timestep < 0.002 seconds) this has a severe impact on the simulation performance. A target fps higher than 60 is unnecessary for most applications, but you can change it using `--targetFPS <float>`, or disable throttling altogether with `--targetFPS 0`

When stepping manually through a simulation, each simulation step will always update the graphics window.

45. Journal Viewer

Note

The Journal viewer is only available under Windows operating system.

Journal Viewer is a Graphical User Interface (GUI) that allows you to play and post-process **journals**, as shown in Fig. 45.1.

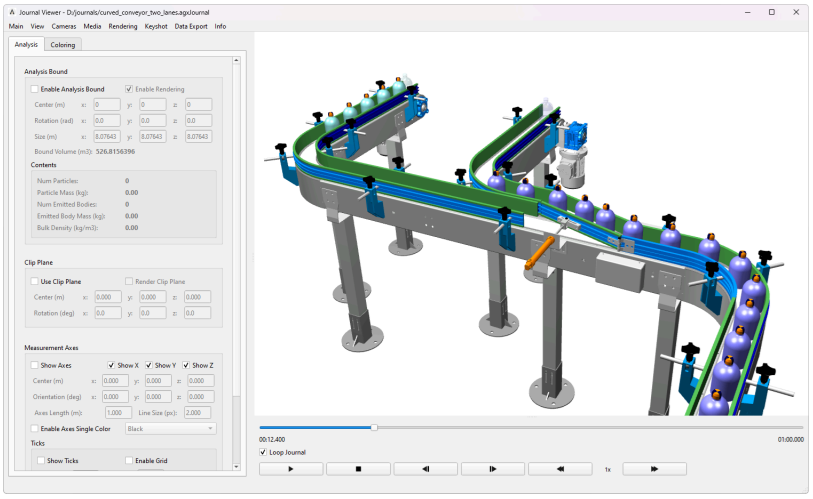


Fig. 45.1 The interface of the Journal Viewer. A journal from a simulation of a curved two-lane conveyor transporting bottles is loaded.

In AGX Dynamics (and Algoryx Momentum), simulation data can be stored in an `.agxJournal` file. This simulation data file, known as a **journal**, is a recording of the simulation.

By playing the journal with the Journal Viewer, the saved simulation is displayed similarly to a video, but unlike a traditional video that consists of image frames, this playback uses simulation data, such as object positions, velocities, and other properties. This allows for dynamic viewing from any angle, zooming in or out, and offers post-processing capabilities such as coloring particles based on scalar quantities (e.g., velocity), and much more. Additionally, custom scripts can be appended for tailored post-processing.

In this section, we will describe how to find and use Journal Viewer, including how to load and play a journal, as well as its post-processing features. First, we will describe the key functions and features. Following this, we will detail the specific menu items, explaining how each one contributes to the overall functionality. Lastly, we'll describe how to create and use custom scripts for tailored post-processing.

45.1. Save Journal

In AGX Dynamics (and Algoryx Momentum), simulation data can be stored in an `.agxJournal`, which can later be played using the Journal Viewer. This file, referred to as a **journal**, contains a detailed recording of the simulation, including various data points (object positions, velocities, etc.). To save a journal using AGX Dynamics, you can use *agxViewer* (see agxViewer). To save a journal, open your terminal, set up your AGX environment, and run the following command:

```
>> agxviewer <agxScenePath> --journalRecord
```

where `<agxScenePath>` is the path to the AGX scene file, which can be either relative to the current directory or an absolute path. This saves the journal to the default journal filename `Journal.agxJournal`.

For example, to save a journal for the scene located at `D:/agx_scenes/excavator_E85_terrain.agxPy`, run this command from any directory:

```
>> agxviewer D:/agx_scenes/excavator_E85_terrain.agxPy --journalRecord
```

Alternatively, if you are already in the directory `D:/agx_scenes/`, you can simplify the command:

```
>> agxviewer excavator_E85_terrain.agxPy --journalRecord
```

To save the journal to a custom path, either relative to the current directory or an absolute path, use the argument `--journalRecordPath` followed by the custom path. For example:

```
>> agxviewer excavator_E85_terrain.agxPy --journalRecord --journalRecordPath excavator_E85_terrain.agxJournal
```

To explore more options related to journal recording, use the following command:

```
>> agxviewer --help
```

45.2. Launch Journal Viewer

The Journal Viewer is located in `.../bin/ARCH/`, where `ARCH` corresponds to your system's architecture. For example, if you are using Windows with a 64-bit architecture, the location would be `.../bin/x64/journalViewer.exe`.

Additionally, the Journal Viewer can be accessed from the terminal or shell from any folder, similar to how *agxViewer* is accessed. To do this, open your terminal, set up you AGX environment, type `journalViewer`, and press Enter. The interface of the Journal Viewer will appear, as shown in Fig. 45.2.

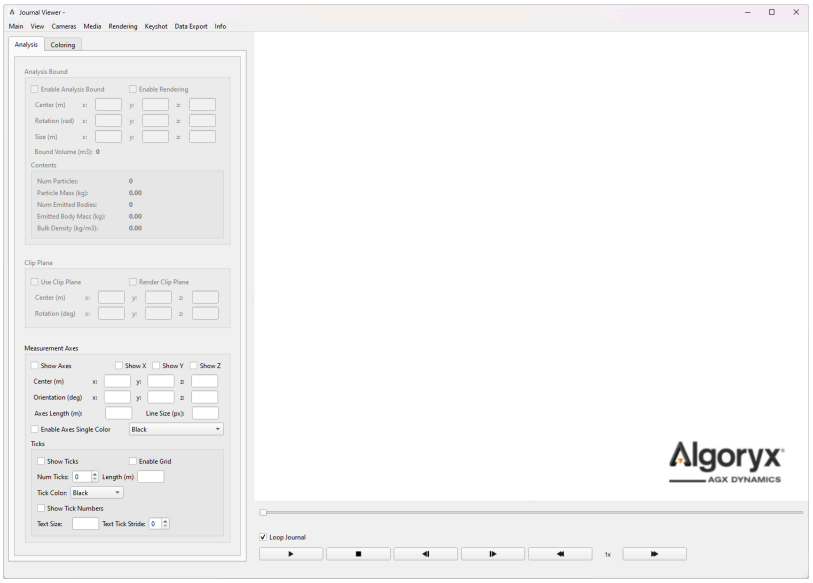


Fig. 45.2 The interface of the Journal Viewer with no journal loaded.

To launch the Journal Viewer and simultanously load a journal, refer to Load journal.

45.3. Load journal

To load a journal, navigate to the main menu (*Main* > *Open File*) and select the desired journal file. Once a journal is loaded, the interface will display the simulation, as shown in Fig. 45.3.

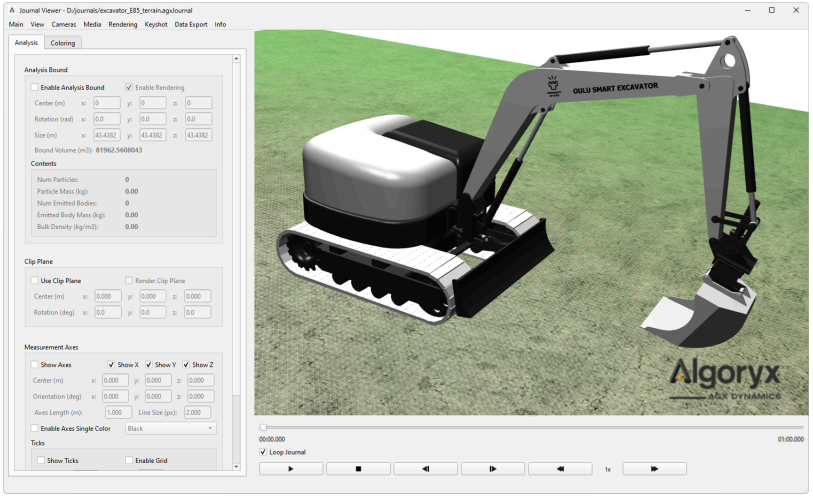


Fig. 45.3 A loaded journal of a scene with an E85 excavator.

You can also launch the Journal Viewer and simultaneously load a journal directly from the terminal. To do this, use the following command:

```
>> journalviewer <journalPath>
```

where `<journalPath>` is the path to the journal, which can be either relative to the current directory or an absolute path.

For example, to load the journal located at `D:/journals/excavator_E85_terrain.agxJournal`, you would run the following command from any directory:

```
>> journalviewer D:/journals/excavator_E85_terrain.agxJournal
```

Alternatively, if you are already in the directory `D:/journals/`, you can simply use:

```
>> journalviewer excavator_E85_terrain.agxJournal
```

45.4. Playback

The playback panel of the Journal Viewer, see Fig. 45.4, functions similarly to the controls of a traditional video player.

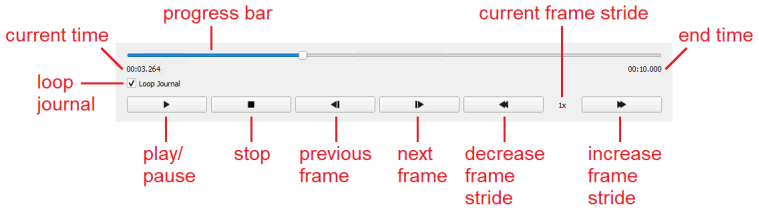


Fig. 45.4 The playback panel.

Table 45.1 describes the buttons and items found on the playback panel.

Table 45.1 Playback Panel Items

ITEM	DESCRIPTION
Play/Pause	Starts the playback of the journal. If playback is active, the play button changes to a pause button.
Stop	Stops playback and returns to the beginning of the journal.
Previous Frame	Steps backward by N simulation frames, where N is the current frame stride.
Next Frame	Steps forward by N simulation frames.
Decrease Frame Stride	Reduces the frame stride. Frame stride refers to the number of simulation frames skipped at each playback step. For example, if the frame stride is set to 4, every 4th frame is displayed during playback.
Increase Frame Stride	Increases the frame stride.
Current Frame Stride	Displays the current frame stride value.
Current Time	Shows the current playback time.
End Time	Displays the total duration of the journal.
Loop Journal	When enabled, the journal replays continuously after reaching the end.

45.5. Adjusting the viewing window

The viewing window is the area where the simulation and its objects are displayed. Users can adjust the view using the mouse, keyboard, or specific settings. In this section, we describe the available mouse and keyboard commands for interacting with the viewing window.

Table 45.2 lists the mouse commands for manipulating the viewing window.

Table 45.2 Mouse commands for the viewing window.

MOUSE BUTTON	ACTION
Left	Zoom in or out by moving the mouse up or down
Scroll wheel	Pan the view horizontally or vertically by moving the mouse in any direction.
Right	Rotate the object by moving the cursor in any direction.

Table 45.3 lists the keyboard commands for manipulating the viewing window.

Table 45.3 Keyboard commands for the viewing window.

KEY	ACTION
space	Reset and center view to display all objects.
b	Draw statistics (time, timestep, etc.).

KEY	ACTION
+	Increment the clip plane position in the direction of the clip plane normal.
-	Decrement the clip plane position in the direction of the clip plane normal.
p	Toggle the left side docket (containing Analysis and Coloring).
Ctrl+1 to Ctrl+6	Set the view to the positive/negative direction of a coordinate axis. 1: X, 2: -X, 3: Y, 4: -Y, 5: Z, 6: -Z.
Ctrl+7	Set the view to the normal direction of the clip plane.
Ctrl++	Zoom in with Orthographic View (if orthographic mode is enabled).
Ctrl+-	Zoom out with Orthographic View (if orthographic mode is enabled).
J	Open a Camera File.
K	Save Cameras. The currently stored camera views are saved to an `.cfg` file.
H	Show Views. Opens a panel with the currently stored camera views.
M	Store Current Camera.
PgUp	Next Camera. Selects the next camera view stored.
PgDown	Previous Camera. Selects the previous camera view stored.
G	Debug Rendering.
O	Standard Rendering.
Home	Mesh Rendering > Collision Mesh.
Ins	Mesh Rendering > Render Mesh
F12	Take screenshot. The image is saved to the directory of the journal.
e	Plays/pauses playback.
l	Sets the light source in the direction of the current view.

45.6. Cameras

The **Cameras** menu allows you to store different camera views, either temporarily or permanently in a file, making it easy to switch between saved views. To save a camera view, first adjust the viewing window to your desired position (refer to Adjusting the viewing window for details). Then, select **Store Current Camera** (shortcut: *M*) to add the current view to an internal list. You can repeat this process by adjusting the viewing window and selecting Store Current Camera again to save multiple views.

To view the list of saved camera views, select **Show Views**, which will display the list on the right side of the screen. To navigate between stored views, use **Next Camera** (*PgUp*) to cycle forward through the list or **Previous Camera** (*PgDown*) to go back.

To save your list of camera views to a file, select Save Cameras (shortcut: *K*). This will open a file dialog where you can save the views as a `.cfg` file in a folder of your choice. To load a previously saved camera views file, select **Open Camera File** (shortcut: *J*) and choose the desired file.

Note

If you save a camera file in the same folder as a journal with the same name as the journal (except for the file extension), then, by loading the journal, the camera file will automatically be loaded.

45.7. Media

The **Media** menu offers options to save a video file of the journal and capture screenshots.

To save a video, start by loading the desired journal and adjusting the size of the viewing window. Keep in mind that the window size directly affects the video's resolution; for instance, a full-screen viewer will result in a larger video than a smaller window. Next, select **Video Generation Settings** (shortcut: *V*), which opens the window shown in Fig. 45.5.

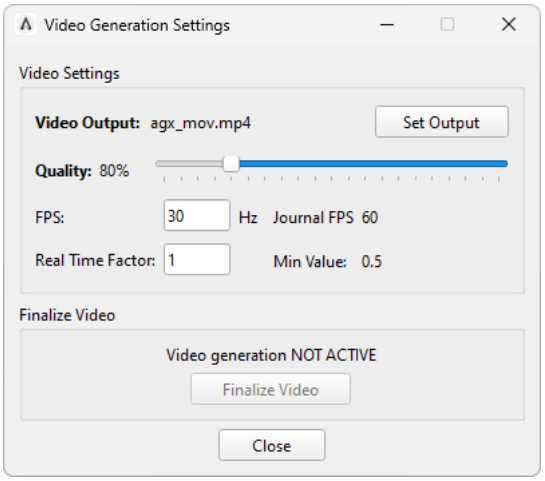


Fig. 45.5 The Video Generation Settings window.

In the settings window, start by clicking **Set Output** to choose the filename and directory for the video. You can adjust the video quality using the quality slider. **FPS** determines the Frames Per Second of the video. The journal's current FPS is displayed to the right. You'll also set the **Real Time Factor** (RTF), which controls the playback speed relative to real time:

- **RTF = 1**. The video will play in real time. A 10-second simulation will result in a 10-second video.
- **RTF > 1**. The video will play faster than real time. For example, with an RTF of 2, a 10-second simulation will produce a 5-second video.
- **RTF < 1**. The video will play slower than real time. For instance, with an RTF of 0.5, a 10-second simulation will produce a 20-second video.

Once the settings are configured, select **Close**.

To begin recording, ensure the journal is paused, then select **Capture Video** (shortcut: *C*) from the Media menu. A red "Recording Video" message will appear above the playback panel. As the simulation progresses (either by playing or advancing frame by frame), the content shown in the main window will be recorded. You can pause and resume recording as needed, and adjust the camera view during the recording. Be careful not to use the Stop button, as this will reset the playback. Once you have recorded the desired content, pause the video, return to **Video Generation Settings**, and select **Finalize Video** to save the recording.

To take a screenshot, select **Take Screenshot** (shortcut: *F12*). This captures an image of the current frame and saves it to the journal's directory. Similar to video generation, the screenshot will reflect the current size of the viewing window.

45.8. Analysis (tab)

The Analysis tab located in the left side docket is shown in Fig. 45.6. It provides a range of features designed to enhance the visualization and exploration of scenes, and carry out measurements and analyzes. It allows users to isolate specific regions, manipulate views, and obtain detailed data about objects or areas of interest. The features are described in the section below.

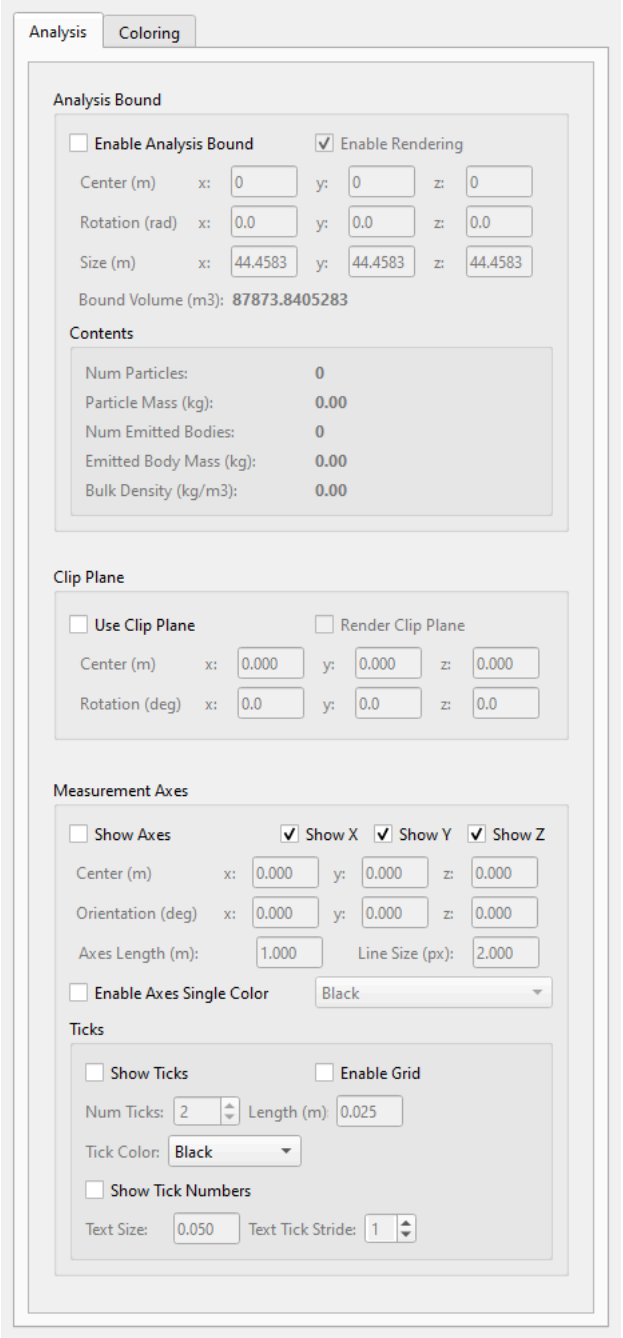


Fig. 45.6 The analysis tab.

45.8.1. Analysis Bound

The Analysis Bound function is available within the Analysis tab, as shown in Fig. 45.7.

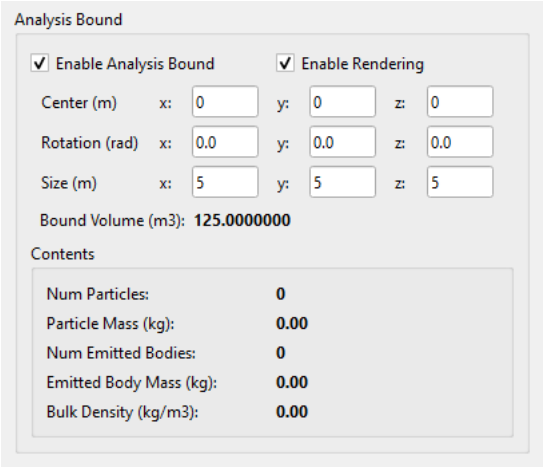
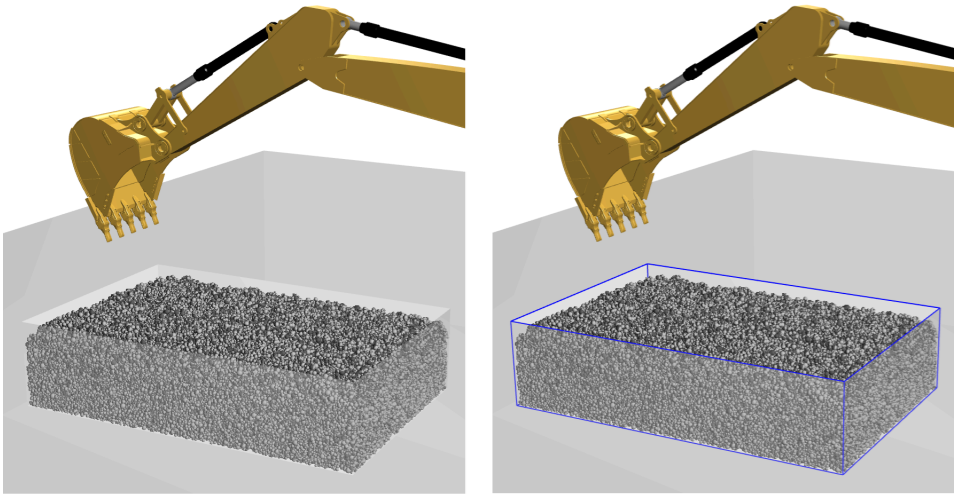


Fig. 45.7 The Analysis Bound Tab.

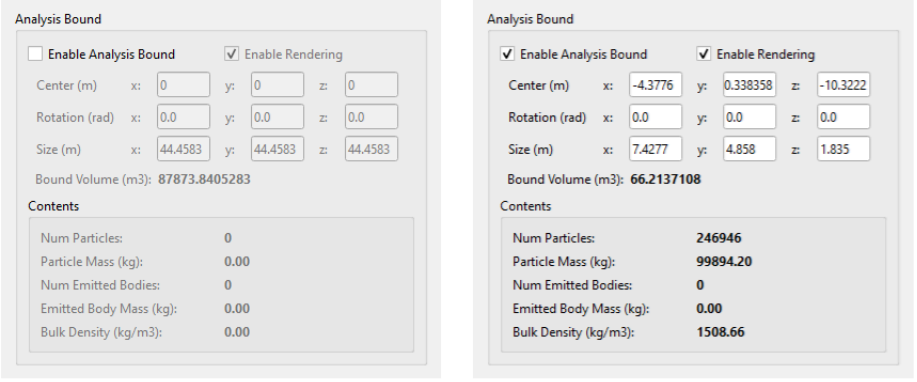
This feature allows users to define a rectangular region in the scene with customizable size, position, and orientation. **Enable Analysis Bound** activates or deactivates this region, while **Enable Rendering** toggles the visual display of the Analysis Bound, shown as a blue rectangle. Once enabled, statistical data for the defined region are displayed in the **Contents** panel. An example with and without the Analysis Bound is illustrated in Fig. 45.8.



(a) (b)

Fig. 45.8 A scene with a CAT365 excavator about to dig. (a) No analysis bound enabled. (b) Analysis bound enabled.

The settings and data for Fig. 45.8 are shown in Fig. 45.9.



(a) (b)

Fig. 45.9 The analysis bound settings for Fig. 45.8. (a) No analysis bound enabled. (b) Analysis bound enabled.

In this example, the Analysis Bound has been configured to match the excavator's digging area. By analyzing the Contents panel in Fig. 45.9 (b), we can derive useful data such as the number of particles (246946) and the bulk density (1509 kg/m³) within the defined region.

45.8.2. Clip Plane

The Clip Plane function is available in the Analysis tab, as shown in Fig. 45.10.

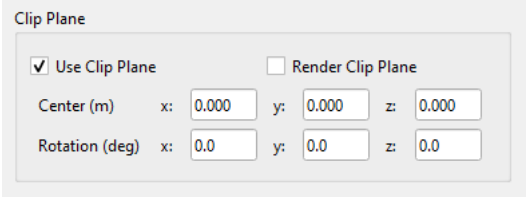


Fig. 45.10 The Clip Plane panel.

This feature introduces an *xz*-plane that slices through the scene, rendering all objects with negative *y*-coordinates invisible, effectively "clipping" those portions. The position and orientation of the Clip Plane can be adjusted by changing its center or rotating it, allowing users to clip the scene in any desired configuration. To active the feature, check **Use Clip Plane**. To activate the visual display of the Clip Plane, which appears as a red rectangle with diagonal lines, check **Render Clip Plane**. The point where these lines intersect represents the center of the Clip Plane, aiding in precise positioning and orientation.

A usage example is shown in Fig. 45.11.

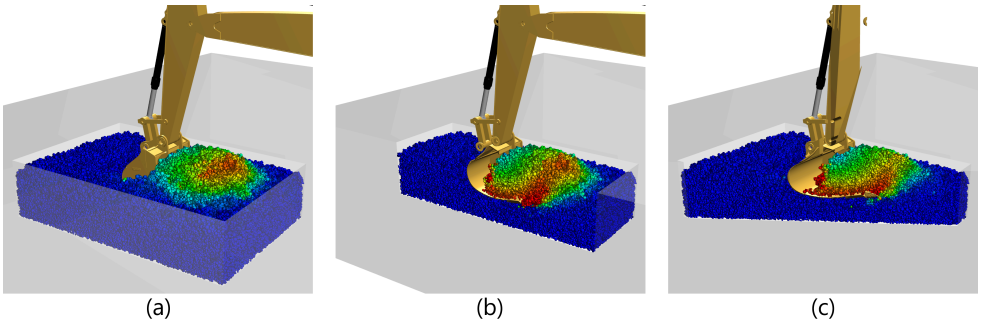


Fig. 45.11 An CAT365 excavator in the process of digging. Particle coloring by velocity. Particle color interval: [0, 1] m/s. (a) Clip Plane not enabled. (b) Clip Plane enabled and positioned to split the scene along the length of the digging area. (c) Clip Plane enabled, centered at the bucket, and rotated 30 degrees about the vertical axis.

In Fig. 45.11 (a), the soil dynamics are obscured by particles and the excavator bucket, demonstrating the need for the Clip Plane. In contrast, Fig. 45.11 (b) reveals the dynamics within the soil and inside the bucket. Finally, Fig. 45.11 (c) shows the effect of rotating the Clip Plane for a different view angle.

45.8.3. Measurement Axes

The Measurement Axes function can be accessed from the Analysis tab, as shown in Fig. 45.12.

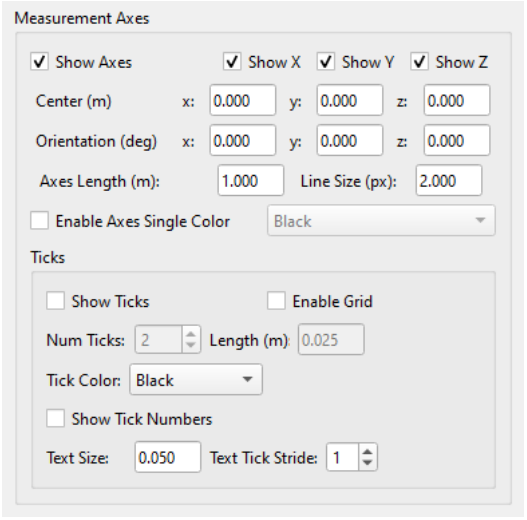


Fig. 45.12 The Measurement Axes panel.

This feature provides a customizable coordinate system that can be moved, rotated, and customized.

To activate the coordinate system, check **Show Axes**, which displays a red *x*-axis, a green *y*-axis, and a blue *z*-axis. Individual axes can be shown or hidden by toggling **Show X**, **Show Y**, or **Show Z**. The coordinate system can be repositioned by adjusting the **Center (m)** values for *x*, *y*, and *z*, with the default position at *(0,0,0)* representing the scene's origin. The axes length and line thickness can be adjusted using **Axes Length (m)** and **Line Size (px)**, respectively. Additionally, you can change the default colors to a single one using **Enable Axes Single Color**, which is helpful when the default colors blend into the scene.

Enabling **Show Ticks** adds tick markers to the axes, and the number and length of these ticks can be adjusted via **Num Ticks** and **Length (m)**. To display grids, check **Enable Grid**, and customize tick colors using **Tick Color**. You can also show tick numbers by enabling **Show Tick Numbers**, with options to adjust the font size (**Text Size**) and change how often numbers appear by specifying a **Text Tick Stride**. The default stride value of 1 labels every tick, while higher values (e.g., 2 or more) reduce the frequency of numbered ticks.

An example of Measurement Axes usage is provided in Fig. 45.13.

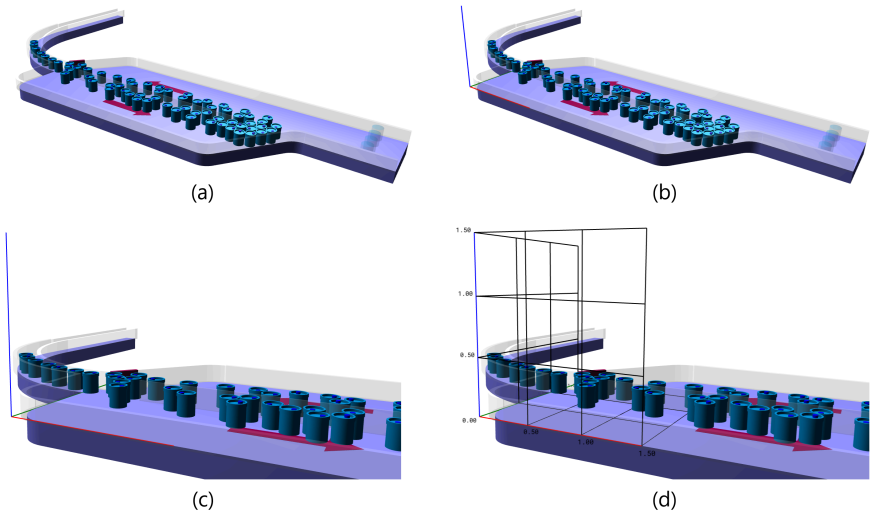


Fig. 45.13 A recirculation conveyor with curved walls transporting barrels. (a) Measurment axes disabled. (b) Measurment axes enabled. Axes length: 1.5 m. Line size: 4px. (c) Close-up view of (b). (d) Same as (c), but with Show Ticks and Enable Grid checked, and Num Ticks set to 3.

45.9. Coloring (tab)

The Journal Viewer provides a variety of tools for coloring and post-processing granular particles, accessible through the **Coloring** tab in the left side docket. This tab contains two sub-tabs: **Particle Color** and **Particle Primitives**. Below is a brief overview of each:

- **Particle Color**: Allows particles to be colored based on scalar values such as velocity, displacement, or radius.
- **Particle Primitives**: Provides visualization of particle dynamics using trajectories and contact networks.

These two sub-tabs are described in the following sections.

45.9.1. Particle Color

The Particle Color tab is displayed in Fig. 45.14.

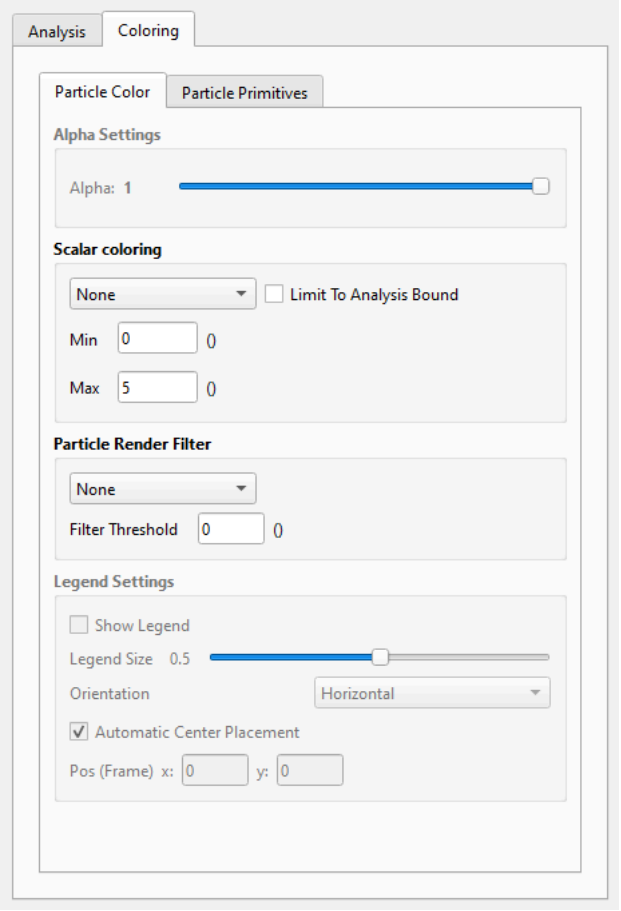


Fig. 45.14 The particle color tab.

The central feature of this tab is to color particles based on scalar quantities such as velocity, displacement, or radius.

The **Alpha** value, found under **Alpha Settings**, controls the opacity of particles when Alpha Sprites are enabled (see Particle Shader).

The dropdown list titled **Scalar coloring** provides a range of scalar quantities for coloring. If **Limit to Analysis Bound** is checked, only particles within the analysis bound will be colored. The colormap is defined by specifying minimum (**Min**) and maximum (**Max**) values. Particles at or below the minimum are colored blue, while those at or above the maximum are colored red. Particles with intermediate values will transition smoothly through a gradient of colors, as shown in Fig. 45.15.

The **Particle Render Filter** applies a threshold to the coloring, meaning only particles exceeding a selected quantity will be colored. For an example, refer to Velocity.

In the **Legend Settings**, the color legend, which reflects the Min-Max color interval, can be customized in terms of size, orientation (horizontal or vertical), and placement. An example legend for displacement, ranging from -2 to 4 m, is shown in Fig. 45.15.

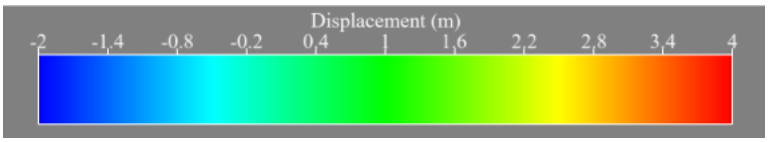


Fig. 45.15 An example legend for displacement. Interval: [-2, 4] m.

45.9.1.1. Contact Force

This feature colors particles based on the magnitude of the contact forces acting on them. Visualizing particles in this way helps identify areas of high stress or compression, which is useful for analyzing how forces are distributed in granular systems. For instance, it can highlight potential breakage points in materials such as powders or soils under pressure. To use this feature, select **Contact Force** from the **Scalar coloring** dropdown. An example is shown in Fig. 45.16.

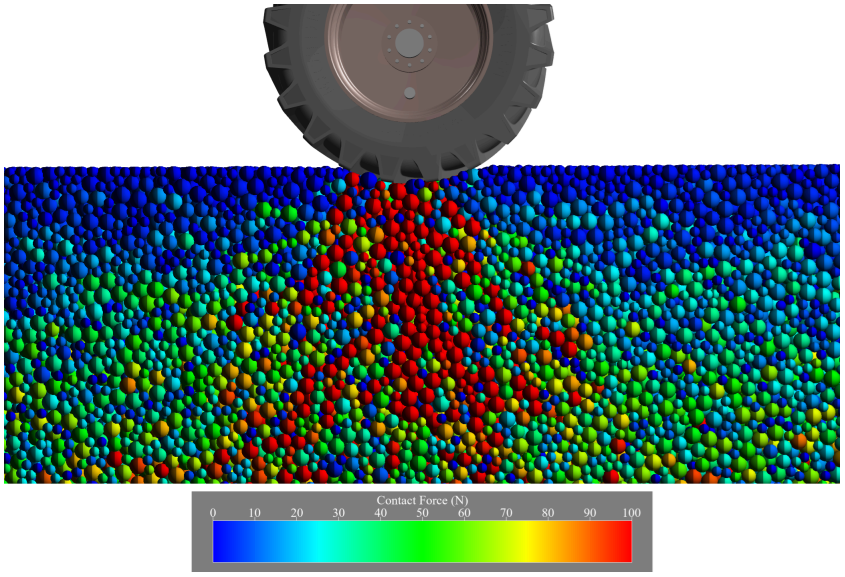


Fig. 45.16 A tractor tyre on a bed of particles. Contact force coloring interval: [0, 100] Nm.

In the example (Fig. 45.16), the highest contact forces are concentrated beneath the tire and spread outward from the tire-soil interface. This pattern is characteristic of granular materials like soil, where forces are transmitted through networks of interconnected force chains. These force chains distribute the load unevenly, creating regions of high stress extending from the point of contact.

45.9.1.2. Displacement

Coloring particles by displacement reveals how far they have moved from their initial positions, making it useful for identifying deformation patterns or flow paths over time. This feature is particularly valuable for studying pile formations, static regions, or other processes involving particle rearrangement.

To use this feature, first navigate to the desired time in the simulation. Then, select **Displacement** from the **Scalar coloring** dropdown, which establishes the reference positions of the particles at that specific moment. Next, set the desired coloring interval. As the journal progresses, particles will be colored according to how far they have moved from their reference positions, representing their displacement. An example is shown in Fig. 45.17.

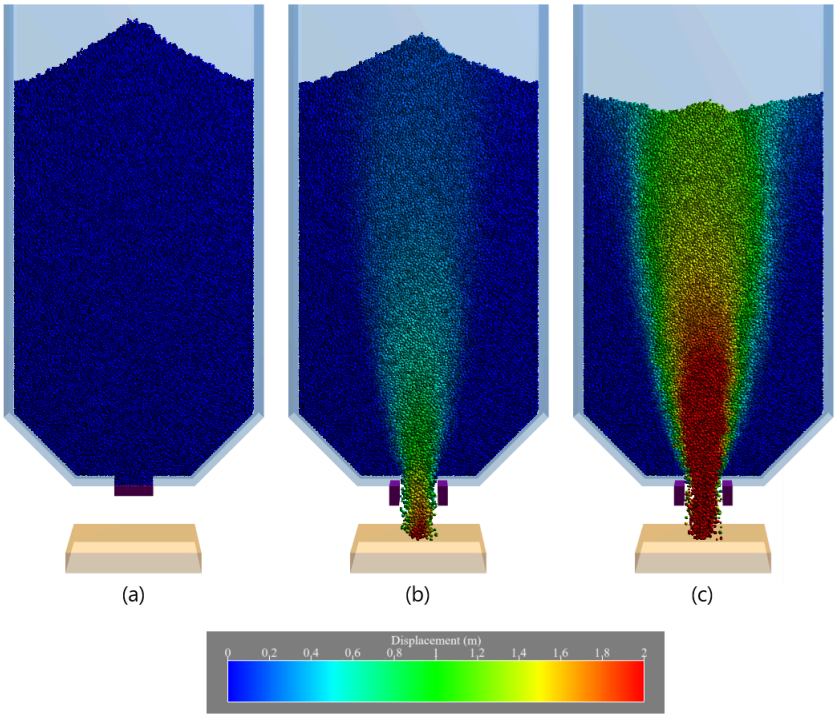


Fig. 45.17 A tall silo with an outlet. Displacement coloring interval: [0, 2] m. (a) The reference time at which the outlet is opened; (b) 1.5 s after the outlet has opened; (c) 4 s after the outlet has opened.

In Fig. 45.17 (a), all particles are blue, indicating a displacement of 0 m, as this is the reference time. In Fig. 45.17 (b), 1.5 seconds later, displacement is concentrated in a semi-elliptic region aligned with the outlet, suggesting this area experiences the most flow. By Fig. 45.17 (c), 4 seconds after opening the outlet, the displacement has expanded, forming a red cylinder through the outlet, extending halfway up the silo. Above this region, a gradient of orange, yellow, and green indicates decreasing displacement, showing non-uniform flow. Meanwhile, particles near the bottom walls remain blue, indicating they are static.

45.9.1.3. Height X (Y, Z)

Coloring particles based on their height along a specific axis provides useful insights into the spatial distribution and layering of particles. It can also help identify particles that exceed a critical height, such as in a silo scenario. To use this feature, select **Height X**, **Height Y**, or **Height Z** from the **Scalar coloring** dropdown. An example is shown in Fig. 45.18.

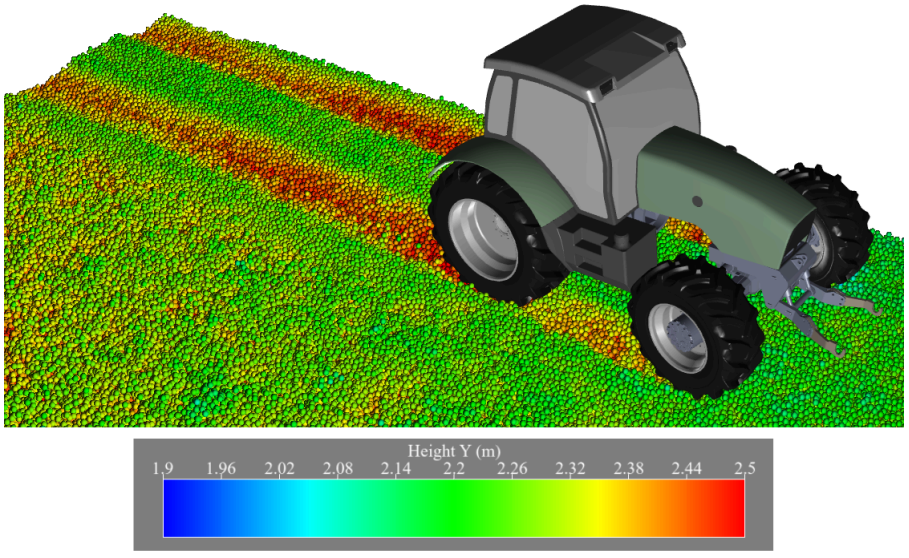


Fig. 45.18 A tractor driving on a bed of particles. Height (Y) coloring interval: [1.9, 2.5] m.

In Fig. 45.18, particles are colored by their height along the *Y* axis, as gravity acts in this direction. This height-based coloring provides an intuitive way to differentiate undisturbed soil from areas affected by the tractor's movement. In this example, the surface of the soil is mostly green, while the tracks left by the tractor are identifiable by their red color.

45.9.1.4. Kinetic Energy

Kinetic energy coloring offers valuable insights into how energy is distributed among particles, helping to track energy dissipation within the system. This is particularly useful for analyzing collisions, identifying areas where energy is concentrated or lost, and optimizing process efficiency. To use this feature, select **Kinetic Energy** from the **Scalar coloring** dropdown. An example is shown in Fig. 45.19.

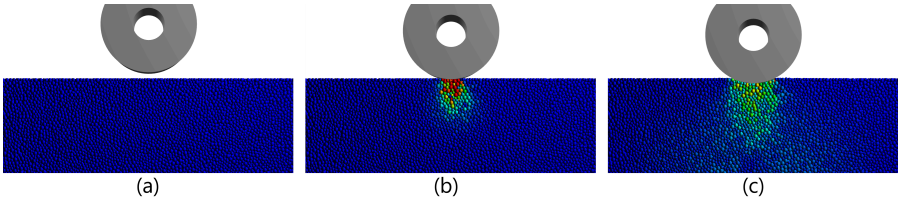


Fig. 45.19 A heavy wall pipe section falling onto a particle bed. Kinetic energy coloring interval: [0, 0.01] J. (a) Prior to impact. (b) About 5 ms after impact. (b) About 25 ms after impact.

We note in Fig. 45.19 (a) that just after impact, the particles just beneath the pipe has absorbed a relatively large amount of kinetic energy. In Fig. 45.19 (b), a short time later, this energy has dissipated and also been transmitted to more distant regions.

In Fig. 45.19 (a), prior to impact, the kinetic energy is essentially zero, as the particles are static. By Fig. 45.19 (b), 5 ms after impact, the particles directly beneath the pipe absorb a significant amount of kinetic energy from the pipe section. By Fig. 45.19 (c), 25 ms after the impact, this energy has begun to dissipate and spread to surrounding regions.

45.9.1.5. Velocity

Coloring particles by velocity helps visualize dynamic regions within the system. This method highlights areas of rapid motion or flow, making it useful for assessing granular flows, detecting eddies, or evaluating mixing efficiency. The coloring is based on the magnitude of the velocity vector. To use this feature, select **Velocity** from the **Scalar coloring** dropdown. An example is shown in Fig. 45.20.

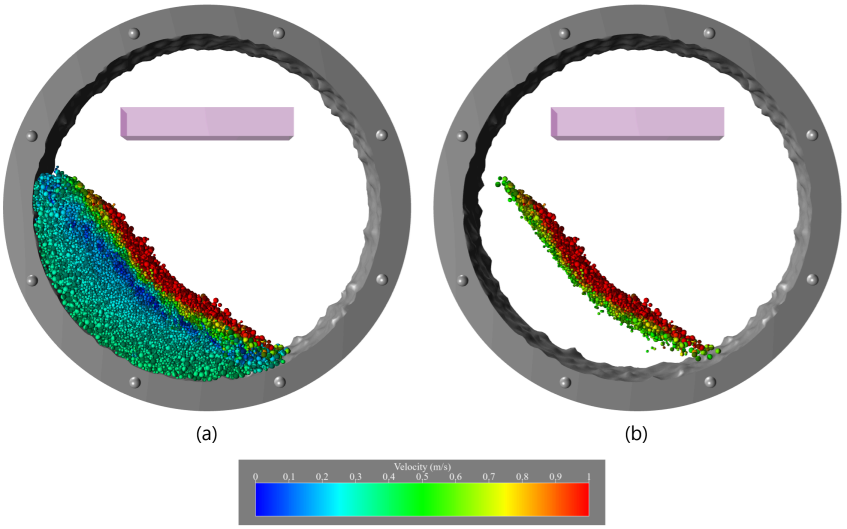


Fig. 45.20 A rotary drum with pellets modelled as granular particles. Velocity coloring interval: [0, 1] m/s. (a) No filter threshold. (b) Filter threshold: 0.5 m/s.

In Fig. 45.20 (a), the highest velocities are observed on the surface of the material, while the velocity is lowest inside the bulk. In Fig. 45.20 (b), a filter threshold of 0.5 m/s has been applied using the **Particle Render Filter**, with **Velocity** selected for filtering. This renders particles below the threshold invisible, effectively isolating regions of faster movement.

45.9.1.6. Radius

This feature colors the particles according to their radii. The radius coloring feature is particularly useful for identifying particle segregation, which is the phenomenon in which granular particles of different sizes separate into distinct regions. This allows users to easily visualize how particles of specific radii migrate and cluster in certain areas of the system over time. To use this feature, select **Radius** from the **Scalar coloring** dropdown. An example is shown in Fig. 45.21.

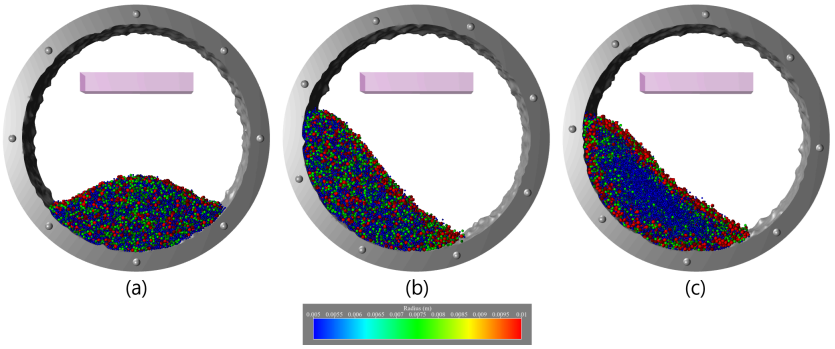


Fig. 45.21 A rotary drum with pellets modelled as granular particles. Radius coloring interval: [0.005, 0.01] m. Particles of radius 0.005, 0.075, and 0.01 m are blue, green, and red, respectively. (a) The initial state; (b) Shortly after the drum has started to rotate; (c) After 30 seconds of rotation.

In Fig. 45.21 (b), shortly after the drum starts rotating, particles of different radii are more or less uniformly distributed. However, as the drum continues to rotate, Fig. 45.21 (c) shows that smaller particles congregate in the center of the bulk.

45.9.2. Particle Primitives

The **Particle Primitives** tab, as shown in Fig. 45.22, provides post-processing tools that visualize particle dynamics using lines and curves. The available features are **Trajectory Lines** and **Contact Network**, which help illustrate the movement and interactions of particles in the system.

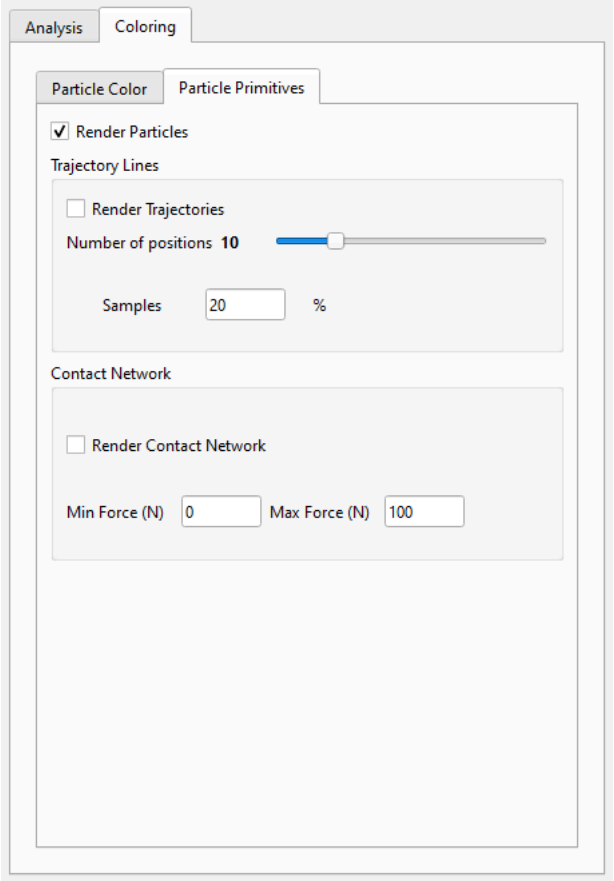


Fig. 45.22 The particle primitives tab.

45.9.2.1. Trajectory Lines

The Trajectory Lines feature analyzes particle motion and visualizes it by drawing trajectory lines that trace the paths of the particles. This feature offers two key settings:

- **Number of positions**: This setting defines how many previous particle positions are used to create the trajectory line. A trajectory is constructed by connecting these past positions with straight segments. If set to 2, the trajectory consists of a single straight line. With values of 3 or higher, the lines become more complex, capturing more detailed particle movement.
- **Samples**: This setting determines the percentage of particles to include when rendering trajectory lines.

An example is shown in Fig. 45.23.

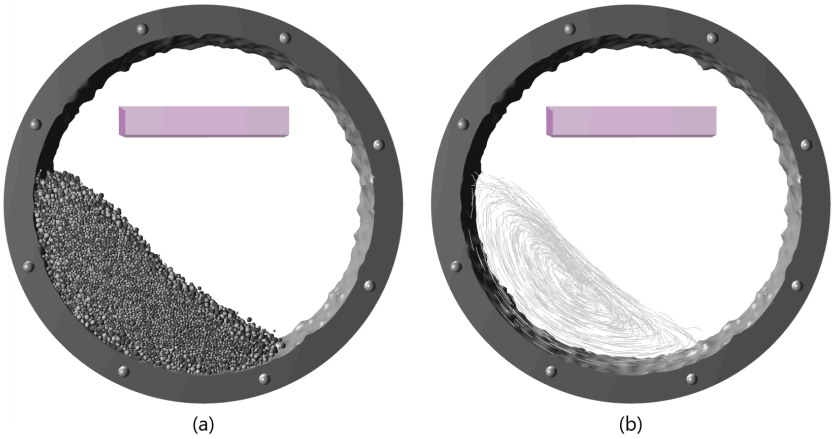


Fig. 45.23 A rotary drum with pellets modelled as granular particles. (a) No particle trajectories rendered. (b) Particle trajectories rendered. Number of positions: 8. Samples: 4%.

45.9.2.2. Contact Network

The Contact Network feature allows users to visualize all contact forces between particles, represented as lines connecting them. It is similar to the Contact Force coloring option (see Contact Force), but it additionally provides a geometric representation of the forces.

To use this feature, enable **Render Contact Network** in the **Contact Network** panel and specify the **Min Force (N)** and **Max Force (N)** values. These values work similarly to the Min and Max settings used in Scalar coloring. When rendering the Contact Network, it may be helpful to disable **Render Particles** at the top of the Particle Primitives tab for a clearer view of the network. An example is shown in Fig. 45.24.

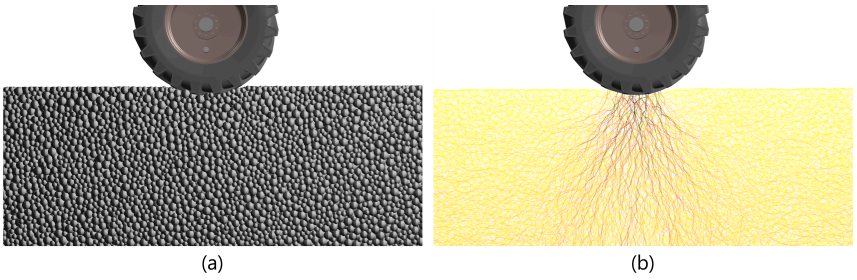


Fig. 45.24 A tyre on a particle soil bed. (a) Contact Network not rendered. Render Particles enabled. (b) Contact Network rendered. Min Force: 0 N. Max Force: 100 N. Render Particles disabled.

45.10. Main (menu)

45.10.1. Open File

This option opens a dialog for selecting and loading a journal file (see Load journal). Journal files use the `.agxJournal` extension. Additionally, you can use this option to open a configuration file containing camera views (see Cameras). Configuration files have the `.cfg` extension.

45.10.2. Save simulation state

This option saves the current simulation state (or scene) as an `.agx` file at the time indicated by the Journal Viewer's progress bar. Note that this saves the scene, not a new journal. The saved scene represents a simulation starting from the selected time, with the same configuration as the journal at that moment. This feature is useful for making moderate adjustments to the simulation, such as adding or removing a rigid body, which is possible in a scene but not within a journal.

45.10.3. Exit

This exits the Journal Viewer. Yes, it's as straightforward as it sounds.

45.11. View (menu)

This menu contains items that influences and changes the settings of the viewing window displaying the playback of the journal.

45.11.1. Draw Mode

The **Draw Mode** setting allows you to choose the method for rendering the scene. The default option is Polygons, but you can also select Wireframe or Points for different visualization styles, see Fig. 45.25.

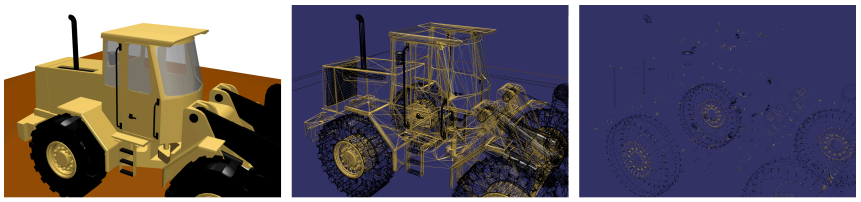


Fig. 45.25 A scene featuring an L70 wheel loader rendered in different Draw Modes. (a) Polygons. (b) Wireframe. (c) Points.

45.11.2. Center Scene (Space)

This command adjusts the camera view to ensure all objects in the scene are visible. If the view appears distant, it may be due to objects positioned far away, such as an object that has unintentionally fallen and accelerated downward for a long time due to gravity.

45.11.3. Draw Statistics (B)

This command toggles the display of simulation statistics on the screen, see Fig. 45.26.

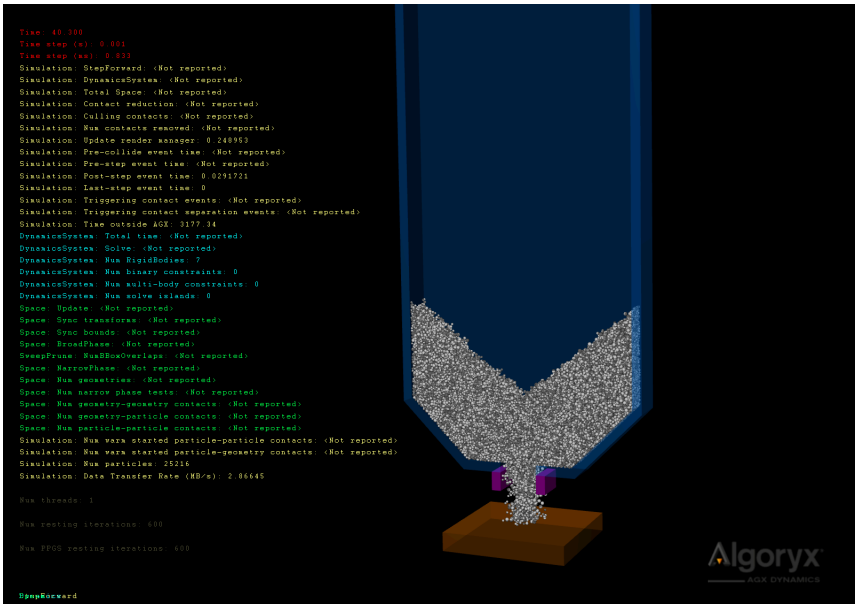


Fig. 45.26 A scene showing a discharging silo with Draw Statistics enabled.

The displayed statistics include details such as simulation time, time step, number of rigid bodies (Num RigidBodies), number of particles (Num Particles), and more. When playing a journal, fewer statistics are shown compared to a running simulation, as not all data is saved during journal recording. If a statistic is unavailable, `<Not reported>` will be displayed.

45.11.4. Increment Clip Plane (+)

This command moves the Clip Plane forward by a small distance along its normal direction. Since the Clip Plane's orientation can be adjusted by rotating it, you can control its movement in any desired direction by applying the appropriate rotations.

45.11.5. Decrement Clip Plane (-)

This command moves the Clip Plane backward by a small distance along its normal direction.

45.11.6. Toggle View Side Docket (P)

This command toogles the visibility of the Side Docket (The panel with tabs *Analysis* and *Coloring*).

45.11.7. Set Window size

This menu item allows you to specify the size of the Journal Viewer window. It provides an alternative to manually resizing the window by dragging its edges. You can select predefined sizes such as HD, Full HD, or 4K.

45.11.8. Set View

This menu item allows you to specify the direction of the camera in a particular direction: *X*, *Y*, *Z*, *-X*, *-Y*, *-Z*, *Clip Plane*.

- If *X* or is selected, the camera is positioned at *(X_0, 0, 0)*, *X_0 > 0*, and faces toward the origin in the World (Scene) coordinate system. (Similarly for *Y* and *Z*.)

- If *-X* is selected, the camera is positioned at *(-X_0, 0, 0)*, *X_0 > 0*, and faces toward the origin in the World (Scene) coordinate system. (Similarly for *Y* and *Z*.)
- If Clip Plane is selected, the camera is positioned at *(0, 0, Z_0)*, *Z_0 > 0*, and faces toward the origin in the Clip Plane coordinate system.

45.11.9. Enable Ortographic View

This menu option toggles the **Orthographic View** on or off. The Journal Viewer supports two types of views: **Perspective** and **Orthographic**. When Orthographic View is disabled, Perspective View is automatically enabled.

In **Perspective View**, the scene is rendered similarly to how the human eye perceives it. Objects appear smaller as they move farther from the camera, with lines converging toward a vanishing point. This creates a sense of depth and spatial relationships, enhancing realism and making it ideal for visualizing how objects interact in 3D space.

On the other hand, **Orthographic View** displays objects without perspective distortion. In this mode, lines remain parallel, and objects maintain their size regardless of their distance from the camera. This is useful for measurements and for easier visualization of geometrical relationships.

When Orthographic View is enabled, zooming with the mouse is disabled, though you can still rotate the camera using the mouse. Zooming in or out can be done via the keyboard, refer to: Zoom Ortographic View + (Ctrl++).

An example is shown in Fig. 45.27.

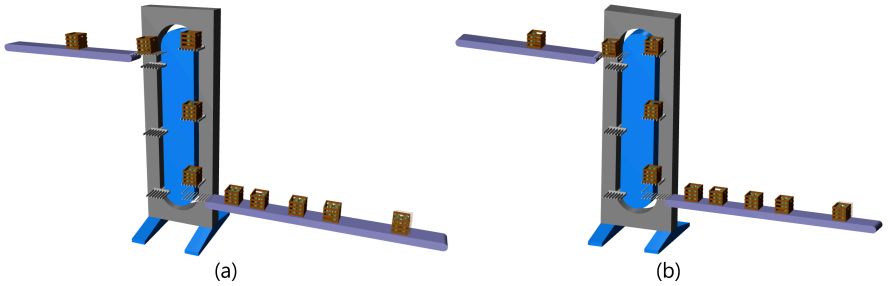


Fig. 45.27 A vertical conveyor lifting wooden boxes with canisters. (a) Perspective (default) view. (b) Orthographic view.

45.11.10. Zoom Ortographic View + (Ctrl++)

This command zooms in a small distance when Orthographic mode is enabled. When disabled, the command has no effect.

45.11.11. Zoom Ortographic View - (Ctrl-)

This command zooms out a small distance when Orthographic mode is enabled. When disabled, the command has no effect.

45.12. Cameras (menu)

45.12.1. Open Camera File (J)

This command opens a Camera file from a directory of your choice. File extension: *.cfg*.

45.12.2. Save Cameras (K)

This command saves a Camera file to a directory of your choice. File extension: *.cfg*.

45.12.3. Show Views (H)

This command display a panel to the right with a list of the views of the camera. If you have not stored any cameras or have not loaded a camera file, the list is empty.

45.12.4. Store Current Camera (M)

This command stores the current camera (view), meaning that is it accessible from the list of views. Note: It does not save the camera views to a file.

45.12.5. Next Camera (PgUp)

This command switches to the next camera view in the camera view list.

45.12.6. Previous Camera (PgDown)

This command switches to the previous camera view in the camera view list.

45.13. Media (menu)

45.13.1. Video Generation Settings (V)

This option opens up the Video Generation Settings window, which is used for the creation of a video, see Media.

45.13.2. Capture Video (C)

This option starts the recording of a video, see Media.

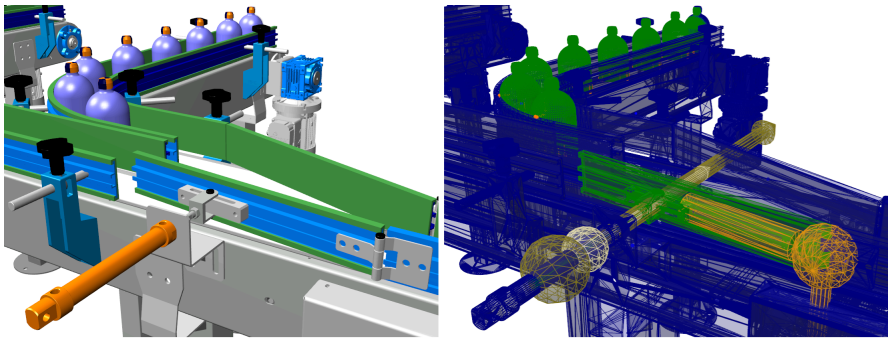
45.13.3. Take Screenshot (F12)

This feature captures a screenshot of the current frame in the viewing window and saves the image to the journal directory. The screenshot reflects the current size of the viewing window, meaning a full-screen viewer will produce a larger image compared to a smaller window.

45.14. Rendering (menu)

45.14.1. Debug Rendering (G)

This option toggles **Debug Rendering** on or off. For more information, see Debug rendering. A comparison of debug rendering enabled and disabled is shown in Fig. 45.28.



(a) (b)

Fig. 45.28 A two-lane conveyor belt transporting bottles. (a) Debug rendering off (standard rendering on). (b) Debug rendering on (standard rendering off).

Although debug rendering is less visually appealing than standard rendering, it is highly useful for inspecting and analyzing the simulation. Examples of what debug rendering reveals include:

- **Meshes**: Mesh geometries are represented by straight lines forming triangles.
- **Contacts**: Contacts are marked by small orange spheres with lines indicating their direction. See for example the interface between the curved wall and the bottles.
- **Constraints**: Constraints are visualized by various objects. For instance, the prismatic constraint governing the dynamics of the actuating orange cylinder is represented by a light brown arrow.

45.14.2. Standard Rendering (O)

This option toggles **Standard Rendering** on or off. Standard rendering is displayed in Fig. 45.28 (a). If both Standard Rendering and Debug Rendering are disabled, the viewing window will appear empty. When using Debug Rendering, disabling Standard Rendering can improve performance and make it easier to identify simulation objects.

45.14.3. Mesh Rendering (O)

The **Mesh Rendering** feature includes two options: **Collision Mesh** and **Render Mesh**.

- **Collision Mesh** toggles the visibility of the collision mesh.
- **Render Mesh** toggles the visibility of the render mesh.

To visualize certain elements, such as a Surface Modifier, the collision mesh must be enabled. For an example, refer to Fig. 45.29.

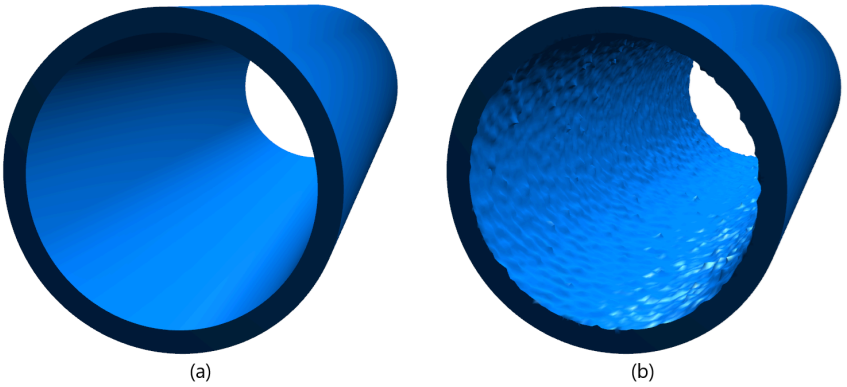


Fig. 45.29 An empty rotary drum with a Surface Modifier. (a) Collision mesh disabled. (b) Collision mesh enabled.

45.14.4. Particle Shader

The **Particle Shader** provides several options for rendering particles, with three available modes:

- **Sprites**: This mode applies a single color to the particles with shading. As a result, any rotation of a particle is not visually detectable.
- **Rotational Sprites**: This mode colors the upper and lower hemispheres of a particle in the same color but different shades. This allows the rotation of a particle to be visually detected.
- **Alpha Sprites**: Similar to Rotational Sprites, but with the added ability to adjust the alpha (opacity) value. The alpha value can be modified under *Coloring* > *Particle Color* > *Alpha Setting*. A value of 1 represents full opacity, 0 represents full transparency, and intermediate values produce corresponding levels of opacity.

The three options are illustrated in Fig. 45.30.

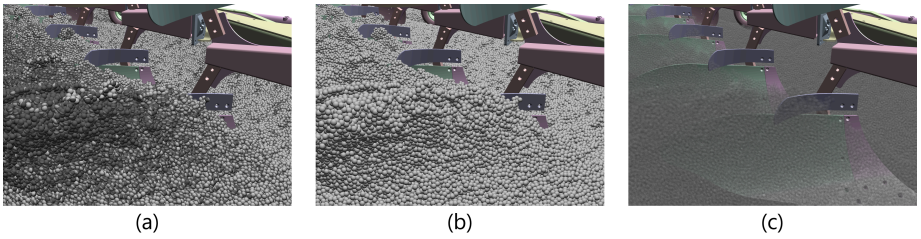


Fig. 45.30 Close-up of a plough digging in a particle soil bed, showing different Particle Shader options. (a) Sprites. (b) Rotational Sprites. (c) Alpha Sprites.

A distant view of the scene is shown in Fig. 45.31.

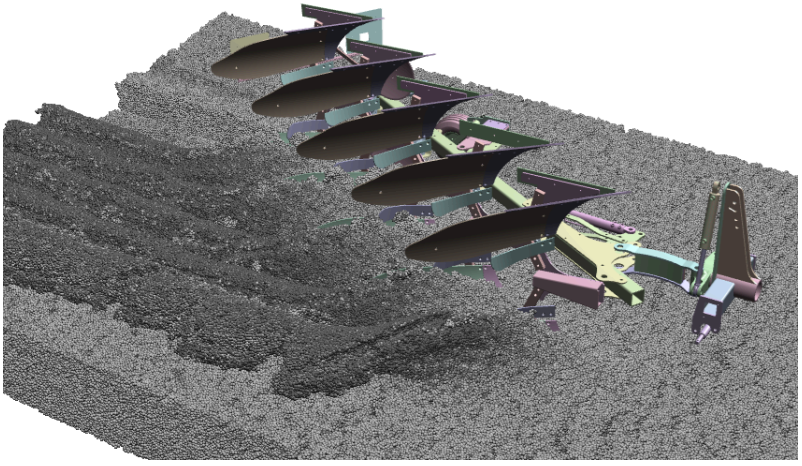


Fig. 45.31 Far view of a plough digging in a particle soil bed.

45.14.5. Use Algoryx Logo

This option controls whether the Algoryx logo appears in the lower-right corner of the viewing window. It also applies to video exports.

45.14.6. Background Color

This option allows you to select the background color for the viewing window. It also applies to video exports.

45.15. Keyshot (menu)

KeyShot is a third-party 3D rendering and animation software and is not included with AGX Dynamics. To utilize KeyShot for rendering journal data, you will need to purchase it separately. BIP is the native file format used by KeyShot, containing all relevant data associated with the model. In KeyShot, you can create 3D renderings or animations using a series of BIP files.

45.15.1. Export Current Frame to Bip

This item exports a (single) BIP file from the current frame to the folder of the journal.

45.15.2. Export Journal To Bip files

This option opens the window shown in Fig. 45.32.

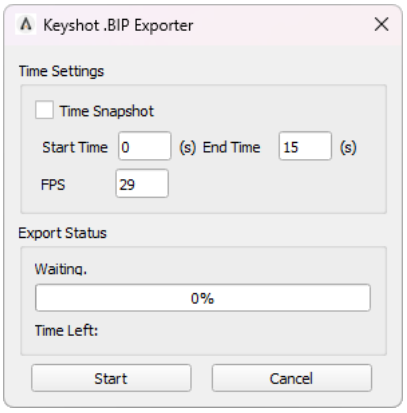


Fig. 45.32 The Keyshot Bip Exporter window.

From this window, you can generate a series of Bip files by selecting a **Start Time** and an **End Time**. You can also specify the **FPS** (Frames Per Second). If **Time Snapshot** is checked, only a single Bip file is generated at the specified Start Time. Clicking **Start** begins the export process, and the Bip files are saved to the journal's folder.

45.16. Data Export (menu)

45.16.1. Write Simulation Data to File

This option opens the window shown in Fig. 45.33.

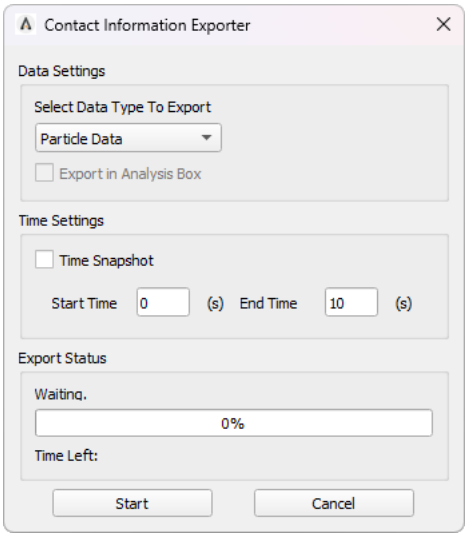


Fig. 45.33 The Contact Information Exporter.

The Contact Information Exporter allows you to export simulation data to a *.csv* file for use in analysis, plotting, or other programs. The *.csv* file is saved in the same folder as the journal.

Select Data Type to Export: This panel allows you to choose between two data types: **Particle Data** and **Impact Contact Data**. The exported data includes:

- **Particle Data**: particleId, radius, pos x, pos y, pos z, accumulatedEnergy
- **Impact Contact Data**: ContactType, bodyId1, bodyId2, pos x, pos y, pos z, contactEnergy, normalContactForce, impactForce

Time Settings: This panel lets you specify the time interval for exporting data using **Start Time** and **End Time**. For example, if Start Time is set to 0 s and End Time to 10 s, data will be exported for this interval, with a row of data written for each timestep. To export data over a time interval, ensure **Time Snapshot** is unchecked. If **Time Snapshot** is checked, End Time is disabled, and only the Start Time can be set, exporting data for a single timestep at the specified Start Time.

Export Data: This panel includes controls for exporting data to a *.csv* file. Click **Start** to begin the export process (click **Cancel** to abort). The progress bar below **Waiting** shows the progress of the export. The *.csv* file is saved in the same folder as the journal.

45.17. info (menu)

45.17.1. Journal Info

The **Journal Info** option provides information about the simulation settings, as shown in Fig. 45.34. This includes details such as the timestep, the number of resting iterations, whether PPGS is used, and other relevant settings. This feature is particularly useful for reviewing the simulation settings, as it serves as a convenient "simulation settings document." Instead of embedding these details in the journal filename or storing them in a separate text file, Journal Info centralizes the information in one place.

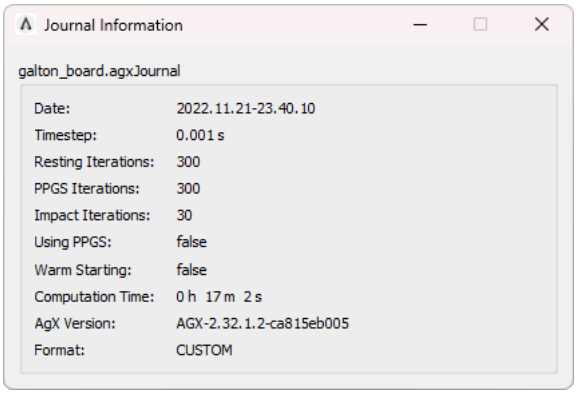


Fig. 45.34 Journal info for the journal *galton_board.agxJournal*.

45.18. Custom Scripting

The Journal Viewer allows for the attachment of custom Python scripts, enabling custom post-processing capabilities. To attach a single Python script, use the command line argument `--attachScript <filename>`. To make all Python modules in a directory available, use the command line argument `--addCustomScriptPath <path>`. You can view all available command line arguments by executing:

```
>> journalViewer --help
```

Examples of using both `--attachScript <filename>` and `--addCustomScriptPath <path>` are provided in the sections below.

45.18.1. Attach Script

To attach a custom Python script, `my_custom_script.py`, to a journal, `my_journal.agxJournal`, run the following terminal command:

```
>> journalViewer my_journal.agxJournal --attachScript my_custom_script.py
```

Let's illustrate this with an example. Consider the scene shown in Fig. 45.35.

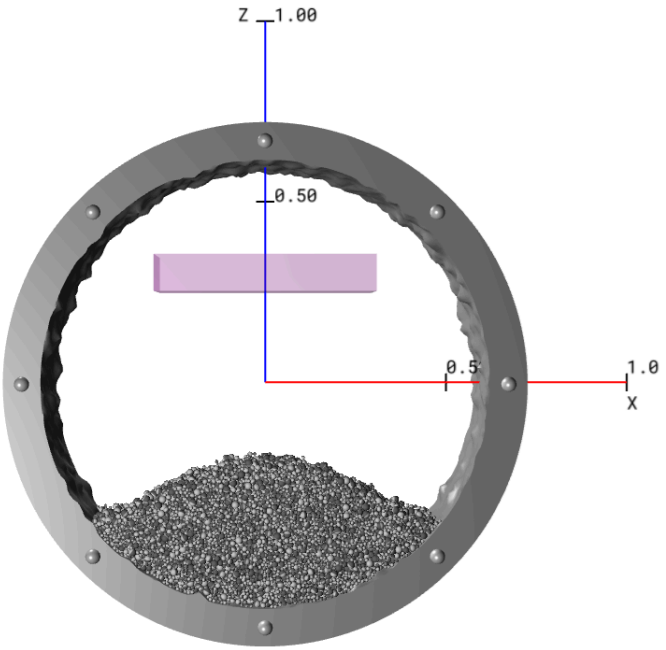


Fig. 45.35 The initial state of a rotating drum with particle pellets. The coordinate system is displayed using Measurement Axes (see Measurement Axes).

To visually inspect the mixing dynamics and efficiency of the pellets in the drum, we can color the particles in two groups: the left half in one color, and the right half in another. We can implement this functionality using a custom script, as described in Code snippet 1 - ParticleColorer.py.

Since the drum is centered at *x=0*, we can divide the particles based on their position. The script defines a class `ParticleColorer`, which listens to each simulation time step. After a certain time (`TIME_TO_COLOR`, set to 4 seconds), the particles are grouped based on their *x*-coordinate. The IDs of particles in the right half (`x >= 0`) are stored in one variable, and the IDs of particles in the left half (`x <= 0`) in another. These variables are assigned once. Then, at each time step, the particles in the right group are colored with one color (`COLOR_A`, set to purple), and those in the left group with another (`COLOR_B`, set to yellow).

The complete script is provided in Code snippet 1 - ParticleColorer.py. The journal filename is `rotary_drum.agxJournal` and we place `ParticleColorer.py` in the same folder as the journal. To apply the script, execute the following command in the terminal:

```
>> journalViewer rotary_drum.agxJournal --attachScript ParticleColorer.py
```

The result is shown in Fig. 45.36.

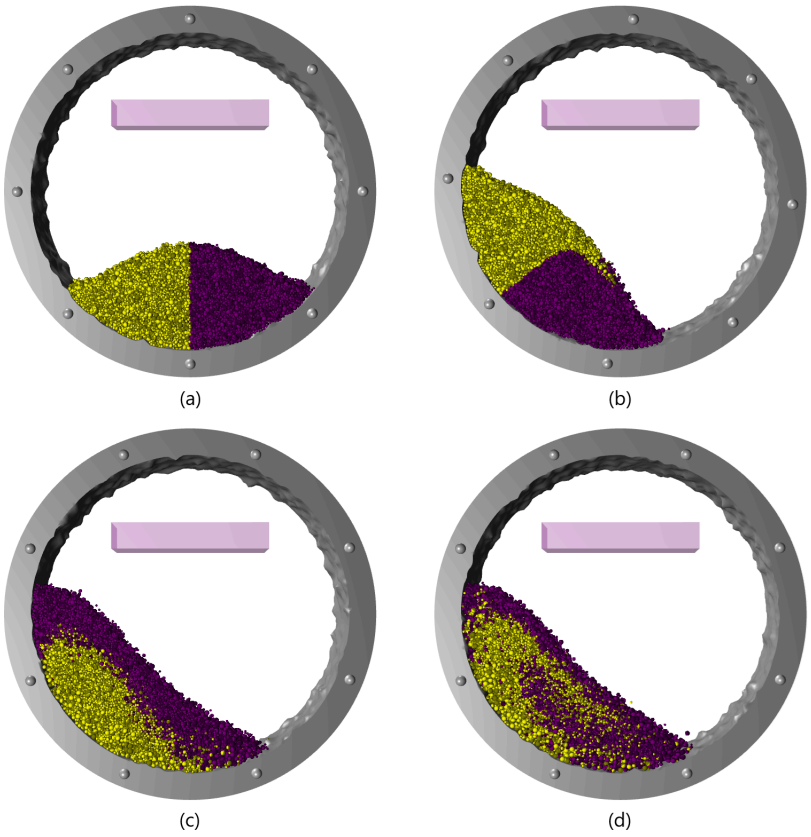


Fig. 45.36 The evolution of the granular material with `ParticleColorer.py` attached. (a) Time: 4 s. (b) Time: 7 s. (c) Time: 10 s. (d) Time: 15 s.

```
import numpy as np
import agxSDK
import agxPython
from agxPythonModules.utils.numpy_utils import BufferWrapper

TIME_TO_COLOR = 4  # Time to start coloring
COLOR_A = [0.502, 0, 0.502, 1]  # Purple
COLOR_B = [1, 1, 0, 1]  # Yellow

def buildViewerScene() -> None:
    '''Build the viewer scene and add the particle colorer to the simulation.'''
    sim = agxPython.getContext().environment.getSimulation()
    particle_colorer = ParticleColorer(sim)
    sim.add(particle_colorer)

class ParticleColorer(agxSDK.StepEventListener):
    def __init__(self, sim: agxSDK.Simulation) -> None:
        super().__init__()
        self.sim: agxSDK.Simulation = sim
        self.particle_ids_x_geq_0: Optional[np.ndarray] = None
        self.particle_ids_x_lt_0: Optional[np.ndarray] = None

    def post(self, time: float) -> None:
        '''Executed after each simulation step.'''
        if time >= TIME_TO_COLOR:
            if self.particle_ids_x_geq_0 is None or self.particle_ids_x_lt_0 is None:
                self.particle_pos = BufferWrapper(self.sim, 'Particle.position').array
                self.particle_id_to_index = BufferWrapper(self.sim, 'Particle._idToIndex').array
                self.particle_color = BufferWrapper(self.sim, 'Particle.color').array
                self.particle_ids_x_geq_0, self.particle_ids_x_lt_0 = \
                    self.get_particle_ids_for_coloring()
            self.color_particles()

    def color_particles(self) -> None:
        '''Color particles based on their IDs.'''
        indices_x_geq_0 = self.particle_id_to_index[self.particle_ids_x_geq_0]
```

```
        indices_x_lt_0 = self.particle_id_to_index[self.particle_ids_x_lt_0]
        self.particle_color[indices_x_geq_0] = COLOR_A
        self.particle_color[indices_x_lt_0] = COLOR_B

    def get_particle_ids_for_coloring(self) -> tuple[np.ndarray, np.ndarray]:
        '''
        Get the IDs of particles based on their x-coordinate.
        particle_ids_x_geq_0: particles with x >= 0
        particle_ids_x_lt_0:  particles with x < 0
        '''
        particle_index_to_id = np.argsort(self.particle_id_to_index[:, 0])
        particle_ids_x_geq_0 = particle_index_to_id[np.where(self.particle_pos[:, 0] >= 0)[0]]
        particle_ids_x_lt_0 = particle_index_to_id[np.where(self.particle_pos[:, 0] < 0)[0]]
        return particle_ids_x_geq_0, particle_ids_x_lt_0
```

Code Snippet 1 - `ParticleColorer.py`

45.18.2. Add Custom Scripts Path

In the example from the previous section, where we colored particles in two distinct colors, it was sufficient to contain all the custom scripting in a single script, as the code was relatively short. However, for more extensive post-processing, it can be more convenient to divide the script across several files for better clarity. This approach can be utilized in the Journal Viewer by using the `--addCustomScriptPath <Path>` argument.

To attach a custom Python script, `my_custom_script.py`, to a journal, `my_journal.agxJournal`, while also making the modules in `D:/my_custom_utils/` accessible to `my_custom_script.py`, execute the following terminal command:

```
>> journalViewer my_journal.agxJournal --attachScript my_custom_script.py --addCustomScriptPath D:/my_custom_utils/
```

Let's illustrate this with an example. We will reuse the functionality from the previous section, where particles were divided at *x=0* into left and right collections. Additionally, we will add functionality to store the centers of mass for both collections and save this data into a `.json` file for later plotting. We will break out the `ParticleColorer` class into a separate file, `particle_colorer.py`, create a new class, `CenterOfMassAnalyzer`, to handle center of mass calculations in `CenterOfMassAnalyzer.py`, and create a class, `ParticlePostProcessor`, responsible for managing both previous classes in `ParticlePostProcessor.py`. We place `ParticleColorer.py` and `CenterOfMassAnalyzer.py` in `D:/particle_post_processor_utils/`, while placing `ParticlePostProcessor.py` in the same directory as the journal `rotary_drum.agxJournal`. (See Code snippet 3 - ParticlePostProcessor.py, Code snippet 4 - ParticleColorer.py, and Code snippet 5 - CenterOfMassAnalyzer.py for details.)

Navigate to the directory where the journal is located. To execute the post-processing script, use the following terminal command:

```
>> journalViewer rotary_drum.agxJournal --attachScript ParticlePostProcessor.py --addCustomScriptPath
D:/particle_post_processor_utils/
```

A key feature here is that `--addCustomScriptPath D:/particle_post_processor_utils/` makes the modules in that directory available to `ParticlePostProcessor.py`. By studying Code snippet 3 - ParticlePostProcessor.py, we can see that this script imports the `ParticleColorer` and `CenterOfMassAnalyzer` classes from the files in the directory `D:/particle_post_processor_utils/`.

When running the above command, the particle coloring works as described in the previous section. The new addition is the creation of a `.json` file that contains the center of mass positions for both particle collections as a function of time. An (abbreviated) version of this file is shown in Code snippet 2 - com_data.json

```
{
    "time": [
        4.0000000000002185,
        4.033333333333557,
        ...
    ],
    "pos_left": [
        [
            -0.16945233435221013,
            2.187992830899125e-05,
            -0.40797553851422996
        ],
        [
            -0.16945324192040673,
            2.1973870151488706e-05,
            -0.40797750815201633
        ],
```

```
        ...
    ],
    "pos_right": [
        [
            0.16720595499825752,
            -3.968564056495067e-05,
            -0.4074566484874103
        ],
        [
            0.16720676731927853,
            -3.968090879135949e-05,
            -0.4074585782608062
        ],
        ...
    ]
}
```

Code Snippet 2 - `com_data.json`

This `.json` data can be plotted using various programs. A plot generated by a Python script is shown in Fig. 45.37.

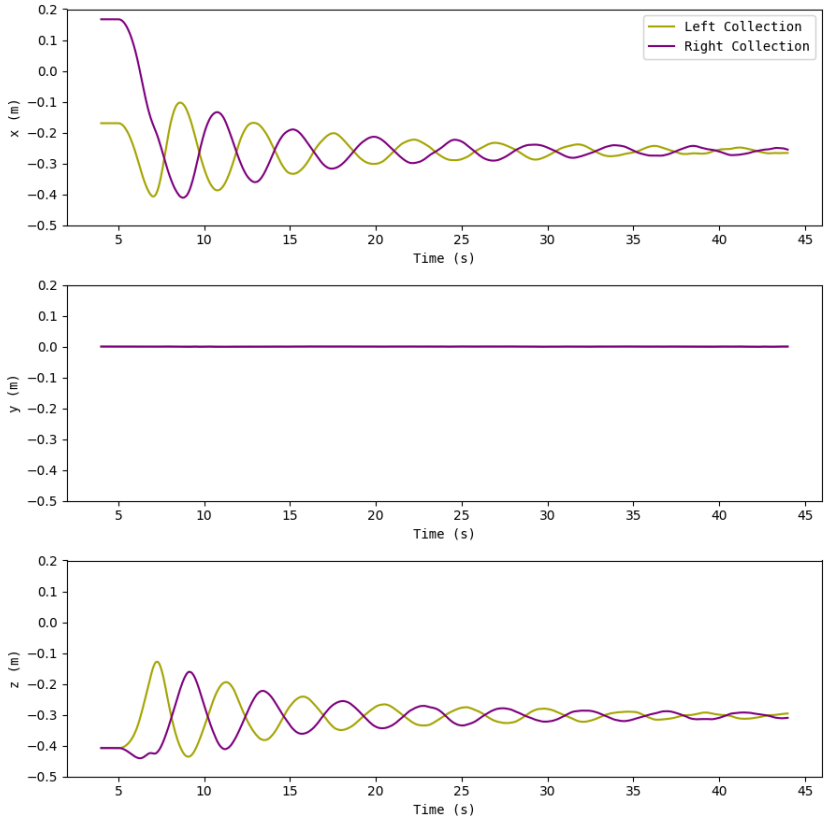


Fig. 45.37 Center of mass coordinates vs. time for the two particle collections in Fig. 45.35. Plotted using the data in Code snippet 2 - com_data.json.

As shown in Fig. 45.37, the mass centers in the *xz*-plane move in elliptical trajectories with decreasing amplitudes. This behavior is expected: as the two collections mix, their centers of mass converge, leading to the decays. Additionally, both *y* coordinates remain almost constant, which is also expected, as the drum's axis lies along the *y*-axis, and there is no significant flow in the axial direction.

```
import numpy as np
import agxSDK
import agxPython
from agxPythonModules.utils.numpy_utils import BufferWrapper
from ParticleColorer import ParticleColorer
from CenterOfMassAnalyzer import CenterOfMassAnalyzer

settings = {
    "time_to_start": 4,  # Start time for coloring and analysis
    "time_to_end": 44,  # End time for coloring and analysis
    "color_A": [0.502, 0, 0.502, 1],  # Purple
```

```
    "color_B": [1, 1, 0, 1],  # Yellow
    # Path to save center of mass data
    "path_to_data_file_to_save": "D:/particle_post_processor_utils/com_data.json"
}

def buildViewerScene() -> None:
    '''Build the viewer scene and add particle post-processor.'''
    sim = agxPython.getContext().environment.getSimulation()
    particle_post_processor = ParticlePostProcessor(sim, settings)
    sim.add(particle_post_processor)

class ParticlePostProcessor(agxSDK.StepEventListener):
    def __init__(self, sim: agxSDK.Simulation, settings: dict) -> None:
        super().__init__()
        self.sim = sim
        self.settings = settings
        self.particle_colorer = ParticleColorer(self.settings)
        self.center_of_mass_analyzer = CenterOfMassAnalyzer(self.settings)
        self.has_saved_data = False
        self.particle_ids_x_geq_0 = None
        self.particle_ids_x_lt_0 = None

    def post(self, time: float) -> None:
        '''Executed after each simulation step.'''
        if self.settings["time_to_start"] < time <= self.settings["time_to_end"]:
            if self.particle_ids_x_geq_0 is None or self.particle_ids_x_lt_0 is None:
                self.initiate_buffers()
                self.particle_ids_x_geq_0, self.particle_ids_x_lt_0 = \
                    self.particle_colorer.get_particle_ids_for_coloring(
                        self.particle_pos,
                        self.particle_id_to_index)
            self.particle_colorer.color_particles(
                self.particle_color,
                self.particle_id_to_index,
                self.particle_ids_x_geq_0,
                self.particle_ids_x_lt_0)
            self.center_of_mass_analyzer.store_data(
                time,
                self.particle_pos,
                self.particle_mass,
                self.particle_id_to_index,
                self.particle_ids_x_geq_0,
                self.particle_ids_x_lt_0)
        elif time > self.settings["time_to_end"] and not self.has_saved_data:
            self.center_of_mass_analyzer.save_data()
            self.has_saved_data = True

    def initiate_buffers(self) -> None:
        '''Initialize simulation buffers for particles.'''
        self.particle_pos = BufferWrapper(self.sim, 'Particle.position').array
        self.particle_id_to_index = BufferWrapper(self.sim, 'Particle._idToIndex').array
        self.particle_mass = BufferWrapper(self.sim, 'Particle.mass').array
        self.particle_color = BufferWrapper(self.sim, 'Particle.color').array
```

Code Snippet 3 - `ParticlePostProcessor.py`

```
import numpy as np

class ParticleColorer():
    def __init__(self,  settings: dict) -> None:
        self.settings: dict = settings

    def color_particles(self, particle_color, particle_id_to_index, particle_ids_x_geq_0, particle_ids_x_lt_0) -> None:
        '''Color particles based on their IDs.'''
        indices_x_geq_0 = particle_id_to_index[particle_ids_x_geq_0]
        indices_x_lt_0 = particle_id_to_index[particle_ids_x_lt_0]
        particle_color[indices_x_geq_0] = self.settings["color_A"]
        particle_color[indices_x_lt_0] = self.settings["color_B"]
```

```
    def get_particle_ids_for_coloring(self, particle_pos, particle_id_to_index) -> tuple[np.ndarray, np.ndarray]:
        '''
        Get the IDs of particles based on their x-coordinate.
        particle_ids_x_geq_0: particles with x >= 0
        particle_ids_x_lt_0:  particles with x < 0
        '''
        particle_index_to_id = np.argsort(particle_id_to_index[:, 0])
        particle_ids_x_geq_0 = particle_index_to_id[np.where(particle_pos[:, 0] >= 0)[0]]
        particle_ids_x_lt_0 = particle_index_to_id[np.where(particle_pos[:, 0] < 0)[0]]
        return particle_ids_x_geq_0, particle_ids_x_lt_0
```

Code Snippet 4 - `ParticleColorer.py`

```
import numpy as np
import json

class CenterOfMassAnalyzer():
    def __init__(self, settings: dict) -> None:
        self.settings = settings
        self.data = {
            "time": np.array([]),  # Array to store time steps
            "pos_left": np.empty((0, 3)),  # Empty array for left side positions
            "pos_right": np.empty((0, 3))  # Empty array for right side positions
        }

    def store_data(self, time, particle_pos, particle_mass, particle_id_to_index, particle_ids_x_geq_0,
particle_ids_x_lt_0) -> None:
        '''Store the center of mass for left and right particle collections.'''
        center_of_mass_left = self.get_center_of_mass(particle_pos, particle_mass, particle_id_to_index,
particle_ids_x_lt_0)
        center_of_mass_right = self.get_center_of_mass(particle_pos, particle_mass, particle_id_to_index,
particle_ids_x_geq_0)
        self.data["time"] = np.append(self.data["time"], time)
        self.data["pos_left"] = np.vstack([self.data["pos_left"], center_of_mass_left])
        self.data["pos_right"] = np.vstack([self.data["pos_right"], center_of_mass_right])

    def save_data(self) -> None:
        '''Save center of mass data to a JSON file.'''
        data_to_save = {
            "time": self.data["time"].tolist(),
            "pos_left": self.data["pos_left"].tolist(),
            "pos_right": self.data["pos_right"].tolist()
        }
        with open(self.settings["path_to_data_file_to_save"], 'w') as f:
            json.dump(data_to_save, f, indent=4)
        print(f'Data saved to {self.settings["path_to_data_file_to_save"]}')

    def get_center_of_mass(self, particle_pos, particle_mass, particle_id_to_index, particle_selected_ids) -> np.ndarray:
        '''Calculate the center of mass for the selected particles.'''
        particle_selected_indices = particle_id_to_index[:, 0][particle_selected_ids]
        particle_selected_pos = particle_pos[particle_selected_indices]
        particle_selected_mass = particle_mass[particle_selected_indices]
        total_mass = particle_selected_mass.sum()
        particle_selected_pos_weighted = (particle_selected_pos[:, :3] * particle_selected_mass).sum(axis=0)
        return particle_selected_pos_weighted / total_mass
```

Code Snippet 5 - `CenterOfMassAnalyzer.py`

46. Locating files and resources

All resources (images, models, scripts) and runtime libraries (.dll/.so) are located using the singleton class `agxIO::Environment`. Currently there are two separate types of files:

- Runtime files: .dll (WIN32), .so (Unix), .dylib (OSX)
- Resource files: (.png, .obj, .png, .cfg, .agxPy, .schema, .agxKernel)
- Textures and models: (For example .png, jpg, .obj, .stl etc.)

The path where these files can be found can be either controlled using environment variables as specified in Section 47, or programmatically using the *agxIO::Environment class*. This should be done before creating any other AGX object where file reading will be involved.

```
// Get a file path container to where runtime files are searched
agx::FilePathContainer& runtime_path =
  agxIO::Environment::instance()->getFilePath(
      agxIO::Environment::RUNTIME_PATH );

runtime_path.clear(); // Remove all previous settings
runtime_path.pushbackPath( "MyPath/plugins" );

// Try to find a file in the specified path
std::string filename = runtime_path.find( "plugin.dll" );
```

Same procedure can be done for RESOURCE_PATH to specify in what directories images, models, scripts and other non-runtime resources should be searched for.

```
// Get a file path container to where resource files are searched
agx::FilePathContainer& resource_path =
  agxIO::Environment::instance()->getFilePath(
      agxIO::Environment::RESOURCE_PATH );

resource_path.clear(); // Remove all previous settings
resource_path.pushbackPath( "MyPath/data/images;MyPath/data/models;" );
resource_path.pushbackPath( "MyPath/data/textures" );

// Try to find a file in the specified path
std::string filename = resource_path.find( "heightfield.png" );
```

In the windows version ';' should be used to separate paths, and under UNIX the ':' letter is used. Paths can be absolute or relative.

47. Environment variables

Values within [] is the default value.

Table 47.1 Environment variables controlling logging behavior

Environment variable	Description
AGX_LOG_OVERWRITE = [0]/1	Specify whether the log file will be overwritten, or if new files should be generated as AGX_LOG_FILE.0001 .0002, …
AGX_LOG_ENABLE=1/[0]	Specify whether the messages will be logged to a file or not.
AGX_LOG_LEVEL =0/[1]/2/3	Specify the level for which messages will be logged to file.
AGX_LOG_FILE= agx.log	Specify the path to the log file that should be generated.
AGX_OUTPUT_LEVEL=0/[1]/2/3	Specify the level for which messages will be logged to the console.
AGX_THROW_ON_ERROR= [1]/0	Should an exception be thrown when an error message is written?
AGX_BREAK_ON_ERROR=[0]/1	Should a breakpoint be set when an error message is written? This only works in DEBUG build in VisualStudio.

Table 47.2 Other environment variables

Environment variable	Description
AGX_FILE_PATH	Specifying where resource files will be found (models, images, scripts…)
AGX_PLUGIN_PATH	Specifying where plugins can be found.
AGX_PROCESS_PRIORITY= [2]/0..4	0 is lowest, 4 is real time priority. 2 is default priority. Works in Win32 only
AGX_PROCESS_AFFINITY=[0]/1	Specify whether the main thread for agxSDK::Simulation will be restricted to run on the current executional unit, or if it is allowed to be scheduled by the OS. Works in Win32 only
AGX_STATISTICS_ENABLE=[0]/1	Enable writing statistics information to a file specified with AGX_STATISTICS_PATH
AGX_STATISTICS_PATH= <filepath>	Specifies the file to which the statistics information will be written if enabled.
AGX_STATISTICS_INTERVAL= <float>	Specifies how often statistics information will be written to file in seconds. 1==write once every second.

AGX contains a mechanism for logging messages to file and/or console which is implemented through two classes, **agx::Logger** and **agx::Notify**. There are four different levels of messages:

Table 47.3 Values for Notify level

Value	Meaning	Description
0	Debug	Use for low level debug/test messages, seldom displayed
1	Info	Use for information of loading files, finding data, etc, shown by default
2	Warning	For warnings that occur.
3	Error	For fatal errors. Error can also throw std::runtime_exception() if it is enabled.

48. Python Scripting

- AGX Python Installation and requirements
- Modules and Library structure
- Python/AGX Dynamics integration
- Using Python for modelling in C++
- AGX Python coding guide
- C++ to Python guide
- Documentation of agxPythonModules

Python (www.python.org) is a powerful and popular scripting language for scientific computing and engineering. The main reason behind this is the the vast number of libraries, toolkits and SDK's available.

The AGX Python library comes as SWIG-generated modules and extensions, mirroring the C++ native API as far as possible. Each C++ namespace that is part of the AGX C++ SDK and exposed to Python makes up a Python module.

48.1. AGX Python Installation and requirements

This section will give you instructions on how to use and integrate AGX Dynamics with Python.

48.1.1. Requirements

Note

- For Windows, AGX Dynamics comes with an integrated/embedded version of Python. This is to ensure that it runs in a controlled/verified environment. This is due to the sensitivity between different versions of Python.
- This AGX version is built against Python version: 3.12.10

At the final stage of the installation of AGX Dynamics, a question related to installing Python requirements was asked. By default this is enabled and all the required modules where installed. If not, you need to make sure you have all the required modules installed before running some of the scripts that comes with AGX Dynamics. This is also true if you plan to use a custom Python installation instead of the one that comes with the AGX Dynamics installer.

For AGX-supported Linux x64 distributions, an installed *developer package* is required, which can be installed using the command:

```
$ sudo apt-get install python3.10-dev
```

48.1.2. Using a custom Python installation

This section describes how to use AGX Dynamics with an existing python installation. To make sure your existing python installation works you need to verify that:

- It is the same version of Python that was used when building AGX Dynamics: 3.12.10. You can also check which version AGX is built against using: `agxViewer --version`
- That it uses the same architecture as the current AGX Dynamics build: x86/x64

Python can be downloaded at: Python 3.12.10

If you want to use your own local Python installation, there are a few ways of achieving this:

1. In the START menu (windows) there are two shortcuts to the AGX Dynamics Command Window

```
AGX Dynamics Command Window - Will create a cmd window with environment setup
for AGX to use the (default) embedded Python that comes with AGX.
```

```
AGX Dynamics Command Window (External Python) - Will create a cmd window
with environment setup for AGX to use an external Python installation.
```

2. or call the file setup_env.bat with the argument: */USEEXTERNALPYTHON*

```
c:\program files\Algoryx\AGX-2.40.0.0\setup_env.bat /USEEXTERNALPYTHON
```

3. Set the environment variable AGX_USE_EXTERNAL_PYTHON before executing setup_env.bat (in the AGX folder)

```
set AGX_USE_EXTERNAL_PYTHON=1
cmd> c:\program files\Algoryx\AGX-2.40.0.0\setup_env.bat
```

You will also have to make sure you have your python environment correctly setup:

```
set PYTHONHOME=c:\Program Files\Python312
```

Tip

Tip: If you are unsure where Python is installed, type 'where python' in a command prompt.

48.1.3. Installing required libraries

Some tutorials might use additional packages such as plotting etc. Because of licensing, these modules cannot be bundled with AGX Dynamics but must be downloaded to the local python installation using the `python -m pip` command.

The file *<agx-dir>/data/python/requirements.txt* contains the required modules (and versions) that some scripts need to run. On Windows:

Use –user if you want the libraries to be installed for your current user only. If you use the pre-installed Python that comes with AGX Dynamics, you might want to skip the –user argument. This will install the libraries to: *<agx-dir>python-x64libsite-libraries* hence it will not affect your system installed Python version.

```
c:\Algoryx\AGX-2.40.0.0> python -m pip install --user -r data\python\requirements.txt
```

On Unix:

```
$ pip install --user -r data/python/requirements.txt
```

On Mac you need to use a virtual env:

```
$ python3 -m venv path/to/venv
$ source path/to/venv/bin/activate
$ python3 -m pip install -r data/python/requirements.txt
```

48.1.4. Environment

To be able to use AGX Python you need to make sure that the environment is correctly setup. By executing *<agx-dir>/setup_env.bat (.bash)* you ensure that the required environment variables are set-up. If you do not use setup_env, then make sure you investigate what is required to use AGX Python. The two important environment variables are:

- **PYTHONHOME** - Specifies where python can be found on your computer

If you use a custom version of Python you also need to make sure that PYTHONPATH is updated:

- **PYTHONPATH** - List directories with python modules, including AGX Python

Note

When using the embedded python that comes with the Windows installer, the PYTHONPATH is ignored in favour of the .pth file that you can find in the python-x64 directory.

48.1.5. Python IDE for development

There are many tools and IDE:s for developing Python code. Below are a few listed with short information how to use them with AGX Dynamics

48.1.5.1. Visual Studio code

Visual Studio Code is a text editor with support for many file formats, including Python.

1. Download and install Visual Studio Code
2. Start an AGX Dynamics Command Prompt (from the START menu or by starting a cmd windows and execute the *<agx-dir>/setup_env.bat* to setup the execution environment (it is important so that AGX Dynamics files can be found in PATH and that the python modules can be located: PYTHONPATH).
3. Start Visual Studio Code from the command-line with the correct environment setup.

4. Open a .py or .agxPy file in the *data/python/* directory.
5. You might want to associate the .agxPy file format to Python script. (see documentation on the Visual Studio Code site for how to do this)

48.1.5.2. PyCharm

Another Python development environment is PyCharm Below you can find a short tutorial on how to install and setup PyCharm for editing AGX Python files under the Windows platform.

1. First you need to download and install PyCharm
2. Start an AGX Dynamics Command Prompt (from the START menu or by starting a cmd windows and execute the *<agx-dir>/setup_env.bat* to setup the execution environment (it is important so that AGX Dynamics files can be found in PATH and that the python modules can be located: PYTHONPATH).
3. Start PyCharm. Depending on where you installed PyCharm the path might be different:

```
D:\Algoryx\AGX-2.28.0.0> "c:\Program Files (x86)\JetBrains\PyCharm Community Edition 2016.3.1\bin\pycharm.bat"
```

4. Name and create a PyCharm project.
5. Go to Settings->Project: <project name>->Project Structure and under *Add Content Root* add the *<agx-dir>/bin/x64/agxPy* directory in the AGX Dynamics installation.
6. Create a file and start coding:

```
import agx
import agxSDK
init = agx.AutoInit()
sim = agxSDK.Simulation()
sim.stepForward()
```

48.2. Modules and Library structure

Each C++ namespace of AGX exposed to Python is exposed as its own Python module available for import. For example: agxSDK is encoded in _agxSDK.pyd. These modules are found together with their respective native extension shared library in:

- In Windows, beneath the directory of your AGX installation: *<agx-dir>/bin/x64/agxpy*
- In Unix the default path should be something similar to: */opt/Algoryx/AGX-<VERSION>/bin/agxpy/* This might be different on your specific Unix flavor.

For Python to be able to locate the AGX Python modules, you might need to set the PYTHONPATH environment variable. See the chapter Environment below.

You can verify the AGX Python installation by creating a Python script that contains the following:

```
import agx
print(agx.__version__)
```

If this works, then Python can locate the required libraries for AGX Python.

When using AGX, the library has to be initialized and then later shutdown as described in the getting started section of the user guide. AGX has the same requirement when being used from Python as from C++. As a convenience helper, the *agx* Python module takes care of *init* and *shutdown* automatically.

This makes usage of *agx.init()* or *agx.AutoInit()* optional in user Python scripts. Manual usage of these init-methods does not cause problems for the automatic helper and does not have to be removed from existing scripts.

48.3. Python/AGX Dynamics integration

Tip

File script *<agx_dir>/data/python/template.agxPy* works as a starting point for a new script using the AGX Dynamics Python API.

AGX Dynamics and Python can be used together in two different ways:

1. In the *native* Python scripting environment using the command: *python*
2. Embedded as a scripting format for the *agxViewer.exe* application

Both ways are simple to support from the same Python application or library, as long it is initialized correctly. Aside from importing the required Python modules used by the program, the following bare-bone example will determine how the script is being executed; whether its embedded mode from

agxViewer or in a native Python environment:

```
import agx
import agxSDK
import agxIO
# import of additional modules ...
import agxPython
init = agx.AutoInit(); # Important so that we initialize the AGX SDK

def buildScene1(sim, app, scene_root):
    # build your AGX scene here by adding to the agxSDK.Simulation
    # object sim, aagxOSG.ExampleApplication object app and
    # the agxOSG.Group object scene_root.
```

Below we have the entry point for *agxViewer*, or an application based on `agxOSG::ExampleApplication`. It will look for the function `buildScene()` and execute its content.

```
def buildScene():
  # Script is run from agxViewer. Everything is set up, so
  # all we need to do is fetching our sim, app and osg root node
  # and call the function which builds our scene
    sim = agxPython.getContext().environment.getSimulation()
    app = agxPython.getContext().environment.getApplication()
    scene_root = agxPython.getContext().environment.getSceneRoot()

    # That is all we need before we teleport...x
    buildScene1(sim, app, scene_root)
```

Below, we have the entry point for when the script is executed with the *python* executable. It will find that the `agxPython.getContext()` return `None` and call the main function.

```
def main(fileName):
    # Initialize the AGX ExampleApplication here (see any script in data/python for specifics)
    app = agxOSG.ExampleApplication()

    # Create a command line parser.
    argParser = agxIO.ArgumentParser()
    argParser.readArguments(arg) ## feed the parser with the arguments

    # Add a scene to the application. Whenever the key '1' is pressed, this file will be reloaded
    # and the def "buildScene" is called
    app.addScene(fileName, "buildScene1", ord("1"))

    # Call the init method of ExampleApplication
    # It will setup the viewer,windows etc.
    if app.init(argParser):
        initCamera(app)
        app.run()
    else:
        print("An error occurred while initializing ExampleApplication.")


# Test to see if we are running via the native python interpretator or in embedded mode (agxViewer)
if agxPython.getContext() == None:
  # Script is run from native Python interpreter, call main with the script file name
  # as argument
    main(sys.argv[0])
```

48.4. Using Python for modelling in C++

It is also possible to use python for modelling and C++ for control. The tutorial `tutorial_python_excavator_controller.cpp` that can be located among the C++ tutorials illustrates this.

1. Construct the model in Python, including parameterization
2. Load the python script from C++.

3. Access constraints, bodies etc. from C++.
4. Control the simulation from C++.

In this way you can get the best of both worlds:

- The fast iteration/flexibility of programming with Python without compilation/linking.
- Low performance overhead because no code is actually running in Python during the simulation.

So this is a good way of mixing the two worlds.

When running the tutorial_python_excavator_controller you can reload the python script by pressing the button '1' just as with the agxViewer application.

48.5. AGX Python coding guide

48.5.1. Sharing AGX buffers as numpy arrays

Internally AGX entities are stored in contiguous memory buffers. Looping through these buffers using the AGX C++ API is very fast. By creating numpy arrays, that points at the same memory as the AGX buffers, it is possible to inspect and change the AGX buffers using all the usual numpy operations, with comparable performance as the C++ API.

One danger, is that every time AGX is in control it can decide to reallocate the buffer. From python the user is never notified about this. Therefore, the user must recreate the numpy array between every timestep. This process is made more convenient by using the `BufferWrapper` python class implemented in `agxPythonModules`. The wrapper finds the specified buffer and creates a numpy array with the correct type and size that wraps around it. Every time the numpy array is accessed from the wrapper by the user, the wrapper class recreates the array around the buffer.

```
from agxPythonModules.utils.numpy_utils import BufferWrapper

# ...

# Get the position buffer for all the particles in sim
position_buffer = BufferWrapper(sim, "Particle.position")

position_array = position_buffer.array

# Get the shape of the array
shape = position_array.shape

# Set the z-position of every particle
position_array[:,2] = np.linspace(-10, 10, shape[0])

# step agx
sim.stepForward()

# re-fetch buffer in case agx has reallocated
position_array = position_buffer.array

# Print the z position for every particle.
print(position_array[:,2])
```

48.5.2. Proxy classes, their instances and reference counting

All C++ classes exposed to python will be accessible through a proxy class. Each namespace of AGX will belong to a separate Python module.

For example the class `agx::RigidBody` will be located in the module _agxPython which is imported to Python using the *import* keyword. Below is an example of how to import the agx namespace and instantiate a class in that namespace:

```
import agx
rb = agx.RigidBody()
```

C++ classes derived from agx::Referenced are automatically handled by Python, so that as long as there is a C++ `agx::ref_ptr` *and/or* a Python object referring to it, it will not be deallocated.

48.5.3. Attributes

Attributes added to proxy class instances in Python does not exist outside the Python object to which the attributes belong, and can't pass through the native C++ layer back to Python again. The following code demonstrates this:

```
rb_vector = agx.RigidBodyVector()
rb = agx.RigidBody()
rb.my_attr = 5.0 # An attribute added to the rb object
rb_vector.append(rb)

rb_retrieved = rb_vector[0]
print(rb.my_attr)                    # Will print 5.0
print(rb_retrieved.my_attr)   # Will throw AttributeError exception! Because rb_retrieved is
                                                          # not the same Python object as rb
```

The reason for this is that: *Proxy class instances, or Python objects of proxy classes, are not exact representations of C++ objects and multiple Python proxy objects may reference the same C++ object.*

48.5.4. Function/method overload ambiguity

In Python it is forbidden to declare anything using the same identifiers or names in any scope where they already exist. In C++ however, it is commonplace to overload functions and methods by their signatures while sharing the same names. In AGX Python, proxy classes only have one representation for all possible versions of static functions or methods found in C++.

Ambiguity resolving is done within the Python extensions before a match is found based on which arguments the Python code passed to it during a call, where no match throws a NotImplementedError exception back to the caller.

48.5.5. Inherited classes/virtual methods

Some AGX classes allow for their virtual methods to be overridden in derived classes, where EventListener family of classes being the most relevant example of this. The table below show examples of classes which can be implemented in Python:

Python Class	Virtual Methods (in Python)
agxSDK.EventListener	`def addNotification(self) -> "void"` `def removeNotification(self) -> "void"`

agxSDK.ContactEventListener	`def preCollide(`
	`  self,`
	`  t: "float"`
	`) -> "void"`
	
	`def pre(`
	`  self,`
	`  t: "float"`
	`) -> "void"`
	
	`def post(`
	`  self,`
	`  t: "float"`
	`) -> "void"`
	
	`def last(`
	`  self,`
	`  t: "float"`
	`) -> "void"`

agxSDK.GuiEventListener	`def mouseDragged(`
	`  self,`
	`  buttonMask: "MouseButtonMask",`
	`  x: "float",`
	`  y: "float"`
	`) -> "bool"`
	
	`def mouseMoved(`
	`  self,`
	`  x: "float",`
	`  y: "float"`
	`) -> "bool"`
	
	`def mouse(`
	`  self,`
	`  buttonMask: "MouseButtonMask",`
	`  state: "MouseState",`
	`  x: "float",`
	`  y: "float"`
	`) -> "bool"`
	
	`def keyboard(`
	`  self,`
	`  key: "int",`
	`  modKeyMask: "int",`
	`  x: "float",`
	`  y: "float",`
	`  keyDown: "bool"`
	`) -> "bool"`

Python Class	Virtual Methods (in Python)

	`def update(`
	`  self,`
	`  x: "float",`
	`  y: "float"`
	`) -> "void"`

agxSDK::ContactEventListener	`def impact(`
	`  self,`
	`  time: "double",`
	`  geometryContact: "agxCollide::GeometryContact"`
	`) -> "KeepContactPolicy"`
	`def contact(`
	`  self,`
	`  time: "double",`
	`  geometryContact: "agxCollide::GeometryContact"`
	`) -> "KeepContactPolicy"`
	`def separation(`
	`  self,`
	`  time: "double",`
	`  geometryPair: "agxCollide::GeometryPair"`
	`) -> "void"`

agxIO::JoystickListener	`def axisUpdate(self,`
	` state: "agxIO::JoystickState",`
	` axis: "int",`
	`) -> "bool"`
	
	`def buttonChanged(`
	` self,`
	` state: "agxIO::JoystickState",`
	` button: "int",`
	` down: "bool"`
	`) -> "void"`
	
	`def povMoved(`
	` self,`
	` state: "agxIO::JoystickState",`
	` pov: "int"`
	`) -> "void"`
	
	`def sliderMoved(`
	` self,`
	` state: "agxIO::JoystickState",`
	` slider: "int"`
	`) -> "void"`
	
	`def addModification(`
	` self`
	`) -> "void"`
	
	`def removeModification(`
	` self`
	`) -> "void"`

It's important to not forget about the 'self' argument when you override any of these methods, and that the right type of any value they return is used. Violations to this are considered an error and will break your script.

Furthermore, it's important to follow Python rules as well, something known as "Zen of Python", and call the base class version of the method being overridden when you wish to return some default value. In Python, you do not wish to "override" base class functionality with polymorphic inheritance as you do in C++, instead you "enhance" with new functionality by adding instead of replacing.

48.5.6. Scope locality and garbage collection

In Python, all declarations made in some scope are also bound to that scope once evaluated. In other words, the default scope used in Python is "local". Assigning or returning a value binds them to whatever object or scope that accepts the value, and the value will persist in memory for as long it is strongly referenced by another object or scope. If no such references exist for a value it gets garbage collected. Whether this happens right away or some time later can vary, but it is safe to assume it mustn't be touched by any code once it becomes garbage.

Reference-counted C++ objects complicate things sometimes if care is not given to avoid premature decrement reference counts of references not owned by the dereferencer. Always keep a reference around for AGX objects you create by keeping a Python object/proxy class instance alive for as long you want to be on the safe side.

48.5.7. Ref smart pointer objects

Sometimes it's not always a raw pointer object you deal with but their smart pointer counterparts. The smart pointer template class instances are suffixed with Ref, so for example smart pointer class for the Geometry class is called GeometryRef and for RigidBody, it's RigidBodyRef. Normally they expose the same methods as the original class with the addition of those declared by `agx::ref_ptr<T>` template class, such as get(), isValid and so on.

It is not, however, possible to pass them as arguments to methods expecting objects of template argument type; e.g. if a Geometry is expected, passing GeometryRef instead is not valid. Sometimes you deal with <Class>RefVectors, and any accessors will inevitably always return <Class>Ref objects - not

<Class> objects.

To dereference these smart pointers, use the get() method on them, but check if they are valid before you do, or else you risk the execution of dereferencing null pointers within the C++ layer which will crash your application. Here's an example where we define a function which extracts all the geometries associated with a rigid body and returns a Python list with raw Geometry objects:

```
def rigidBodyGeometriesToList(rb):
 geometries = rb.getGeometries() # geometries is of type GeometryRefVector
 geometries_list = list()        # the list to return
 for geo_ref in geometries:      # geometries contain items of type GeometryRef
     geometries_list.append(geo_ref.get()) # geo_ref.get() gives us its Geometry
 return geometries_list            # geometries_list now contain Geometry items
```

48.6. C++ to Python guide

Most primitive types, classes, functions and methods of the AGX C++ API exposed to Python have wrappers which work as one would expect in Python. For example, wherever a C++ NULL or nullptr would be used, you use None in Python. There are however some things to watch out for, especially when the philosophical and conventional similarities between C++ and Python end.

Below is a number of sample cases demonstrating the syntax difference between Python and C++:

48.6.1. Construction

C++:

```
agx::RigidBodyRef body = new agx::RigidBody();
```

Python:

```
body = agx.RigidBody()
```

48.6.2. Calling a normal method

C++:

```
body->getMassProperties()->setMass( 10 );
```

Python:

```
body.getMassProperties().setMass( 10 )
```

48.6.3. Calling a static method

C++:

```
agx::Constraint::calculateFramesFromBody(agx::Vec3(1,2,3), agx::Vec3(1,0,0),
                                                                  body1, frame1,
                                                                  body2, frame2);
```

Python:

```
agx.Constraint.calculateFramesFromBody(agx.Vec3(1,2,3), agx.Vec3(1,0,0),
                                                                body1, frame1,
                                                                body2, frame2);
```

48.7. Documentation of agxPythonModules

This section documents AGX python modules that are located in `data/python/modules/agxPythonModules/`.

48.7.1. Replace particleEmitter with rigidBodyEmitter

ParticleEmitters can only emit spherical bodies that belong to a particleSystem, while a rigidBodyEmitter can emit arbitrary rigid bodies that are treated like any other rigid body in the simulation. Rigid bodies are generally more computationally expensive than particles in a particleSystem, but useful for modeling

granular systems of rigid bodies. A common workflow is to make prototype simulations of the system with particleEmitters and particleSystems, to figure out the rough 3D design and control of the system, and then substitute the particleEmitters with rigidBodyEmitters. Making the modeling cycle of the system more effective. This requires a systematic way of replacing an **agx::ParticleEmitter** with an **agx::RigidBodyEmitter**.

The script replaceParticleEmitterWithRigidBodyEmitter.py in the directory `data/python/modules/agxPythonModules/utils`, takes a serialized AGX simulation as input and replaces all the particle emitters in the simulation with rigid body emitters and saves the resulting simulation as a new .agx file. After setup of the AGX environment variables the utility script can be called from the command line with:

```
python -m agxPythonModules.utils.replaceParticleEmitterWithRigidBodyEmitter <path/to/simulation.agx>
```

By default the resulting simulation is saved as output.agx. It is possible to specify the output name with command line argument `--output <name/of/output.agx>`. You can then run the simulation and record a journal with the command `agxviewer output.agx --journalRecord --journalIncrementalStructure`.

The default geometries used as templates are taken from the `data/models/convex_stones` directory. It is possible to create and use a new shape library, by using the `--shapeLibrary` command line argument. The shape library is a directory with one .obj file per shape. The shapes must be convex. The size of the rigid body template are estimated using the diagonalized inertia tensor, of the rigid body, assuming uniform mass distribution and a cuboid shape:

$$x = \sqrt{6m^{-1}(Izz + Iyy - Ixx)}$$
$$y = \sqrt{6m^{-1}(Izz + Ixx - Iyy)}$$
$$z = \sqrt{6m^{-1}(Iyy + Ixx - Izz)}$$

Choosing the smallest of x, y, z as the size is a good approximation of how mechanical sieving measures the size of rocks. The distributionTable of the rigidBodyEmitter is created by scaling the rocks to match the size of the previously used particles.

The material of the rigid bodies are copied directly from the particles materials. This can lead to unexpected changes in the bulk granular flow, since the rigid bodies have different geometry. Recalibration of the material properties can be necessary.

The friction model for rigid bodies in AGX is by default IterativeProjectedConeFriction, with the solve type **SPLIT**. Since these simulations often contains a large contact system the script changes the solve type to **ITERATIVE**. This trades realistic friction for performance. Depending on the application, it possible to speed this up even more together with agxSDK::MergeSplitHandler - AMOR.

A step-by-step description of the workflow when modeling granular flows with spherical particles and rigid bodies:

1. Do prototype simulations with spherical particleSystems and particleEmitters.
2. Export the simulation as an .agx file.
3. [Optionally] Create your own shape library. One shape per .obj file.
4. Run the script `replaceParticleEmitterWithRigidBodyEmitter.py` to replace particleEmitters with rigidBodyEmitters.
5. Run the new simulation with correct solver settings and journal configuration.
6. Post-process the results.
7. If needed, return to point 1 and re-design the system.

49. C#/.NET bindings of the AGX API

Some versions of AGX Dynamics will come with libraries which export the AGX Dynamics API to the .NET development environment (currently version 4.6.1). By utilizing swig (https://github.com/swig/swig), a large portion (and growing for each version) of the AGX API is exposed to use in .NET. This means that you can utilize the development environment from Microsoft and build C#/VB-based applications with physics from AGX.

The .NET wrapper comes with three layers of run time libraries. First is the C++ wrapper:

Library name	Description
agxDotNetRuntime.dll	AGX C++ .NET binding

Then the top layer which implements the actual .NET wrapper in C#:

Library name	Description
agxDotNet.dll	Interface library for using AGX in .NET

Then the math library:

Library name	Description
agxMathDotNet.dll	Math library implemented in C# for bridging between AGX and .NET

`agxDotNet.dll` and `agxMathDotNet.dll` are the interface libraries to be referenced in your typical .NET application. The C++/.NET binding library `agxDotNetRuntime.dll` will be automatically located according to the rules by which Windows locates dynamic libraries.

49.1. Functions

Since there are no free functions in C# (functions which are not members of a class), only classes with namespaces, the free functions that are available in the C++ API will get a somewhat different look. They will be added into an extra namespace (class). For example, the function `agx::init()` : As this namespace (agx) resides in the agxDotNet plugin, it will in C# be accessible as:

```
agx.agxSWIG.init();
```

This convention goes for all (class-less) functions found in the AGX API.

Also, some extensions methods are used to bridge `agxDotNet.dll` and `agxMathDotNet.dll`, which can be used like this:

```
using agx.extensions;
```

49.2. Sample application using the C# binding

49.2.1. testApplication

Below is a short code snippet, illustrating various API calls in the .NET version of the AGX programming API in C# code. For more examples, look into the directory: agx\swig\configuration\agxDotNet\testApplication

```
using agx.extensions;

class Program
{
 static void Main(string[] args)
 {

   // Try to use the Environment class.
   agxIO.FilePathContainer fp = agxIO.Environment.instance().getFilePath();
   string script = fp.find("data/python/tutorials/tutorial1_graphics.agxPy");
   Console.WriteLine("Script {0}", script);

   // This must be the first call to AGX (except for setting up the FilePath).
   // This is because the call to agx.init() (or in .NET: agx.agxSWIG.init())
   // will try to load plugins and
   // various resources. You have to specify where AGX can find these plugins,
```

```
    // license file etc.
    // See the user manual for more (important) information about this.
    //
    agx.agxSWIG.init();
    agx.agxSWIG.setNumThreads(3);

    {
      agxSDK.Simulation sim = new agxSDK.Simulation();
      sim.setDynamicsSystem(new agx.DynamicsSystem());

      agx.RigidBody body = new agx.RigidBody();
      body.setName("this is a body");
      body.setPosition(4, 0, 0);
      sim.add(body);

      var frame = new agx.Frame();
      frame.setLocalTranslate(2, 0, 0);
      var hinge = new agx.Hinge(body, frame);
      hinge.getMotor1D().setEnable(true);
      hinge.getMotor1D().setSpeed(0.1);
      sim.add(hinge);

      agx.RigidBodyRefVector bodies = sim.getDynamicsSystem().getRigidBodies();

      sim.stepForward();

      for (int i = 0; i < bodies.Count; ++i)
      {
        var bodyName = bodies[i].getName();
        Console.WriteLine("A body: " + bodyName);
      }

      agx.TimeGovernor tg = sim.getDynamicsSystem().getTimeGovernor();

      sim.remove(body);

      sim = null;
      body = null;
    }
    GC.Collect();
    GC.WaitForFullGCComplete();
    agx.agxSWIG.shutdown();
  }
}
```

49.2.2. CSharpViewer

CSharpViewer is located in the directory `swig\SWIGDotNet\CSharpViewer`. It is an example application which utilizes the agxOSG::ExampleApplication to create a graphics window and handle input like the agxViewer application.

With this application you can load a .agxPy script or a .agx file. All hinges and prismatic joints will be listed with a slider for each constraints's Lock1D()/position controller. The range for each constraint will be set to [-PI,PI] for hinges and [-0.1,0.1] for prismatics. With the slider you can then control each constraints target position.

The sample application shows how to:

- Create an application
- Access objects in the simulation
- Run and step the simulation through user events

This application can be built as a separate project:

- Go to the directory
- Start cmake-gui .
- Choose correct build environment
- Press Configure, Generate

- Start visual studio with: devenv /useenv
- Build the project. CSharpViewer.exe is the final result.

49.3. C#, SWIG and garbage generation

The C# bindings made by SWIG are created so that there are C# classes matching the C++ classes. For example there is a C# RigidBody class that matches the C++ agx::RigidBody. The SWIG-created C# RigidBody implementation is mostly a proxy object that holds a pointer and ownership information about the C++ RigidBody.

When methods in C# are called on the proxy object, P/Invoke is utilized to call native wrapper code generated by SWIG that converts the C# IntPtr held by the proxy object to the correct C++ pointer type and then calls the desired C++ code.

The flow from managed code in `agxDotNet.dll`, to native wrapper code in `agxDotNetRuntime.dll` to native agx and back is handled automatically and normally not something the user has to think about.

There is one drawback, when accessing C++ classes, C# proxy objects are generated and when those proxies are no longer needed, they will be garbage collected. The garbage collection can then cause unwated spikes and performance artifacts.

The AGX C# bindings has two different solutions to work around this problem with short lived proxy classes:

- Simple datatypes such as e.g `agx.Vec3` and `agx.AffineMatrix4x4` are implemented as structs in C# and data marshalling is performed to convert to/from C++ datatypes to completely avoid proxy classes on the C# side.
- Objects owned by C++ that are likely to be accessed often have had their proxy classes changed so that they are fetched from an object pool and can be returned to the object pool when no longer needed.

To revisit part of the short example above:

```
agx.RigidBodyRefVector bodies = sim.getDynamicsSystem().getRigidBodies();

for (int i = 0; i < bodies.Count; ++i)
{
  var bodyName = bodies[i].getName();
  Console.WriteLine("A body: " + bodyName);
}
```

Each iteration in the loop will create garbage. `bodies[i]` will lead to the creation of a `agx.RigidBodyRef` which will shortly there after be seen as garbage.

A more efficient way to perform the same thing would be

```
agx.RigidBodyRefVector bodies = sim.getDynamicsSystem().getRigidBodies();

for (int i = 0; i < bodies.Count; ++i)
{
  var bodyI = bodies[i];

  var bodyName = bodyI.getName();
  Console.WriteLine("A body: " + bodyName);

  bodyI.ReturnToPool(); // bodyI should be considered to be null now
                        // and no further methods should be invoked on the instance
}
```

The datatypes that are made poolable are:

- `agx.RigidBodyRef`
- `agx.ConstraintRef`
- All constraints (Constraint, DistanceJoint, …)
- All constraint controllers (Speed1D, Lock1D, Range1D, …)
- `agxCollide.GeometryRef`
- `agxCollide.ShapeRef`
- All collision shapes (Box, Sphere, Mesh, …)
- Contact data datatypes: * `agxCollide.GeometryContact` * `agxCollide.ContactPointVector` * `agxCollide.ContactPoint`

All the classes that are poolable implements the `agx.IPoolable` interface. That interface requires that three different methods should be implemented and of those three methods, only `ReturnToPool` should be used by users. The other two are needed by the ObjectPool to reconfigure the proxy object so that it can wrap a different C++ object without generating garbage.

If one needs to get access to one of the ObjectPool singletons that are created automatically in the background, one can access them via `namespace.classname.GetObjectPool()`, for example `agx.RigidBodyRef.GetObjectPool()`.

The Object Pool has a property `MaxPoolSize` which controls the maximum number of empty proxies the pool should hold. The Object Pool also makes it possible to create empty proxies in advance via `PreAllocate( int count )`.

Normally, there should not be a big need to access to object pool directly, the important thing to remember is to call `ReturnToPool()` for poolable types when one is done with them.

49.4. Passing large Vectors to AGX

The C# interfaces for `agx::Vector`, `agx::VectorPOD` are wrapped by SWIG so that they should work in a similar way as C# Collections. For most operations this adds convenience and familiarity.

When passing large amounts of data, there is a performance downside due to the managing of data between C#/managed C++/unmanaged C++. This typically occurs for operations such as updating the heights of a `agxTerrain.Terrain` or a `agxCollide.HeightField` from C#.

Using the default functionality to set all elements in a `agx.VectorPOD` would perform *managed -> unmanaged -> managed* transitions for each element in the container. To resolve this problem and provide an efficient way to set the entire vector, the C# interface for AGX have augmented `agx::VectorPOD<T>` -based classes that use int- och float-based data types. The interface contains a method `Set(T[] array)` and a corresponding `Get(T[] array)` where the datatype T matches the VectorPOD being wrapped. Using these methods only *one* single *managed -> unmanaged -> managed* transition will be done when setting the entire vector instead of one for each element.

Below is a recommendation how to use the set method with an `agx.RealVector`, although the process can be used for any wrapped `agx::VectorPOD` class such as `agx.Vec3Vector, agx.Vec2Vector, agx.UInt32Vector` etc.

1. Reuse an `agx.RealVector`
2. Compute/set the data values in a native C# array
3. Copy the data from the native C# array to the `agx.RealVector` with `real_vector.Set( csharp_array )`

The get method can be used in a similar way:

1. Reuse a native C# array but make sure that it has the same size as the `agx.RealVector`.
2. When you need to access the data in a `agx.RealVector`, copy the data from the `agx.RealVector` to the native C# array using `real_vector.Get(csharp_array)`

Note

While the Set method resizes the wrapped array to match the size of the C# array, the Get method will throw an ArgumentException if the sizes of the arrays do not match.

Below is a “complete” toy example using both the Set and Get methods:

```
// Create a C# array to copy into the agx.RealVector
double[] input = new double[100];
agx.RealVector agxVector = new agx.RealVector();

// Assign the elements in the input C# array
for ( int i = 0; i < input.Length; i++ )
  input[ i ] = (double)i;

// Copy the elements into the agx.RealVector
agxVector.Set( input );

// Create an output C# array with the same size as the agx.RealVector
double[] output = new double[agxVector.Count];

// Copy elements from the agx.RealVector into the C# array
agxVector.Get( output );
```

49.5. Building the SWIG binding

The sub-directory `<agx-install-dir>swig` contains everything needed to build the C#/.NET binding. For information of how to build the libraries, read the README.txt in the SWIG directory.

49.6. Extending the C# interface

It is possible to extend the .NET interface by adding more classes/header files to the .i files found in the `<agx>swigconfiguration` sub-directory. For more details about SWIG's interface files look at the SWIG documentation (https://github.com/swig/swig).

Adding another header file to SWIG can be done with the following steps:

1. Locate the .h file which contain the class you want to export to .NET
2. Identify which namespace the class/header file belongs to.
    1. For example agx::Constraint belongs to the *agx* namespace, then it should be included into agx.i
3. Edit the interface file (for example *agx.i*) and add two include directives:
    1. Locate the use of the INCLUDE() macro. Add your header file: INCLUDE(namespace/headerfile.h) which will be added to the generated .cpp code.
4. Follow the instruction in `swig/README.txt` to configure the wrapper project and build the binding libraries.
5. To test that the new class exists, add code in program.cs that references the new class.
6. When it works, copy the resulting dll files (`agxDotNet.dll` , `agxDotNetRuntime.dll` and `agxMathDotNet.dll`) to the path where you keep the AGX runtime libraries.

50. Matlab/Simulink plugin

AGX Dynamics has a plugin to Matlab and Simulink, which can be used in order to run AGX as a co-simulation. This is done via a predefined AGX-block which appears in the Simulink library after installation of AGX, and which can easily be modified to create and load your own AGX simulations.

Furthermore, a lower level Matlab coupling is provided.

The Matlab/Simulink plugin is limited to Microsoft Windows operating systems. It has been tested on Windows 10 64bit and Matlab version R2018b.

If your AGX-license does not include a license for the Matlab/Simulink plugin, then please contact Algoryx Simulation for an evaluation license.

Technically, the Simulink coupling is done via an S-function, and the Matlab coupling via a mex-function. Both need to be explicitly compiled as shown below.

Note

To properly run AGX Dynamics inside of Matlab/Simulink. You need to ensure the runtime environment variables is correctly setup. For more information see Section 5.2.1

50.1. Installation

Both the Simulink S-function and the Matlab mex-function source code must be compiled prior to using the AGX functionality inside Simulink/Matlab. All the source code can be found in the following directory:

```
<agx-dir>/data/Matlab/src
```

50.1.1. Manual Installation

During the installation process, the following file will be run to register the AGX path to Matlab's search path, and to compile the s-function and the mex file within Matlab:

```
<agx-dir>/data/Matlab/addAgXToMatlabAndSimulink.cmd
```

You can run this file manually (by double-clicking on it).

Attention

The install/uninstall scripts will ask for administrative rights to ensure that it can write in the AGX installation path.

In order to remove the AGX path from Matlab's search path and to remove the compiled s-function and mex file, the uninstaller will run

```
<agx-dir>/data/Matlab/removeAgXFromMatlabAndSimulink.cmd
```

Run this file manually (by double-clicking on it).

50.1.2. Compilation of s-function and mex file

When running the addAgXToMatlabAndSimulink.cmd script the Matlab mex-command will be called in order to compile the source files to the s-function and the mex-function.

These source files are:

Table 50.1 Source files for Matlab/Simulink plugin

Filename	Purpose
<agx-dir>/data/Matlab/src/agx_sfun.c	Code file for Simulink s-function.
<agx-dir>/data/Matlab/src/agx.c	Code file for Matlab mex-function.

The Matlab mex-command is not a compiler in itself, but rather calls a compiler which has been assigned to it. The first time the command is called (which might be now when you install AGX), a compiler has to be chosen from the Matlab prompt by following the instructions there.

A compiler called *lcc* comes with most Matlab versions on windows, and the two files named above compile fine with it.

Another possible, free compiler under Windows is Microsoft Visual C++ 2015, or its free community version. You might need to restart Matlab in order to find the newly installed compiler.

You can always change the default compiler for Matlab's *mex*-command by running **mex –setup** in the Matlab prompt and following the instructions there.

50.1.3. Manual compilation

The Matlab (agx.c) and Simulink (agx_sfun.c) code can manually be compiled from within Matlab with the following command:

```
>     cd <AGX-dir>/data/Matlab/src
>     mex agx.c
>     mex agx_sfun.c
```

Note

If you need to compile with debug information, use the mex –g command.

50.2. Limitation

Within one Matlab session, AGX can be used either only from Simulink using the s-function, or only from Matlab using the mex function. Using the two after each other or next to each other in the same Matlab session is not recommended.

50.3. Starting AGX from Simulink

After a successful installation, an AGX-library is going to appear in your Simulink library browser.

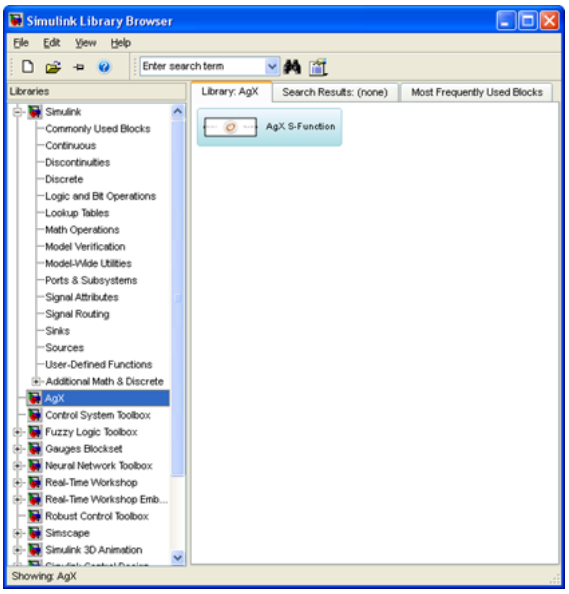


Fig. 50.1 AGX-block in Simulink library view.

Simply drag a block from there into your model and connect its input and output ports. The simulation is ready to run.

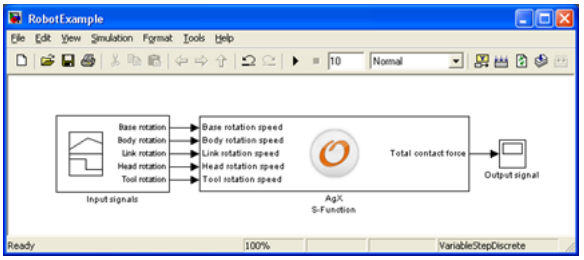


Fig. 50.2 AGX-block in model

Note

There can only be one AGX-block per model. You will get a warning when pressing play in Simulink if you have more than one AGX-block per model.

You can modify the block by double-clicking on it.

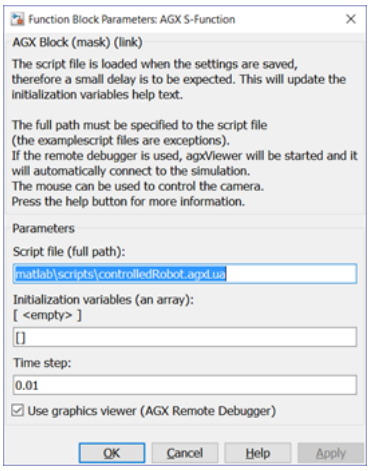


Fig. 50.3 Modifying the AGX block.

The following changes can be made:

- *Script file (full path):*

  Contains the AGX script-file: It is an .py/.agxPy (Python) script (a script file with AGX commands) which setup the simulation and the coupling between AGX and Simulink. To learn more, look at the sample scripts in <agx-dir>dataMatlabscripts.

- *Initialization variables:*

  An array of values to initialize the scene. The length and meaning of the scalars depend on the scene. Descriptions of the parameters are shown in the above the field. If the wrong number of arguments is given, all are set to zero (and a message about that will be sent to the Matlab prompt). Otherwise, after pressing 'Ok', the initialization output will be written to the Matlab variable agx_init_output.

- *Time step:*

  The simulation time step. AGX uses an internal fixed-size stepper whose time step you can set here.

- *Use graphics viewer (3D Window with the simulation visualization):*

  Enables a 3D viewer which render the simulation scene using agxViewer.exe. Using graphics might slow down the simulation.

  By checking this box, the viewer will run in the same process as Matlab. For stability reasons we instead strongly recommend that you instead start the viewer prior to Matlab.

  This will ensure the viewer run in a separate process. The AGX Dynamics block will automatically detect the running viewer and will not start another one. For more information see Section 50.5

Note

The Help-button gives not only useful help, but also links to a few premade examples which you might want to inspect!

After saving the settings in the block (via Ok/Apply), the AGX script-file will be reloaded. This is necessary in order to adapt the number of ports in the block to the ones in the new or modified script-file.

50.4. Starting AGX from Matlab

An AGX simulation can be controlled from Matlab using the

```
<result> = agx(<command>, arguments)
```

Available commands are:

Command	Description
load	Load and initialize a new simulation from a specified AGX script file.
step	Take one simulation step
visual	Enable/Disable the 3D rendering window

The syntax for loading an AGX-simulation in Matlab is:

```
>>  [sizeInitInput, initOutput, sizeStepInput, sizeStepOutput] = agx('load', <simulationFile>, <initInput>, <timeStep>)
```

where the inputs consist of:

- **<simulationFile>** is the name of a AGX script file containing the simulation data (For example: *<AGX-dir>/data/matlab/scripts/rotatingBox.py*),
- **<initInput>** is a row vector of size sizeInitInput (one might have to call the function once in order to get to know the size from the error message in case it is not previously known) and
- **<timeStep>** is the simulation time step, e.g. 0.01 for 100Hz.

If **<initInput>** does not have the right format, all init input parameters will be set to 0.

The outputs are:

- **<sizeInitInput>** gives the size of the initial input data which is evaluated once at scene startup,
- **<initOutput>** holds the initial output data from scene startup,
- **<sizeStepInput>** gives the size of the input data to subsequent agx('step' …) calls, and
- **<sizeStepOutput>** gives the size of the output data from subsequent agx('step' …) calls.

Example: *data/Matlab/scripts/bouncingSphere.py*

This example has a 3D vector as init input, the position of the sphere:

```
argA = SphereInitArgument(3)
agxSDK.SimulationControl.instance().addInitInputArgument("sphere", argA)
```

and a scalar for the restitution:

```
argB = RestitutionInitArgument(1)
agxSDK.SimulationControl.instance().addInitInputArgument("restitution", argB)
```

In total we have 4 scalar values: x,y,z and restitution:

```
>> cd data/Matlab/scripts
>> [sizeInitInput, initOutput, sizeStepInput, sizeStepOutput] = agx('load',' bouncingSphere.py', [0,0,0, 0.5], 0.01)
```

The time step in the AGX simulation is set to 0.01 seconds (100Hz).

Running **agx('load' …)** while there is a running agx simulation called from the same Matlab instance will end this simulation and load the new simulation file instead. In order to modify the simulation, the AGX script file that is loaded by agx('load') has to be modified.

For doing a simulation step, run from the Matlab prompt

```
stepOutput = agx('step', <stepInput>)
```

where **stepInput** is a row vector of size 1xsizeIn and represents the input parameters for the simulation step.

stepOutput is a row vector of size 1xsizeOut and represents the output parameters for the simulation step.

In order to enable visual output in a separate 3D window, run from the Matlab prompt

```
agx('visual', 1)
```

This will start the AGX Remote Debugger as a 3D graphics viewer. Using graphics might slow down the simulation. After having enabled it, it can be disabled with `agx('visual', 0)`.

50.5. Graphics Viewer (AGX Viewer)

The graphics viewer window opens if:

- Simulink: the 'Use graphics viewer (AGX Remote Debugger)' checkbox has been activated in the AGX Simulink block.

- Matlab: The command `agx('visual', 1)` has been given in the prompt. Note that you have to reload the scene if you change the visual output setting from Matlab after the scene has been loaded.

Note

For stability reasons we strongly recommend that you start the viewer *before* you start Matlab/Simulink. This way, the viewer will run in a separate process instead of the same process as Matlab. In some cases, Matlab will interfear with externally loaded dynamic libraries. Running the viewer in a separate process will therefore always be more stable.

50.5.1. Starting 3D Viewer from command-line

To start the 3D Viewer from the command-line, make sure you are running the command from a command prompt with AGX Dynamics environment correctly setup, see: Section 5.2.1.

```
agxViewer --coSimulationServer
```

This will create 3D Viewer that is waiting for signals from an AGX Dynamics block running in either Matlab or Simulink.



Fig. 50.4 Model view and AGX Remote Debugger view of example scene.

The view can be changed (rotated, zoomed, panned) using the mouse.

For a list of available key bindings of the agxViewer application, see Table 29: Keyboard bindings.

On some systems, you might get warnings from your firewall when using the remote debugger for visual output, since data is sent via a socket on localhost from the plugin to the general AGX application. No internet connection is needed, so refusing the application connection to the internet should not impede the functioning of the program. If your firewall settings do not allow the opening of a socket on localhost and you do not want to/cannot change that, you might have to fall back to using the plugin without the visual output via remote debugger (and thus without graphics).

50.6. Defining input and output signals

The coupling between Matlab/Simulink and AGX Dynamics is done by setting up callback functions. These functions will be called at initialization or every time step for reading input and output data. Below is an example written in Python.

This example scenario consists only of one rigid body, a box, which is rotating along a hinge which is fixed in the world coordinate system. It receives one input per time step: The speed of the hinge motor. The simulation has one output per time step: The summed-up angle of rotations around the hinge axis (this angle can thus be $> 2\pi$ or $< -2\pi$ in case of several full rotations).

The following two subsections will explain how this input and output values are defined in Python.

50.6.1. Python

Here we will use *<AGX-dir>/data/matlab/scripts/rotatingBox.py* as an example.

50.6.1.1. Defining the parameters

When loading the Python script, AGX will first parse it and execute all commandos which are not inside in a function, that is all code in *global scope*.

Note

Inputs and outputs (both for scene initialization and step-function) have to be defined outside of functions (global scope).

The scene itself is defined in the function **buildScene()** in the Python script. This function will load the simulation scenario.

In the simple case of the example file, the whole scene is setup directly in this Python function. In more complex cases, you might want to call one or several separate Python files to set up the scene, or load in data from other files.

The input signals (data from matlab to AGX) and output signals (data from AGX to matlab) are implemented in classes derived from the class agxMex.PythonControlArgument:

```
class HingeSpeedInputControl(agxMex.PythonControlArgument):
    def __init__(self, numInput, numOutput):
        # Call the constructor of PythonControlArgument
        # Specify the number of input and output variables
        super().__init__(numInput, numOutput)

    # This method is called at the startup of the simulation
    # It will get the intialization data from Matlab/simulink
    def input(self, sim, data):
        speed = data[0]
        constraint = sim.getHinge("Hinge")
        assert constraint
        motor = constraint.getMotor1D()
        motor.setSpeed(speed)
        return True

    # This method will be called every time step
    # Reading data from AGX and feeding it back to Matlab/Simulink
    def output(self, sim, data):
        assert(len(data) == 1)
        constraint = sim.getHinge("Hinge")
        assert constraint
        # read output data from simulation
        angle = constraint.getAngle()
        data[0] = angle
        return True
```

The init input control and the output control is then registered with the following lines:

```
# Specify number of input/output variables
numInput = 1
numOutput = 1
argA = HingeSpeedInputControl(numInput, numOutput)

# Register the Input port (to AGX) to the controller
agxSDK.SimulationControl.instance().addInitInputArgument("HingeSpeedInput", argA)

# Register the Output (from AGX) port to the controller
agxSDK.SimulationControl.instance().addOutputArgument("HingeAngleOutput", argA)
```

50.6.1.2. Evaluating inputs and outputs

Each time step, AGX does the following three steps:

1. Call all input functions in the order of their declaration in the Python script, making use of the input from Simulink to control the AGX simulation.
2. Take one simulation step.
3. Call all output functions in the order of their declaration in the Python script, writing results from the AGX Simulation back into the Simulink output data.

50.6.1.3. Initialization input and output

If you want to make the scene setup depend on parameters, you can set these using the function:

```
agxSDK.SimulationControl.instance().addInitInputArgument(<object derived from agxMex.PythonControlArgument>)
```

where <InputArgument> is the same kind of argument as for:

```
agxSDK.SimulationControl.instance().addInputArgument(<Object derived from agxMex.PythonControlArgument>)
```

If you want to get one-time output from the scene initialization, this can be specified using the function:

```
agxSDK.SimulationControl.instance().addInitOutputArgument(<object derived from agxMex.PythonControlArgument>)
```

The order of execution at scene initialization is:

- Parse Python script, and register all arguments which must be added to **agxSDK.SimulationControl**.
- Execute all instances of agxMex.PythonControlArgument added with addInitInputArgument
- Execute the function buildScene().
- Execute all instances of agxMex.PythonControlArgument added with addInitOutputArgument

50.7. Examples

50.7.1. MatLab

Examples of matlab scripts which how to use the AGX-Matlab connection is located in <AGX-dir>\data\Matlab\

All corresponding Python scripts are located in <AGX-dir>\data\Matlabscripts

To run an example, start Matlab, change directory to <AGX-dir>\dataMatlab\ and enter the name of the matlab script.

50.7.2. Simulink

Examples of Simulink projects which demonstrates the AGX-Simulink connection is located in located in <AGX-dir>\dataMatlab\simulink\

50.8. Limitations

This version of the Matlab/Simulink integration has several limitations.

- Dynamically linked libraries like agxMex.dll, agxCore.dll, agxPhysics.dll and agxOSG.dll will not be unloaded once they have been loaded in Matlab, until Matlab is shut down.
- **You cannot use Matlab and Simulink in the same session. Please restart Matlab if you are working with the Matlab connection and need to use Simulink.**
- Only one AGX block allowed per model.
- After the Matlab command clear has been executed, AGX should not be used (this includes also calls to **clear mex** or **clear all**).

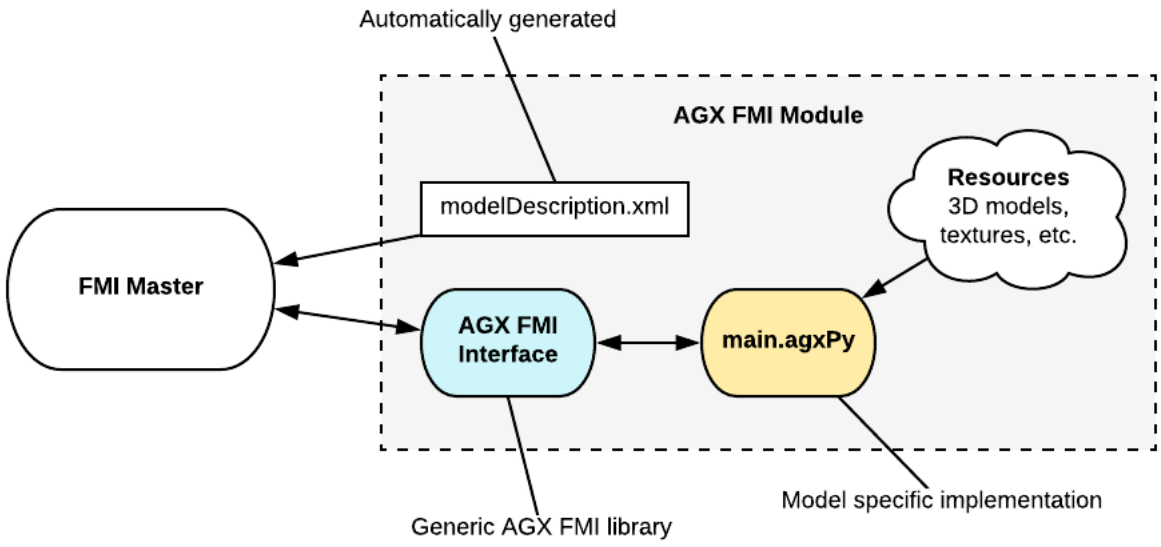
51. FMI Export

AGX Dynamics supports FMI export for co-simulation using the FMI-standard (see http://fmi-standard.org/). The standard (and its interface) is called Functional Mockup Interface (FMI), and one unit created or connected via this standard is called a Functional Mockup Unit (FMU).

The following tutorial demonstrates how to build an AGX Dynamics FMI module compatible with the FMI 2.0 standard. The resulting FMI module can, according to the FMI standard, also be referred to as an FMU (see above).

The FMI export interface is exposed in AGX using the *agxFMI2::Export* namespace. The interface bindings are defined using Python. AGX Dynamics only supports creating FMUs using the FMI 2.0 standard.

The following diagram displays the structure of a Python based AGX FMI module:



Building an AGX FMU consists of the following steps:

1. Setting up a directory structure matching the structure described in the FMI standard. The FMI module structure is described in section 2.3 (FMU Distribution) in the FMI2 specification.
2. Implementing the Python script, which is responsible for
    1. Defining the AGX scene to be simulated (in Python or C++)
    2. Registering the FMI variables to be exported
3. Running the AGX FmiExporter tool that will
    1. Automatically generate the FMI XML interface definition (`modelDescription.xml`)
    2. Package the FMU as a zip file according to the standard

51.1. AGX Python FMU generation

The FMU root directory, from here on referred to as `$FMU_ROOT`, must contain a Python file named `main.agxPy`. This is the main entry point for building the scene and setting up the module parameters.

The `main.agxPy` file must contain a function called fmiInit, which takes no arguments. The module of type `agxFMI2::Export::Module*` can be accessed through the global agxPython context:

```
def fmiInit():
    context = agxPython.getContext()
    export_module = context.environment.getFMI2Module()
```

This module contains a simulation instance, to which the simulated scene is added, an OSG render node for debug rendering, and methods for registering FMI parameters.

You can find an FMU example in the `data/FmiModules/Tutorial/Python/SpinningBox` directory.

The AGX-FMUs are exported using the FmiExporter tool, which has the following syntax

`FmiExporter --name <FMU module name> --output <FMU output path> <FMU data root>`

Name and output have implicit values so you can simply run

`FmiExporter myFmu`

where myFmu is the directory containing the FMU resources, including a mandantory `main.agxPy` file. In this case the module name will be myFmu and output path will be myFmu.fmu.

The FmiExporter will automatically generate a FMI modelDescription.xml and package the FMU zip file.

In order to run the FMU, AGX has to be installed on the computer with a valid license and valid environment variables.

You can test your generated FMU using the command:

```
> fmucheck -n 10 myFmu.fmu
```

51.2. AGX C++ FMU generation

The directory `data/FmiModules/Tutorial/CppModule/ExternalBuild` contains an example of how to structure and setup an AGX C++ FMU. It assumes an installed AGX package, and use a cmake build file to locate AGX, compile source code, and run the `FmiExporter` command line tool to produce an FMU zip package. To build the sample FMU:s as part of building the tutorials/examples, the CMake flag `AGX_BUILD_FMI_TUTORIALS` must be set to ON.

Also, if the CMake flag `AGX_ENABLE_TESTS` is set to ON, it is possible to test the built C++ FMU by executing the command:

```
> ctest -C Release
```

Note

It is important to build and run the FMU with the correct configuration. Because the naming convention of FMU:s it is not easy to make a distinction between release or debug builds. In Mac and Linux releases, it is only possible to build and run the FMU:s in release mode. We recommend building FMU:s in Release mode when possible due to conflicts between different runtime libraries.

The general structure for a C++ based FMU is similar to the python FMU:s. At least one source file named `main.cpp` must be present in the resource directory and all files in the named directory will be part of the FMU, including the source files. The main.cpp source file must at least contain the following functions:

```
#include <agxFMI2/export/Module.h>
#include <agxFMI2/export/VariableUtil.h>
#include <agxCollide/Box.h>
#include <agx/Hinge.h>

using namespace agx;

namespace
{
  // Called when the fmi is shutdown
  fmi2Status fmiShutdown(agxFMI2::Export::Module *)
  {
    return fmi2OK;
  }

  // Called when the fmi is initialized
  fmi2Status fmiInit(agxFMI2::Export::Module *module)
  {
    agxSDK::Simulation *simulation = module->getSimulation();

    // Here is where we build our model
    RigidBody *body = new RigidBody(new agxCollide::Geometry(new agxCollide::Box(1, 1, 1)));
    simulation->add(body);


    agx::Hinge *hinge = new agx::Hinge(body, new agx::Frame());
    hinge->setName("TheHinge");

    hinge->getMotor1D()->setSpeed( 1.456 );
```

```
    hinge->getMotor1D()->setEnable( true );
    simulation->add(hinge);

    // Create motor speed input parameter
    auto vSetSpeed = new agxFMI2::Export::InputVariable_Real("targetMotorSpeed");

    // Callback when parameter is changed from FMI master
    vSetSpeed->setCallback([=](Real val) {
      hinge->getMotor1D()->setSpeed(val);
    });

    vSetSpeed->setDescription("The motor speed controller");
    vSetSpeed->setStartValue(hinge->getMotor1D()->getSpeed());
    module->registerVariable(vSetSpeed);

    // Create box angular velocity output parameter
    auto vSpeedOutput = new agxFMI2::Export::OutputVariable_Real("boxAngularVelocity");

    // Callback when parameter is read from FMI master
    vSpeedOutput->setCallback([=](Real& val) {
      val = body->getAngularVelocity().z();
    });

    vSpeedOutput->setDescription("The current angular velocity");
    module->registerVariable(vSpeedOutput);

    return fmi2OK;
  }

  // Called when the application is initialized
  fmi2Status fmiInitApplication(agxFMI2::Export::Module *)
  {
    return fmi2OK;
  }
}
```

52. Debugging

52.1. Debug rendering

AGX has an internal scheme for rendering rigid bodies, geometries and constraints (including contacts). It is handled by a class **agxRender::RenderManager**. When debug-rendering is enabled, this class will dispatch calls for rendering and create **agxRender::RenderProxy's**. A render proxy is a placeholder for a class that can be rendered in the clients rendering system. By default if AGX is built with OpenSceneGraph, there is an implementation of render proxies for that scene graph.

Each primitive has its own render proxy which need to be implemented at the client side. Responsible for creating these proxies is the **agxRender::RenderProxyFactory**, a class which is derived to a specialization for each rendering system. The class **agxOSG::RenderProxyFactory** is one such specialization for OSG.

This rendering system should not be considered to be the rendering system for your simulations in simulators etc. It is merely a system for debugging your simulation, being able to see if bodies are enabled, static, kinematic, sleeping. To see where your constraints are attached to bodies etc. When you need a proper rendering engine, you should connect that directly through the geometries and rigid bodies.

Table 52.1 Classes for rendering

Class	Description
agxRender::RenderManager	Manages the debug rendering system. Called from agxSDK::Simulation::stepForward()
***agxRender::RenderProxyFactory**	Abstract base class responsible for creating agxRender::RenderProxy for rendering in a specific rendering system.
agxRender::RenderProxy	Abstract base class for all renderable objects. Derived down to various primitives (sphere, cylinder, etc.)
***SphereProxy, *CylinderProxy, *LineProxy, *TextProxy, *PlaneProxy, *CapsuleProxy, *BoxProxy, *HeightfieldProxy, *TrimeshProxy**	Specialization of a RenderProxy for each supported shape.
agxRender::GraphRenderer	An abstract class for rendering graphs related to statistics information.

Classes marked with * in Table 52.1: indicates that they need to be implemented and adopted for your specific rendering engine.

Objects with different properties are rendered differently: geometries belonging to static bodies are rendered in blue, whereas geometries associated to dynamic bodies are rendered in green. Sensors are rendered with a lighter blue.

52.1.1. RenderManager

agxRender::RenderManager. This class is responsible for updating all the RenderProxy instances that are created during debug rendering. Each render manager need a reference to a RenderProxyFactory, specific for a rendering engine. Several render managers can use the same RenderProxyFactory, so if you are using several agxSDK::Simulation, you can assign the same RenderProxyFactory to those simulations.

```
// Create a render proxy factory, specific for OpenSceneGraph
agxRender::RenderProxyFactoryRef factory = new agxOSG::RenderProxyFactory;

// Create a simulation
agxSDK::SimulationRef sim = new agxSDK::Simulation;

// Tell this simulation's render manager to use our factory
sim->getRendermanager()->setProxyFactory( factory );

// Create another simulation, use the same factory for the debug rendering
agxSDK::SimulationRef sim2 = new agxSDK::Simulation;
sim2->getRendermanager()->setProxyFactory( factory );
```

During a Simulation::stepForward(), a call to RenderManager::update() will be called which in turn will query the render proxy factory for spheres, lines, cylinders etc.

There is no cache functionality implemented in the RenderManager, that is, when a render proxy is required, a call to the render proxy factory will be executed. There will be no storage of unused render proxies in the render manager. If caching is required (to achieve better performance for scenes where the number of proxies varies a lot), it has to be done in the render proxy factory implementation

52.1.2. Global transformation

Because simulation can occur in any coordinate system, there might be limitations to what the rendering system can handle. For example, if a simulation occurs far away from the origin (0,0,0) of the world coordinate system, there might be precision problems in the rendering system. To handle this, the RenderManager has a global transformation that will be applied to all render proxies. This transformation can be set using the method:

`agxRender::RenderProxyFactory::setGlobalTransform`

This should be set to the inverse of the transformation of any object that should be rendered in (0,0,0) in the rendering system.

```
// Assume centerObject is an object in the scene that should be at (0,0,0) in the rendering system
agx::AffineMatrix4x4 matrixTransform = centerObject->getTransform().inverse();
simulation->getRenderManager()->getProxyFactory()->setGlobalTransform(matrixTransform);
```

52.1.3. Scale factor

By default the overall scaling of debug rendering is configured to fit a scene where objects are around 1 m. For a very small or very large scene it might be useful to increase/decrease the size of the rendered constraints, contacts etc. This can be done using the *setScaleFactor* method:

```
simulation->getRenderManager()->setScaleFactor(0.1f);
```

The scale factor can also be controlled with the keyboard in an application based on the `agxOSG::ExampleApplication` class such as agxViewer. *Note that the application/script might have changed the mapping for those keys.* For more information see keybindings

52.1.4. Render flags

There are three methods that control what should be rendered by the render manager:

```
void setFlags( unsigned int flags );
void enableFlags( unsigned int  flags );
void disableFlags( unsigned int  flags );
```

setFlags will override the current set of flags with the specified one. *enableFlags* will enable the specified flags and leave the rest untouched. *disableFlags* will disable the specific flags and leave the rest untouched.

As an example, to specify that the default set of flags should be used, plus the rendering of bounding volumes (which is not enabled by default) the following call can be done:

```
// Render default, plus the bounding volumes
simulation->getRenderManager()->setFlags(agxRender::RENDER_DEFAULT|
                                                                    agxRender::RENDER_BOUNDING_VOLUMES);
```

52.1.5. Renderable objects

There are various types of objects that will be rendered in the debug rendering:

52.1.5.1. Shapes

All shapes (agxCollide::Shape) for all enabled geometries will be rendered. Every time a new shape is added/removed from a geometry part of the simulation, it will be added/removed from the debug rendering system (if it is enabled). A proxy will be associated for each shape. When the shape is deleted/removed, the proxy will get a call to onChange(REMOVE) (see below). For each time step, the state (color, alpha) and shape (size/form) will be synchronized with the corresponding AGX shape. For most of the time, only transform changes will be updated for these shapes, which should be efficiently handled by the rendering engine. The rendering of shapes will occur of the flag RENDER_GEOMETRIES is enabled in the render manager.

52.1.5.2. Constraints

The constraints in AGX has a virtual *render* method. This method will be called whenever the RENDER_CONSTRAINTS flag is enabled in the render manager. These render methods, will query the render manager for various shapes such as cylinder, spheres or lines. These queries will be dispatched to the specific RenderProxyFactory.

52.1.5.3. Renderables

There is a special class, agx::Renderable, which can be used for some specific rendering. It has a virtual method *render* which will be called from the RenderManager if the RENDER_RENDERABLES flag is enabled in the render manager.

52.1.5.4. Bodies

For each enabled RigidBody a sphere proxy will be rendered. These sphere proxies will be acquired via the render manager to the specific render proxy factory. This will occur if the RENDER_BODIES flag is enabled.

52.1.5.5. Contacts

Contacts will be rendered analogous to bodies. A sphere and a line will be acquired from the render manager. This occurs if the flag RENDER_CONTACTS is enabled.

52.1.5.6. Statistics

Statistics involve both the textual information (as collected by the agx::Statistics class) and graph drawing, showing some of the statistics as visual graphs on the screen. This occurs if the RENDER_STATISTICS is enabled.

52.1.6. Disable debug rendering for a specific geometry

By adding a specific property to a geometry, debug rendering can be disabled for that specific geometry. This is done by adding the property “DebugRenderEnabled” with value false to the geometry:

```
geometry->getPropertyContainer()->addPropertyBool( "DebugRenderEnabled", false );
```

Or in python:

```
geometry.getPropertyContainer().addPropertyBool( "DebugRenderEnabled", False )
```

52.1.7. Implementation of custom debug rendering

52.1.7.1. agxRender::RenderProxyFactory

This class need to be specialized for any specific rendering engine. In <agxOSG/RenderProxy.h> an implementation for OpenSceneGraph can be found. The virtual methods that need to be implemented are:

```
/// Interface for creating and returning a SphereProxy
virtual SphereProxy *createSphere( float radius ) = 0;

/// Interface for creating and returning a BoxProxy
virtual BoxProxy *createBox( const agx::Vec3& halfExtents ) = 0;

/// Interface for creating and returning a LineProxy
virtual LineProxy *createLine( const agx::Vec3& p1, const agx::Vec3& p2 ) = 0;

/// Interface for creating and returning a CylinderProxy
virtual CylinderProxy *createCylinder( float radius, float height ) = 0;

/// Interface for creating and returning a ConeProxy
virtual ConeProxy *createCone( float radius, float height ) = 0;

/// Interface for creating and returning a CapsuleProxy
virtual CapsuleProxy *createCapsule( float radius, float height ) = 0;

/// Interface for creating and returning TextProxy
virtual TextProxy *createText( const agx::String& text, const agx::Vec3& pos ) = 0;

/// Interface for creating and returning PlaneProxy
virtual PlaneProxy *createPlane( const agx::Vec3& normal, agx::Real distance ) = 0;

/// Interface for creating and returning HeightfieldProxy
virtual HeightFieldProxy *createHeightfield( const agxCollide::HeightField *hf ) = 0;

/// Interface for creating and returning TrimeshProxy
virtual TrimeshProxy *createTrimesh( const agxCollide::Trimesh *mesh ) = 0;
```

Each of these methods are responsible for returning a RenderProxy of a specific type based on the in-data. The RenderProxy is then responsible for holding this renderable reference into the render system in use. As an example, the code for creating a sphere OpenSceneGraph looks like this:

```
agxRender::SphereProxy* RenderProxyFactory::createSphere( float radius )
{
  ShapeData<osg::Sphere> sphereData;
  sphereData.shape = new osg::Sphere();
  sphereData.shape->setRadius( radius );
```

```
  sphereData.geode = createGeode( sphereData.shape, &sphereData.shapeDrawable );
  sphereData.geode->setName("SphereGeode");

  // Create a new Sphere proxy including the rendering representation in OSG
  agxOSG::SphereProxy *proxy = new agxOSG::SphereProxy( radius, sphereData, this );

  addChild( proxy->getNode() ); // Add the node to the scenegraph (hold by the factory)

  return proxy; // Return the proxy
}
```

The specific data required will differ between rendering engines.

52.1.8. agxRender::RenderProxy

Each subclass (primitive type) of RenderProxy need to be implemented for the specific render engine. The virtual methods that can/should be implemented are:

52.1.8.1. onChange

This method need to be implemented (pure virtual method) in your representation of a RenderProxy. It will receive calls from each of the set methods described below. Whenever a proxy is required to change color, shape, alpha or transform, this method will be called. Based on the event-type described in the enum agxRender::RenderProxy::EventType, you should update your rendering representation with the new current value. For the OpenSceneGraph implementation a code snipped looks like the following:

```
void onChange( RenderProxy::EventType type) {
  switch( type ){
      case (RenderProxy::ENABLE):
      setEnableOSG(getEnable());
      break;
      case (RenderProxy::ALPHA):
      setAlphaOSG(getAlpha());
      break;
      case (RenderProxy::TRANSFORM):
      setTransformOSG(getTransform());
      break;
...
```

So based on the event type different calls are done to the rendering API to reflect the changes into the rendering implementation.

52.1.8.2. updateShape

This method need to be implemented (pure virtual method) need to be implemented in your representation of a RenderProxy. It is specific for each shape. When this method is called (probably from your specific implementation of onChange(SHAPE)) the render proxy implementation need to update the rendering representation to reflect changes, such as changing radius for a sphere, change the triangle mesh structure or changing the size of a box.

A code snippet from the OpenSceneGraph representation illustrate how it can be done:

```
void SphereProxy::updateShape( )
{
  // Get the agx representation of the shape associated to this proxy
  const agxCollide::Sphere *sphere = getCastShape();
  if (sphere) // If there IS a shape associated, then read the radius from that
    m_radius = sphere->getRadius();

  // If radius is the same, then just dont do anything
  if (agx::_equivalent((float)m_radius, m_data.shape->getRadius()))
    return;

  // Change the radius of the osg-sphere
  m_data.shape->setRadius(m_radius);
  m_data.geode->getDrawable(0)->dirtyDisplayList();
}
```

52.1.8.3. setTransform

```
virtual void RenderProxy::setTransform( const agx::AffineMatrix4x4& transform );
```

This method will set the m_transform member of the RenderProxy, then it will call onChange(TRANSFORM) which indicates that the transform is changed. If m_transform == transform, no call to onChange will occur (to avoid unnecessary state changes. If you override this method, store transformation in m_transform and call onChange(TRANSFORM).

52.1.8.4. setColor

Store a color in m_color and call onChange(COLOR). If color == m_color, no call to onChange will occur to reduce state changes. If you override this method, update m_color and call onChange(COLOR).

52.1.8.5. getColor

This method return the color of the proxy, the value in m_color. You can override this method to return the color as stored somewhere else, just make sure you sync with the m_color member attribute.

52.1.8.6. setAlpha

Set the transparency value of the proxy. The implementation will store alpha in m_alpha and call onChange(ALPHA). If you override this implementation, you need to update m_alpha, and call onChange(ALPHA).

52.1.8.7. getAlpha

Return the value of the m_alpha member attribute. If you override setAlpha you might want to also override this method.

52.1.8.8. agxRender::GraphRenderer

This class is specific for each rendering engine. An implementation for OpenSceneGraph can be found in <agxOSG/Graphrenderer.h> and corresponding .cpp file.

For more information, see the doxygen generated documentation for the class.

52.2. Web Debugger

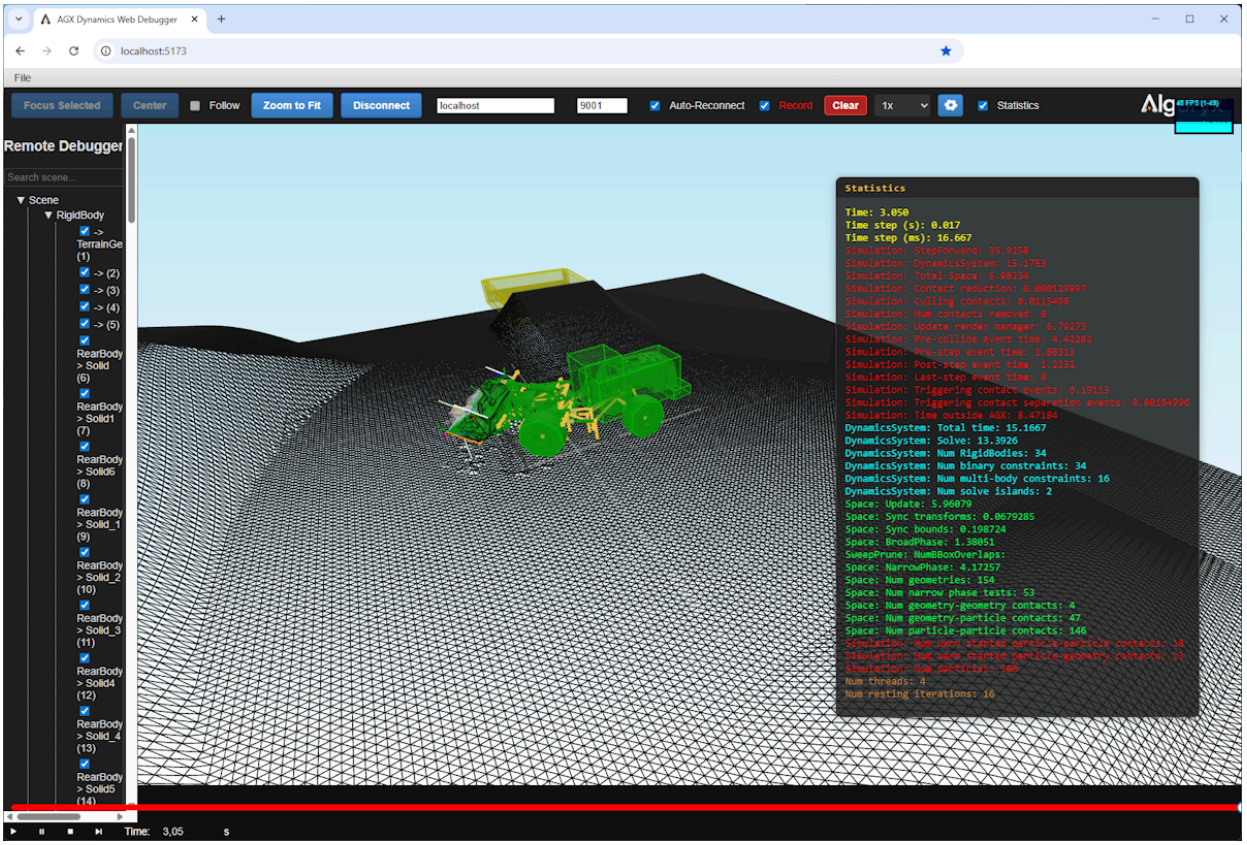


Fig. 52.1 Overview of the Web Debugger interface.

The Web Debugger is a tool that allows you to visualize and debug AGX Dynamics simulations in a web browser with minimal impact on the simulation performance. It provides a user-friendly interface to inspect the simulation state, view geometries, contacts and constraints. You can record and save a session for later analysis. The information presented is the same as what is available in the debug rendering system, but presented in a web interface.

It can be useful if you have multiple simulations running on different machines and you want to investigate the simulation state without having to login to each machine.

Note

- Currently only the visual representation of the simulation is supported. No dynamics information such as forces, velocities, or other simulation parameters are available in the Web Debugger.
- You cannot use the debug view in agxViewer/ExampleApplication at the same time as the Web Debugger.

52.2.1. Start a web debugger session

The Web debugger consists of two parts: A web server that serves the Web Debugger interface (client) and a simulation that streams the simulation state to the web client.

To start the Web Debugger client, you first need to start a web server that serves the Web Debugger interface. This can be done through command scripts both in Windows and Linux.

In Windows you can locate the scripts in bin/x64/WebDebugger and run the following command:

```
> run_server.bat
```

In Linux, you can locate the scripts in bin/x64/WebDebugger and run the following command:

```
$ ./run_server.sh
```

This will start a web server that serves the Web Debugger interface on port 5173 by default. Next you can connect to the Web Debugger interface by navigating to *http://localhost:5173* in your web browser and you will see the Web Debugger interface.

Note

You can change the port in the *run_server.bat* or *run_server.sh* script to use a different port than 5173.

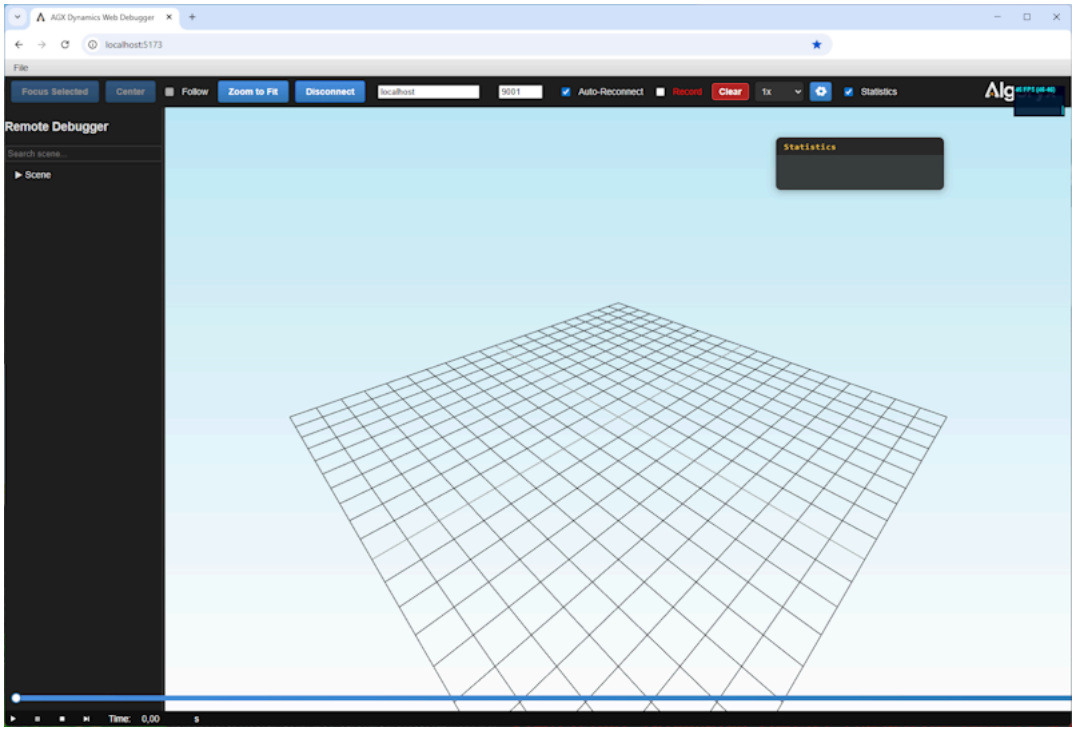


Fig. 52.2 Starting view of the Web Debugger interface.

To create a simulation session to which you can connect, you can use the agxViewer application or any other application that uses AGX Dynamics.

Through the agxViewer application, you can start a web debugger session by using the command line argument `--webDebugger 9001`. This will start a web server that serves the Web Debugger interface on port 9001 on the local machine. Note that it might be a good idea to start the simulation in paused mode by using the `-p` argument, so that you can connect to the Web Debugger before the simulation starts. This is only important if you are keen on recording every single time step of your simulation.

```
> agxViewer --webDebugger 9001 data/python/overheadConveyor.agxPy -p
```

Note

If you run the simulation on a remote machine, you need to replace the localhost with the IP address of the remote machine in the URL to access the Web Debugger.

The Web Debugger can also be enabled via the `agxSDK::Simulation::setEnableWebDebugger` method:

```
agxSDK::SimulationRef simulation = new agxSDK::Simulation();
simulation->setEnableWebDebugger( true, 9001 );
```

You then access the Web Debugger by navigating to *http://localhost:5173* in your web browser. The Web Debugger will automatically connect to the simulation and start visualizing the simulation state.

If you want to use a different port, you can specify it as the second argument to the method. The default port is 9001. By using different ports for different simulations, you can connect to multiple simulations from the same Web Debugger interface. Just open a new tab, connect to the Web Debugger and specify the port/host of the simulation you want to connect to.

Note

Because the Web Debugger is based on the Debug rendering system, there might be a need to transform the scene to origin (0, 0, 0) to avoid precision issues. See Global transformation

```
agxSDK::SimulationRef simulation = new agxSDK::Simulation();
simulation->setEnableWebDebugger( true, 9001 );

// Set the global transform after enabling the web debugger
simulation->getRenderManager()->getProxyFactory()->setGlobalTransform(matrixTransform);
```

52.2.2. User interface

52.2.2.1. Playback control

At the bottom of the interface, you have playback controls to start, stop, pause the simulation view.

- **Play**: This button will start the simulation playback.
- **Pause**: This button will pause the simulation playback.
- **Stop**: This button will stop the simulation playback and reset the time to the beginning of the recorded session or the current simulation.
- **Jump to end**: This button will jump to the end of the recorded session or the current simulation.
- **Time text field**: This text field allows you to enter a specific time in the recorded session or the current simulation. You can use this to jump to a specific time in the playback.
- **Time slider**: This slider allows you to scrub through the recorded session or the current simulation. You can use it to quickly navigate to a specific time in the playback.

52.2.2.2. Keyboard bindings

- **Left arrow**: Step backward in the simulation.
- **Right arrow**: Step forward in the simulation.
- **Space**: Pause or resume the simulation.
- **e / End**: Jump to the end of the recorded session or the current simulation.
- **b / Home**: Jump to the beginning of the recorded session or the current simulation.
- **WASD + QE**: Move the camera in the 3D view.
- **h**: Toggle visibility of any currently selected object.

52.2.2.3. Mouse controls

- **Left click + drag**: Rotate the camera around the target point.
- **Right click + drag**: Pan the camera.
- **Scroll wheel**: Zoom in and out.
- **Middle click + drag**: Zoom in out.

- **Double click**: Select the intersected object and set the rotation point for the camera.

52.2.2.4. File menu

- **Load Session**: You can load a previously saved session by clicking the “Load Session” button. This will allow you to select a JSON file that contains the simulation state, geometries, contacts, and constraints.
- **Save Session (Binary)**: This will store the recorded session in a binary, compressed format. This is the recommended format for saving sessions.
- **Save Session**: This will store the recorded session in a JSON format. This format is human-readable but can be larger in size compared to the binary format.
- **Request AGX Archive**: This will send a request to the simulation server to create and send an AGX archive of the current simulation state that you can save on your local computer for later analysis.

52.2.2.5. Session control

- **Focus Selected**: You can focus the view on a selected geometry (which can be a geometry, contact, or constraint) by clicking the “Focus Selected” button. This will center the view on the selected item and zoom in to make it more visible.
- **Center**: This will center the scene around a selected object. This is done on the server side, so this option is only available when you are connected. Also, it will only transform data sent _after_ it was pressed.
- **Follow**: You can enable the “Follow” mode by clicking the “Follow” button. This will make the camera follow the selected geometry as it moves in the simulation.
- **Zoom to Fit**: You can zoom the view to fit all or the selected geometries in the simulation by clicking the “Zoom to Fit” button. This will adjust the camera position and zoom level to show all geometries in the simulation.
- **Connect**: You can connect to a simulation by clicking the “Connect” button.
- **Simulation Address**: You can specify the IP address of the simulation to connect to in the “Simulation IP Address” field. This is useful if you are running the simulation on a remote machine.
- **Simulation Port**: You can specify the port of the simulation to connect to in the “Simulation Port” field. This is useful if you are running the simulation on a remote machine or if you want to connect to a different simulation instance.-
- **Auto-Reconnect**: If checked, the Web Debugger will automatically try to reconnect to a simulation. This is only possible if the current simulation is paused/stopped.

    Note that you should uncheck this button if you do not want the current session to be overwritten when connecting to a new simulation.
- **Record**: If checked, the Web Debugger will record the current session. This means that the simulation state, geometries, contacts, and constraints will be saved to a file when the session is stopped.
- **Clear**: With this button you can clear the recorded data and start from a new recording.
- **Speed dropdown**: You can select the speed of the simulation playback from the dropdown menu. The available speeds are 0.1x, 0.5x, 1x, 2x, and 4x. This allows you to slow down or speed up the playback of the simulation.
- **Statistics**: Uncheck this box to disable the statistics view in the Web Debugger.

52.2.2.6. Settings

- **Settings** for the Web Debugger can be accessed by clicking the gear icon in the top right corner of the interface. The settings allow you to configure the following:
    - **Enable logging**: Check this box to enable logging of the Web Debugger. This will log messages to the browser console, which can be useful for debugging issues with the Web Debugger.
    - **Show statistics**: Check this box to display the statistics window at startup of the Web Debugger.
    - **Show Grid**: Check this box to enable the grid in the Web Debugger view. The grid can help you to visualize the scale and position of the geometries in the simulation.
    - **Cell size**: You can adjust the size of the grid cells using the slider. This will affect how many grid lines are shown in the view.
    - **Total size**: You can adjust the total size of the grid using the slider. This will affect how far the grid extends in the view.
    - **Solid background**: You can switch between a light and dark background color for the Web Debugger interface.
    - **Camera speed**: You can adjust the camera speed using the slider. This will affect how fast the camera moves when you navigate the view with WASD keys.
    - **Up Axis**: You can select the up axis for the Web Debugger view. The available options are Left handed Z-up (Z+), Right handed Y-up (+Y) and right handed X-Up (X+).

- **Primitive tesselation**: Specifies the tesselation ratio of primitives. Default value is 10.

The settings can be reset using the “Reset settings” button at the bottom of the settings panel.

52.2.2.7. Scene browser

There will be a representation of the current simulation in the scene browser. This will show the geometries, contacts, and constraints in the simulation. You can disable the rendering of individual objects. In the Search scene field, you can filter the objects in the scene browser by name.

Note

If you experience problems it is always a good idea to open the browser console (F12) and check for any error messages.

52.3. Remote debugging

We strongly recommend using the Web Debugger for remote viewing of a running simulation. However, if you want to use the traditional remote debugging feature, it is described below.

Running AGX Dynamics embedded into a simulation framework, it can be sometimes hard to understand what is going on during a simulation. Therefore there is a feature which allows for some remote viewing capabilities.

In short, it is utilizing the serialization of a `agxSDK::Simulation` over a network port. This means that the whole simulation will be serialized, transmitted over the network and de-serialized and visualized in the agxViewer application.

Note

The Remote debugging feature should not be part of any production code, but rather as a way of looking into a running simulation. Enabling remote debugging will greatly affect the performance.

First, let´s have a look at how this works by using the command arguments for agxViewer.

52.3.1. Debugging agxViewer

Start a command prompt with the correct AGX runtime environment setup.

Next, start another command window (you will use one for the remote viewer and one for running the simulation).

From the first command prompt:

```
> agxViewer --server --serverTimeStep 0.1 --serverWait 0 data/python/overheadConveyor.agxPy
[Warning: agxSDK::Simulation::setEnableRemoteDebugging] Starting debugging server at port: 9999
Loading scene took 0.705391 sec
```

and from the second command prompt write:

```
> agxViewer --connect localhost:9999 -p --renderDebug 1
Connecting to remote server at localhost port: 9999
Trying to connect to server...
Connected.
```

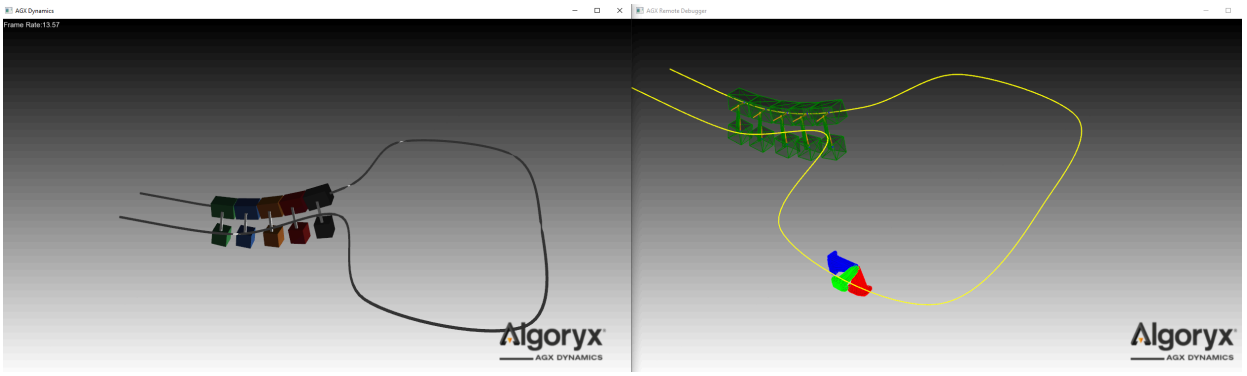


Fig. 52.3 The image to the left show the running simulation and the one to the right show the remote debugger view.

Lets look at the arguments of the two commands. On the server side we have:

- `--server` - Indicates that a port should be setup where the server will wait for a connection. The port can be changed using the `--serverPort <portnum>` argument. Default is 9999

- `--serverTimeStep 0.1` - specifies the rate by which the simulation will be streamed over the network connection in ms. This should be a multiple of the current simulation time step.
- `--serverWait 0` - If this is set to 0 (disabled) the server will not wait for a ready signal from the client and instead just push data onto the network. Hence synchronization between client and server will not be upheld.
- and finally the script that should be executed.

On the client side we have the following arguments:

- `--connect localhost:9999` - This specifies the network ip adress to where the server is running. In this case localhost (127.0.0.1) and a specified port (9999). If port is omitted, the default port 9999 will be used.
- `-p` or `--startPaused` - This is important as it will start agxViewer in *paused* mode. This will show the data being streamed into the client. If this was omitted, or if you press 'e' key to start the simulation, you would no longer stream data into the client, instead you are running the actual simulation locally. This is enabled by the fact that we are doing full de-serialization of the simulation.
- `--renderDebug 1` - This will enable the debug renderer so that we can actually see the parts of the simulation.

52.3.2. Enabling remote debugging in an application

The remote debugging feature can be enabled for any `agxSDK::Simulation` instance.

```
agxSDK::SimulationRef simulation = new agxSDK::Simulation();

// Specify the rate by which we will send data over to the server. Should be a multiple of the simulation time step
double serverTimeStep = simulation->getTimeStep() / 2.0;

// Specify the TCP port on which the server will listen to the connection
agx::Int16 serverPort = 12345;

// Should we compress (zip) the data before sending?
bool compressData = false;

// Lets not wait for client to be ready for new data. Just send new packages as fast as we can
bool waitForClient = false;

simulation->setRemoteDebuggingTimeStep(serverTimeStep);

// Enable the remote debugging
simulation->setEnableRemoteDebugging(true, serverPort, compressData, waitForClient);
```

52.3.3. Performance of remote debugging

Due to the full serialization of the simulation state each time it is being synchronized, the performance can be very much affected. Large objects such as heightfield, triangle meshes and terrains can lead to very slow remote debugging.

In those cases where you cannot achieve acceptable performance you can *disable serialization* for some of the larger objects. For example if you have a heightfield as a seabed, but you are mostly interested in looking at some mechanical simulation above the heightfield, then you can call the `Serializable::setEnableSerialization(bool)` method.

```
// Disable serialization for a agxTerrain::Terrain

terrain->setEnableSerialization(false);

// This will lead to better performance in remote debugging.
// Note that you probably have to loop through your geometries and look for
// triangle meshes/heightfields that you want to disable for serialization.
heightfield_geometry->setEnableSerialization( false );
```

52.4. Logging system

AGX contains a mechanism for logging messages to file, callbacks and/or console which is implemented through two classes, **agx::Logger** and **agx::Notify**. Notify is a class derived from std::ostringstream so it can be treated as any std::ostream using the overloaded << operator. The logging functionality is accessed through a number of macros defined in <agx/Logger.h>.

There are four different levels of messages as seen in Table 47.3: Values for Notify level

Table 52.2 Available macros for logging messages

Macro name	Description
LOGGER_DEBUG()	Start new debug message.
LOGGER_INFO()	Start a new information message.
LOGGER_WARNING()	Start a new warning message.
LOGGER_ERROR()	Start a new error message.
LOGGER_END()	Commit the started message (if the message is an error and an exception should be thrown, the actual error is thrown after this call).
LOGGER_ENDL()	Same as LOGGER_END() but add a std::endl to the stream.
LOGGER_STATE()	Change the output state for this message. Determines what is written as a message, line number, current function etc.

Examples of using the LOGGER macros:

```
LOGGER_WARNING() << "The file: " << filename << " cannot be read" <<
LOGGER_ENDL();

LOGGER_INFO() << LOGGER_STATE(agx::Notify::PRINT_NONE) <<
  "This is a plain message with no extra information " << LOGGER_ENDL();

LOGGER_ERROR() << "A serious problem has occurred" << LOGGER_END();
```

The last example with LOGGER_ERROR() will throw an exception if `LOGGER().setExceptionOnError( true );` is called prior to the call.

52.4.1. Setting Logging level

The level of output/logging can be specified either via Environment variables or via the C++ API:

```
#include <#include <agx/Logger.h>
// For logging to file/streams opened via NOTIFY().openLog(filename);
// This occurs automatically whenever a Logger::instance() is created and is controlled via the environment variable
AGX_LOG_ENABLE
// setLogNotifyLevel specifies what goes into the log file.
//
// The example below will indicate that we only want warning and higher (error) messages written to the log file.
// debug and info will not be displayed in the logfile.
NOTIFY().setLogNotifyLevel( agx::Notify::NOTIFY_WARNING );
```

If you want to control the output to the console (might or might not be written to the log file depending on the setLogNotifyLevel).

```
#include <#include <agx/Logger.h>
// For logging to console
// Show only warning and higher (error) messages.
// debug and info will not be displayed in the console
NOTIFY().setNotifyLevel( agx::Notify::NOTIFY_WARNING );
```

52.4.2. Redirecting Logger messages

Messages that are generated during runtime can contain important information. For example when reading mesh data the constructor of `agxCollide::Trimesh` can generate warnings about the validity of the mesh. If there is no console available, these messages will not be shown to the user. Either one have to enable the logging to file functionality through Table 47.1 or through the API:

```
bool overwrite = true; // overwrite if there is already a log file with this name
bool closeAndOpen = true; // If a log file is already open, close it and open the new, otherwise keep the previous one
opened
LOGGER().openLogFile( "c://temp/logfile.txt", overWrite, closeAndOpen );
```

There are also a number of ways to capture the messages written to the LOGGER system.

52.4.2.1. Deriving from a callback

In many situations it is possible to derive from the `agx::NotifyCallback` class and recieve messages.

Note

Callbacks are triggered from within the std::ostream system in C++. This can cause some issues in certain cases. For example in multi-threaded environments or when used in a C# context such as Unity3D. In those cases we recommend to use the LoggerSubscriber.

To recieve messages through callbacks, one derives a class from `agx::NotifyCallback` and implements the `message` method.

```
class MyNotifyCallback : public agx::NotifyCallback
{
public:
  MyNotifyCallback() {}
  ~NotifyCallback() {}

  /// Virtual method called whenever there is a message ready
  void message( const agx::String& msg, int notifyLevel) override
  {
    // Handle the message
  }
};

agx::ref_ptr<MyNotifyCallback> callback = new MyNotifyCallback();
// Register the callback
LOGGER().addCallback(callback);

// Unregister the callback
LOGGER().removeCallback(callback);
```

52.4.2.2. Subscribing to messages

When you want to have full control over when messages are handled, the `agx::LoggerSubscriber` is a better option. It works like a mailbox. You setup the system and poll messages using the `poll()` method.

```
// Specify which levels you want to listen to
int levelMask = agx::Notify::NOTIFY_ERROR | agx::Notify::NOTIFY_WARNING | agx::Notify::NOTIFY_INFO;

agx::ref_ptr<agx::LoggerSubscriber> subscriber = new LoggerSubscriber(levelMask);

/*
Run simulation....
*/

// Now we want to check for messages:
if (!subscriber->empty()) {
  std::string messages = subscriber->pop();
  std::cerr << "The message: " << message << std::endl;
}

// There is a "mail queue" for each level, so we can check them individually.
// We can store the messages in an array also:
agx::StringVector messages;

// Are there any warnings?
size_t numMessages = subscriber->pop(messages, agx::Notify::NOTIFY_WARNING);
```

53. References

[1] E. Hairer, C. Lubich, and G. Wanner. Geometric Numerical Integration, volume 31 of Springer Series in Computational Mathematics. Springer-Verlag, Berlin, 2001.

[2] C. Lacoursiére. Ghosts and Machines: Regularized Variational Methods for Interactive Simulations of Multibodies with Dry Frictional Contacts. PhD thesis, Dept. of Computing Science, Umea University, June 2007.

[3] K. L. Johnson. Contact Mechanics. 1985. doi: 10.1115/1.3261297.

[4] A. Pérez-González et al. “A modified elastic foundation contact model for application in 3D models of the prosthetic knee”. In: Medical Engineering & Physics 30.3 (2008), pp. 387–398. issn: 13504533. doi: 10.1016/j. medengphy.2007.04.001.

[5] F. Vasca et al. “Modeling torque transmissibility for automotive dry clutch engagement”. In proceedings: 2008 American Control Conference (2008), pp. 306-311.

[6] L. Eriksson et al. “Modeling and control of engines and drivelines”. Wiley, 2014.

[7] P. A. Simionescu et al. “Initial estimates in the design of central-lever steering linkages”. In: Journal of Mechanical Design 124.4 (2002), pp. 646-651. doi.org/10.1115/1.1505853

[8] P. A. Simionescu et al. “Initial estimates in the design of rack-and-pinion steering linkages”. In: Journal of Mechanical Design 122.2 (2000), pp. 194-200. doi.org/10.1115/1.533560

[9] A. Rahmani Hanzaki et al. “Kinematic and sensitivity analysis and optimization of planar rack-and-pinion steering linkages”. In: Mechanism and Machine Theory 44.1 (2009), pp. 42-56. 10.1016/j.mechmachtheory.2008.02.014

[10] V. C. Babu et al. “Design of accurate steering gear mechanism”. In: Mechanics and Mechanical Engineering 22.1 (2018), pp. 93-104.

[11] M. A. Kolekar et al. “Review on Steering Mechanism”. In: International Journal for Science and Advance Research in Technology 3.4 (2017), pp. 1155-1160. DOI: 10.13140/RG.2.2.17787.95525

[12] P. A. Simionescu et al. “Four-bar mechanism, Optimum synthesis, Ackermann linkage, Steering error”. In: Mechanism and Machine Theory 37.12 (2002), pp.1487-1504. 10.1016/S0094-114X(02)00071-X

[13] J. B. Heywood. “Internal combustion engine fundamentals”. New York :McGraw-Hill, 1988.

[14] E. Hendricks et al. “Modelling of the intake manifold filling dynamics”. In: SAE transactions, JSTOR (1996), 122-146.

[15] Y. Li and M. Sundén. “Modelling and measurement of transient torque converter characteristics”. MA thesis. 2016.