# Constraint Fluids on GPU

Martin Nilsson

September 7, 2009
Master's Thesis in Computing Science, 30 ECTS credits
Supervisor at CS-UmU: Lars Karlsson
Supervisor at Algoryx: Kenneth Bodin
Examiner: Per Lindström

UMEÅ UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE
SE-901 87 UMEÅ
SWEDEN

**Abstract**

The processing power of graphics hardware has increased tremendously in the last several years and they are therefore used more and more outside of their intended domain of graphics rendering. This thesis describes the implementation and results of a fluid simulator, using the constraint fluid method, which harnesses the processing power of modern GPUs, in particular NVIDIA's CUDA platform. As demonstrated in this thesis, particle systems with hundreds of thousands of particles can be simulated and visualized at interactive rates and systems containing up to a million particles can be run at a few frames per second. The biggest performance bottleneck is currently in the solver, in particular the lack of a working preconditioned Conjugate Gradient implementation.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# Introduction

Certain types of fluid simulations require the solution to large systems of linear equations and the close match between a GPU's parallel architecture and the parallel nature of some types of iterative solvers makes fluid simulations good candidates for significant speedups when the simulation is written to exploit the processing power of modern GPUs. This Master's Thesis explores the possibility of implementing a novel method for fluid simulations, constraint fluids, using NVIDIA's CUDA platform, which enables developers to run non-graphics related code on CUDA enabled hardware, such as any modern NVIDIA GPU. The constraint fluid method is suitable for visual and interactive applications, and gives improved stability compared to standard fluid simulation techniques, such as SPH [21], allowing for larger time steps and thus faster simulations [6]. The target platform for the software developed during this project, CUDA, is a hardware architecture developed by NVIDIA. It gives developers direct access to the computational units of CUDA capable devices, allowing for general-purpose computations on graphics hardware. This is desirable since the raw floating point performance of graphics hardware greatly exceeds that of a conventional processor.

The project is performed at Algoryx Simulations AB, a company specializing in physics simulators for the professional market. Their physics toolkit is used for example in vehicle, off-shore and medical educational simulators.

The report is divided into four parts. The first describes the purpose and goals of the project and gives a thorough problem description. The second part is a description of the platform that is the target for the libraries and applications that are the final result of the project, i.e. NVIDIA's CUDA. The third part is a collection of chapters, from Chapter 4 to 6, that gives the theory behind each of the topics that is required to implement the fluid simulator. The first chapter in this collection describes the theory behind the constraint fluid method and the steps that must be performed in order to get a working simulator. The other two chapters dive into the details behind pieces of the constraint fluid method and describe several methods to solve the subproblems. The final part of the report details the results of the project, the implementation and its performance and finally the project is evaluated.

A number of symbols are used throughout the report. Matrices will be denoted by capital Latin letters, such as $G$, and diagonal matrices by upper case Greek letters, e.g. $\Sigma$. Vectors are written as bold lower-case letters, for example $\mathbf{g}$. The number of something, such as the number of particles in a simulation, is denoted by $n$. Elements inside a vector is referenced by subscripting the vector with an index. For example, $v_i$ is the $i$th element of the vector $\mathbf{v}$. When talking about a sequence of something, the sequence number is marked as $^{(k)}$. For example, $\lambda^{(k)}$ is the $k$th $\lambda$ created by an iterative solver.

# Chapter 2

# Problem Description

This chapter describes the extent of the project and defines a set of goals that the project should reach. The chapter also includes a short survey of related work in the fields of fluid simulations and linear systems of equations solving.

## 2.1 Problem statement

The project should produce a fluid simulator, using the constraint fluid method, that uses a CUDA capable device to offload the computationally intensive simulation from the CPU. Within this problem lies collision detection between a large number of spheres and solving a sparse linear system of equations.

A demonstrator for the fluid should also be implemented. The demonstrator should visualize the fluid as it is being simulated inside a container and also be able to produce a dam break scenario, i.e., a quick expansion of the container in one direction.

## 2.2 Goals

The primary goal of the project is to have a working demonstrator for the fluid that can simulate hundreds of thousands of particles at interactive rates and even larger systems at non-interactive rates.

## 2.3 Purposes

Several applications can benefit from fast, high resolution fluid simulations and include, for example, engineering [50], visual effects for motion pictures [46], and interactive 3D games and immersive educational software [20]. All interactive applications have strong requirements on the performance of the simulation since a low frame rate severely degrades the user experience.

## 2.4 Methods

The development process involves several stages. A solid understanding of the CUDA platform must be acquired in order to write efficient applications, and algorithms for performing all the necessary steps of the simulation must be acquired or developed and finally implemented and

tested. Each part of the simulation must also be verified to ensure that the application produces correct simulations.

Knowledge about CUDA is obtained from documentation, presentation slides, educational videos, and code examples provided by NVIDIA and others. Skills in CUDA are developed by writing small applications that test concepts and techniques learned from the information sources. The actual simulation is implemented in an incremental fashion where a CPU-based, naive, spring-and-damper particle simulation is developed and then optimized using well established methods and algorithms for efficient collision detection. The CPU application is written with the CUDA platform in mind, using only algorithms and programming techniques that can be transferred to CUDA. The application is then ported to CUDA, which is the first major development effort using the target platform.

When a solid foundation for a particle simulation on CUDA is complete, the next phase is to replace the spring-and-damper mechanics with the real simulation of the Constraint Fluid method. The switch substantially increases the amount of mathematics in the application which makes it harder to debug since there are more layers of data processing between a detected collision in a set of particles and the resulting effect on the system. A number of Matlab programs are therefore constructed to validate the result from individual parts of the simulation.

## 2.5 Related work

There has been much work done in the field of fluid simulations, including efforts to utilize the processing capabilities of graphics hardware for the simulations. Harada et al. have done several projects [24] where fluids are simulated on a GPU [26, 25]. Hong et al. describe an extension to the FLIP [8] method where an incompressible fluid is simulated using particles with adaptable volume and mass [29]. Crane et al. describe a grid-based, as opposed to particle-based, fluid simulation method that they implemented using DirectX shaders [12].

The topic of equation solving using GPUs is explored by Bolz et al., who developed a conjugate gradient solver for sparse, unstructured matrices using shaders [7]. A more recent paper [9] was published by Luc Buatois et al. in which they describe their GPU implementation of the Jacobi-preconditioned Conjugate Gradient algorithm for sparse linear systems. The paper discusses both NVIDIA's CUDA API as well as the competing AMD/ATI CTM API. A more general discussion is made by Wiggers et al. who explore optimization techniques and multi-core solutions in order to speed up a Conjugate Gradient solver on both a dual-core processor and the NVIDIA G80 GPU [51].

# Chapter 3

# General-Purpose Computation on Graphics Processing Units

## 3.1 Introduction

The fast development of graphics hardware, or GPU for Graphics Processing Unit, in the last several years has resulted in an increase of floating point computation power that exceeds that of the CPU [14]. The main reason behind the steady progress is a pronounced focus on parallel execution [38]. In graphics rendering, which has been the primary focus of graphics hardware since their introduction, tasks such as transforming vertices and calculating lighting are naturally parallel [47]. Because of this, adding more computational units, or cores, to the graphics hardware significantly increases the peak performance. In contrast to CPUs, graphics hardware dedicate a large portion of its transistors to computational units such as ALUs and FPUs [10], while the CPUs spend more of the transistors on caches and control hardware that implement for example branch prediction and out-of-order execution [44]. A graphical representation of this can be seen in Figure 3.1. This difference makes the CPU and GPU suited for different kinds of problems. In a simplified way, one can say that a CPU performs best when a few, small pieces of data are processed in a complex, but sequential, way. This lets the CPU utilize the many transistors used for caching, branch prediction and instruction level parallelism [44]. The GPU, on the other hand, need massively data parallel problems to work efficiently [28].

The programming model most commonly used when programming a GPU is based on the stream programming model [31]. In the stream programming model, input to and output from a computation comes in the form of streams. A stream is a collection of homogeneous data elements on which some operation, called a kernel, is to be performed, and the operation on one element is independent of the other elements in the stream. Because of this, the graphics hardware can assign one element from each input stream to individual cores and have them run in parallel. When more cores are added, additional elements from the stream can be processed concurrently.

Another important difference between a general-purpose processor and a typical GPU is the memory bandwidth. Because of simpler memory models and no requirements from legacy operating systems and memory models, the GPU can support more than 100 GB/s of memory bandwidth, while the bandwidth of general-purpose processors is around 20 GB/s [14]. Another important aspect of memories, latency, is increasingly becoming the weakest link in a GPU. For each year, the computational power is increased about 70% and memory bandwidth by 25%, but

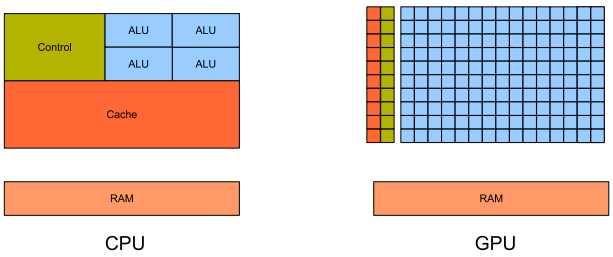


Figure 3.1: *Design differences between a general-purpose CPU and a GPU.*

memory latencies are improved by only 5% [44]. Because of this, communication with memory is becoming increasingly more expensive in terms of the number of arithmetic operations that can be performed while waiting for the data to arrive.

Because of the properties outlined above, scientists and engineers have begun to use GPUs for numerical calculations outside the field of graphics. Using GPUs for these types of applications is called General-Purpose computation on Graphics Processing Units, or GPGPU, and began with the introduction of programmable shaders in 2001 [35]. At first, programmers had to use graphics-centric APIs such as DirectX or OpenGL [23]. Therefore, concepts from the CPU realm had to be mapped to their GPU counterparts [27]. For example, arrays were stored as textures and inner loops represented as fragment shader programs. Later, in 2006, AMD announced Close To Metal (CTM), a hardware interface designed for stream computing [2] and NVIDIA announced CUDA, a general-purpose parallel computing architecture [43]. Both of these technologies provide programming models that let the programmer write programs without having to think in terms of textures and shaders.

## 3.2 CUDA

CUDA, Compute Unified Device Architecture, is a general-purpose hardware interface designed to let programmers use NVIDIA graphics hardware for purposes other than graphics in a more familiar way. In general, the hardware need not be a graphics related card at all, as there are cards designed specifically for general-purpose calculations [42]. From here on, instead of using the term GPU, any CUDA capable hardware will be referred to as a *device*. CUDA defines a programming model and a memory model that is consistent between all CUDA devices. The programming model describes how parallel code is written, launched and executed on a device and how threads are grouped into blocks. The memory model defines the different types of memories that are available to a CUDA program. The memory model is described further in Section 3.2.1 and the programming model in Section 3.2.2.

To the programmer, CUDA is an extension to the C programming language that allows the programmer to express parallel sections in the program for execution on the GPU. Such a section is called a kernel. The CPU, or host, sets up the CUDA environment and optionally copies input data to device memory and then launches the kernel. When the kernel has been launched, the host is free to continue performing computations on its own, in parallel with the device. When the host needs the result from the launched kernel, it initiates a memory copy operation which waits for the running kernel to finish and then transfers the result from the device to host memory.

Kernel code is written on the thread level with access to built-in variables that identify the

executing thread. As described in the introduction above, graphics hardware, which is the foundation of the CUDA architecture, has gained their high performance by dramatically increasing the number of cores in each device. The first generation CUDA devices, the GeForce 8800 series released in 2006, contain 128 cores [40] and the GeForce 285, released in 2008 has 240 cores [41]. To achieve high performance, it is therefore necessary to have many threads running each kernel. On current hardware several thousand threads may be required and this number is likely to increase with each new hardware generation. Threads are organized in a hierarchy and at the highest level of the hierarchy is the currently running kernel. The threads launched for a kernel are divided into blocks of threads, and the collection of all blocks for a given launch is called a grid. Threads within a block can synchronize and communicate via shared memory, but blocks can not be synchronized and must be independent of each other. The size and layout of the blocks are controlled by the program at kernel launch. The grid can be a one- or two-dimensional array of blocks and the blocks themselves contain threads organized in up to three dimensions.

The division of a kernel into thread blocks is what makes CUDA programs scalable. By forcing the blocks to be independent of each other, the CUDA runtime system can schedule the blocks in whatever way is most suited for the current device. Figure 3.2 shows two possible schedulings of a set of blocks, where the device with more cores finishes the computation faster. A CUDA application may get increased performance on new hardware with more cores if computational tasks are divided into enough blocks to utilize the available hardware resources.

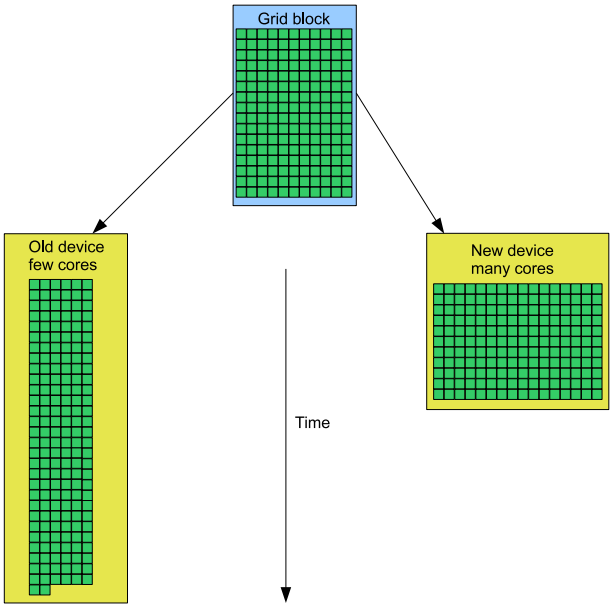


Figure 3.2: *Different devices result in different scheduling of thread blocks. More cores reduce the time required to compute the whole grid, if the grid contains enough blocks to fill available hardware resources.*

There is another property, other than highly parallel algorithms, that is important for achieving high performance on CUDA devices and that is the arithmetic intensity. Adding more cores to a chip adds computational power, but neither increases memory bandwidth nor reduces memory latency. To avoid idling cores, the ratio of arithmetic operations to memory operations should be as high as possible. CUDA partially mitigates the latency problem by using a form of

simultaneous multi-threading [18], i.e., by assigning several threads to each core and switching between them when a thread becomes idle. This is described in more detail below.

### 3.2.1 Hardware architecture

The right part of Figure 3.1 shows a simplified version of the internal layout of a CUDA device. In this section, a more detailed picture of the hardware will be given and the G80 family of graphics cards will be used as an example. The G80 was released in November 2006 and included in this family is for example the GeForce 8800 series of cards. Figure 3.3 shows a block diagram of the G80 family of graphics cards.

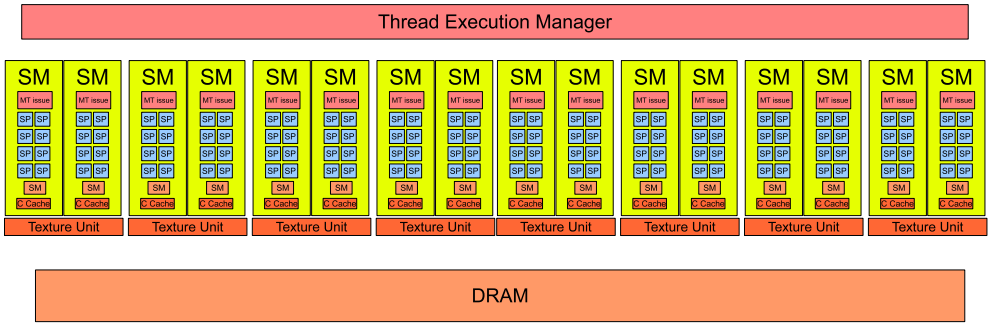


Figure 3.3: *Internals of an NVIDIA G80 GPU.*

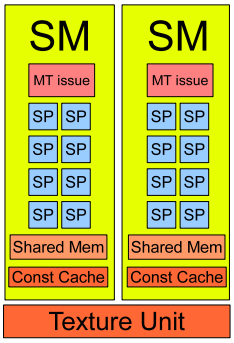


Figure 3.4: *Magnification of one multiprocessor block.*

The central part of Figure 3.3 illustrates the computational units, labeled SP for Scalar Processor, but they can also be called thread processors. The thread processors are split into groups of eight, creating several Streaming Multiprocessors (SM). Each block of threads is assigned to a SM and several blocks can be assigned to the same SM, allowing for fast thread switching to hide the memory latency.

#### Memory hierarchy

There are several types of memories in the CUDA memory hierarchy. The four types of physical memories are registers, shared memory, device memory, and host memory. Figure 3.4 shows the device memory and the Streaming Multiprocessor shared memory. Figure 3.5 shows all four

memory types. The registers are part of each scalar processor and the host memory is the regular system memory used by the CPU. It is not accessible by the CUDA threads.

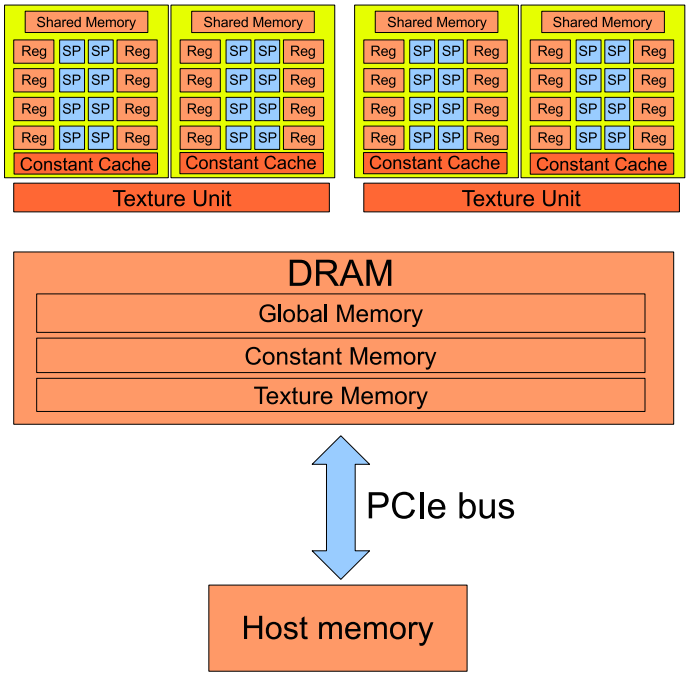


Figure 3.5: *The different memories that are used in a CUDA system. Closest to the Scalar Processors are the register files and the per-SM shared memory. At the other end of the hierarchy is the device RAM and on the other side of the PCIe bus the host memory. Two types of caches are located between the device memory and the SM's, the constant cache and the texture cache.*

The shared memory is very fast and designed with parallelization in mind. It is shared among all scalar processors in a streaming multiprocessor and can be used as a software-managed cache for data that is required by several threads in a block. Performance is increased greatly if threads in a block cooperatively read data from global memory to shared memory and then do all memory accesses on shared memory until a result is produced and finally written back to global memory by all threads in one chunk. Correctly implemented, this method makes use of coalesced accesses to global memory, which is described below. To increase the bandwidth of the shared memory, it is divided into banks. A bank is a hardware memory unit that service memory accesses and a memory can service as many simultaneous accesses as it has banks, which is 16 on the G80. In other words, 16 threads in an SM can read or write shared memory simultaneously if the reads are organized in such a way that each thread reads from or writes to distinct banks. Consecutive 32-bit words are assigned to consecutive banks. This has the effect that threads can read data block-wise: the first thread reads the first value, the second thread the second value and so on and achieve best possible throughput. The result of improper accessing patterns are bank conflicts. When two or more threads read different 32-bit words from the same bank the requests are serialized and throughput decreased as many times as there are requests to the bank. Several requests for data within the same 32-bit word does not result in any bank conflicts. Instead the word is broadcast to all requesting threads.

The global memory can be used in four different ways. The first is by dereferencing a pointer

given to the kernel from the CPU. This will access the device memory directly without using any caches, resulting in latencies of hundreds of cycles. The second way is to use a variable stored in the constant memory. As the name implies, the constant memory is read-only by CUDA threads and can therefore be cached to achieve efficient access. On the G80, the constant memory is limited to 64KB. An alternative to the constant memory, and the third way to access global memory, is texture lookups. In CUDA, textures work much like constant memory in that it is read-only and cached. The difference is that textures can be much larger, are optimized for 2D locality and provides hardware filtering. Any part of the global memory address space can be bound to a texture unit, allowing a kernel to read the output from a previous kernel through that texture unit. The fourth use of global memory is for thread local arrays. Arrays created for the CUDA threads are not stored in registers as are other local variables. Instead they are stored in global memory which results in long access delays and possibly memory bus congestion.

The introduction to this chapter mentions the increasing gap between memory speeds and computational power. This makes it important to use the device memory in the most efficient manner possible. In CUDA, this is known as a coalesced memory transaction. When accessing data in global memory the best performance is achieved when all threads currently executing on an SM access consecutive memory blocks of either 32, 64 or 128 bits that are aligned to the same size. When this is the case, all memory operations are grouped together to a single operation and the highest possible memory efficiency is achieved. The memory operations may be serialized if they do not meet this criterion, resulting in reduced memory bandwidth.

#### Streaming Multiprocessors

The streaming multiprocessors, SM for short, provide the computational power of a CUDA device. In the G80, each SM contains eight scalar processors, 8K 32-bit registers and 16KB of shared memory. When a CUDA kernel is launched the thread blocks of the grid are distributed over the SMs. The number of blocks that can be assigned to each SM is determined by the resource requirements for each block in the form of shared memory, register usage and the total number of threads. On the G80 the maximum number of threads per SM is 768, the maximum number of blocks per SM is eight and each block is limited to have at most 512 threads. Blocks may be queued if there are more blocks than the hardware can handle and these blocks will be launched when another block has finished. On the multiprocessor the thread blocks are split into warps and each thread is assigned to a Scalar Processor. A warp consists of 32 threads and is the smallest schedulable unit in a CUDA device. When issuing an instruction, the SM selects a warp that is ready to execute and issues the next instruction to that warp. Most instructions take four clock cycles to complete and the instruction pipeline for these instructions are four stages long. Therefore, a complete warp can be completed every four clock cycles. Each thread in the warp has its own register state and instruction address. If the threads within a warp diverge because of a data dependent conditional branch instruction, the hardware will execute all paths in serial and disable all threads that are not on the currently executing path.

There is no cost associated with switching between warps since all active warps are stored in the SM until that thread block has terminated. If one warp performs a long operation, such as a read from global memory, the scheduler can issue the next instruction to another warp immediately and in that way hide the memory latency as long as there are enough waiting warps to switch to.

### 3.2.2 Programming model

The programming model in CUDA is based on the thread and memory hierarchies described in Sections 3.2 and 3.2.1 and is exposed to the programmer as a small set of language extensions to the C programming language. The design philosophy behind a CUDA application is to find a serial sequence of segments in the application where each segment can be executed in a parallel fashion, and then split each parallel section into segments that can be solved cooperatively, independent of each other. The first level of segments is what is called a "kernel". A kernel is a function that is run *n* times by *n* different threads logically in parallel. The second level is the grouping of those *n* threads into thread blocks. The threads within the same thread block has access to the same shared memory and can synchronize among each other and thus cooperatively solve their piece of the problem. Each kernel can use the output from a preceding kernel as input and in this way several kernels can be chained together in order to solve more complex problems.

CUDA devices are built around an architecture NVIDIA calls Single Instruction, Multiple Thread (SIMT), a variant of the architectures in Flynn's taxonomy [4]. It is Single Instruction since each SM only issues one instruction at a time, and Multiple Thread since each thread has its own instruction address and can execute any code path. The design is similar to vector machines that are Single Instruction, Multiple Data (SIMD), the difference being that SIMD machines expose the vector width to the software whereas SIMT code specify the execution and branching behavior of a single thread. The programmer could ignore the SIMT architecture, writing the code for each thread independent of all other threads and still get correct behavior, but substantial performance improvements can be achieved by taking care to minimize path divergences within thread warps.

A CUDA program typically follows a pattern similar to the following. A set of input data is created on the host in some way, possibly generated or read from disk. This data is copied from host memory to device memory and then a kernel is launched. The CPU is free to continue execution in parallel with the kernel if there is work available, or simply wait for the kernel to finish. The CUDA threads uses their thread ID variable to find their piece of the data and optionally copies it to shared memory for others to use. The threads then perform some calculations and finally writes the result back to device memory. When the first kernel has terminated, the host can launch a second. There is an implicit synchronization if a kernel is launched while another is still running. The threads of the second kernel each read a piece of the data written by the first kernel, does some calculations and then writes the result back to device memory. This chain of kernels can continue for as long as necessary. When the final result has been computed the CPU can copy the result back to main memory and from there either continue performing operations on the data or store it somewhere.

### 3.2.3 Graphics API integration

CUDA can inter-operate with the two major graphics APIs OpenGL and DirectX. Resources such as vertex buffers and textures can be mapped into the CUDA address space and both read from and written to by CUDA kernels. The API supports mapping OpenGL vertex- and pixel buffer objects and from Direct3D can vertex- and index buffers, surfaces and textures be mapped, with some restrictions. For example can the primary render target and stencil- and depth buffers not be mapped.

### 3.2.4 Vector types

In addition to the primitive data types in C, CUDA supplies vector types of these containing up to four components. They are named by the primitive type of a vector element followed by the

number of components, for example `float4` and `int2`. These data types can also be used in standard C++ host code through `struct` definitions and overloaded operators defined in CUDA header files.

# Chapter 4

# Constraint Fluids

Constraint fluids [6] is a method for simulating fluid flows. It is a particle method, which means that the simulated fluid is represented as a set of particles each carrying a small quantity of the fluid. Each particle has a fixed size and carries a fixed mass. In each simulation step the method tries to move these particles according to the external forces in a way that keeps the density of the fluid as close as possible to a given target density.

## 4.1 Smoothed particle hydrodynamics

A number of properties, such as density, must be known in each point of the fluid when performing a simulation step. The constraint fluid uses a method called SPH, Smoothed Particle Hydrodynamics [21], when calculating these properties. The method was originally developed for astrophysical problems, but its ability to trace material interfaces, free surfaces and moving boundaries made it useful also in other areas such as material strength, metal forming, and fluid simulations [37].

SPH distributes the fluid within a particle according to a smoothing function, or *kernel*. The kernel is a function $W : \mathbb{R}^2 \rightarrow \mathbb{R}$ that has its highest value when the first given argument is zero and monotonically decreases as the first argument increases until finally reaching zero when the first argument is equal to or larger than the second argument, which is the radius $h$ of the particles. Figure 4.1 shows the distribution of fluid atoms inside a particle and an example kernel function.

To compute the density $\rho$ at any point $p$ in the fluid, the mass of each particle is multiplied with the kernel function given the distance, $d$, from the selected point to the particle center as argument. The results are summed and the sum is the fluid density at the selected point:

$$\rho(p) = \sum_i m_i W(d,h) \tag{4.1}$$

A number of kernel functions have been proposed. One that has been used successfully in fluid simulations is the following kernel, poly6, designed by Müller et al. [39]:

$$W_{poly6}(d,h) = \begin{cases} \frac{315}{64\pi h^9}(h^2 - d^2)^3 & \text{if } 0 \leq d \leq h, \\ 0 & \text{otherwise.} \end{cases} \tag{4.2}$$

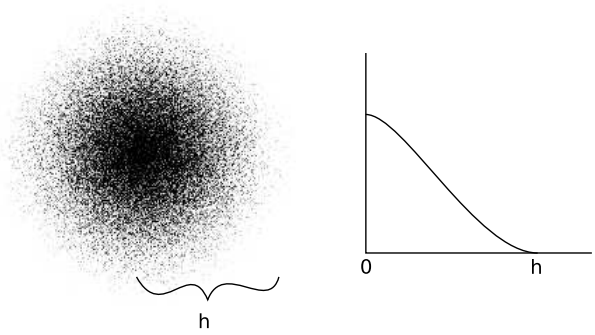


Figure 4.1: *Left: distribution of fluid atoms inside a particle. Right: kernel function that describes the distribution of atoms.*

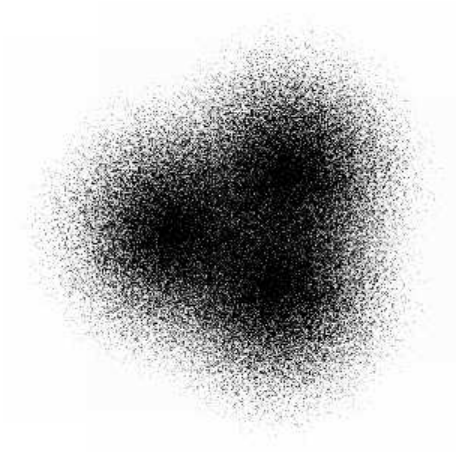

Figure 4.2: *Three overlapping particles. The density at any point in the overlap is calculated as the sum of each particle's contribution.*

## 4.2 Constrained dynamics

A constraint is a representation of what states are allowed or not in a simulation [52]. Examples of constraints in more general physics simulators are *distance constraints*, which say that two objects should remain at a fixed distance from each other, and *non-penetration constraints*, which say that an object may not penetrate for example a plane, such as a floor or a wall. Each constraint is represented by an indicator function that is zero when the system is in a legal state and further from zero the more the constraint is violated. The gradient of the indicator function describes in which direction a force must be applied to the violating object in order to restore the constraint, i.e. move the object into an allowed state. The part of the domain of the indicator function that the function maps to zero is called the *constraint surface* and the gradient is always normal to this surface.

In the beginning of each simulation step the state of the system is inspected and for each constraint one element in the constraint violation vector $\mathbf{g}$ is calculated, as well as the gradient vector of the indicator function. The gradient is inserted into the Jacobian matrix $G$, which is called a Jacobian matrix since it relates changes in the particles' coordinates to changes in the

constraint indicator function. From the matrix $G$, the constraint violations $\mathbf{g}$, and the current state of the system a linear system of equations is formulated and solved, yielding a set of Lagrange multipliers, denoted $\lambda$. The Jacobian matrix $G$, which holds directions, and $\lambda$, which represents magnitudes, are then used to calculate the forces that the constraints induce on the system. The simulation step is completed by applying the external forces and the constraint forces to all objects in the simulation to find their new velocities and positions.

### 4.2.1 Density constraints

The goal of the constraint fluid method is to keep the density of the fluid as close as possible to a target density $\rho_0$ and an indicator function that achieves this is

$$g_i = \rho_i - \rho_0 \tag{4.3}$$

where $\rho_i$ is the density calculated for particle $i$ using SPH as described above and $\rho_0$ is the target density of the simulated fluid. An alternative, but equivalent, formulation is

$$g_i = \frac{\rho_i}{\rho_0} - 1 \tag{4.4}$$

which gives a smaller number for fluids with large target density. This is the formulation used throughout the rest of this text.

### 4.2.2 Jacobian matrix

In addition to the constraint violation, each constraint has a vector that points in the direction in which a force must be applied to restore the constraint. For single particle constraints, such as the non-penetration constraint, the direction is usually simple to find. For the non-penetration constraint it is simply the normal of the plane. For the density constraint, which is a multi-body constraint, things are a bit more complicated. Each particle is influenced by all its neighbors and each such interaction must be recorded. The result is a matrix, denoted $G$, in which each row contains a set of direction vectors for one constraint and each block of columns is the contribution to the constraints from a particular particle. That is, for each density constraint $i$, row $i$ of the matrix $G$ has a non-zero vector in column block $j$ if the center point of particle $i$ lies within the influence radius of particle $j$. All interactions are symmetrical in the sense that all particles have the same radius and thus if particle $i$ influences particle $j$, then $j$ influences $i$ by the same amount, but in the opposite direction. The result is that the Jacobian matrix will, if disregarding the diagonal blocks, become skew-symmetric on the block level.

Figure 4.3 demonstrates the relationship between particle positions and the resulting Jacobian matrix. The matrix created from five particles is shown together with the particles' positions relative to each other. The final matrix is shown block-wise with color gradients to indicate influences between particles.

For $n$ particles in a three-dimensional space the matrix $G$ is an $n \times n$ block matrix where each block is a row vector of length three. Each individual block is calculated as

$$G_{ij} = \begin{cases} -\frac{mF_{ij}\hat{\mathbf{r}}_{ij}^T}{\rho_0} & \text{if } i \neq j, \\ \sum_k \frac{mF_{ik}\hat{\mathbf{r}}_{ik}^T}{\rho_0} & \text{if } i = j \end{cases} \tag{4.5}$$

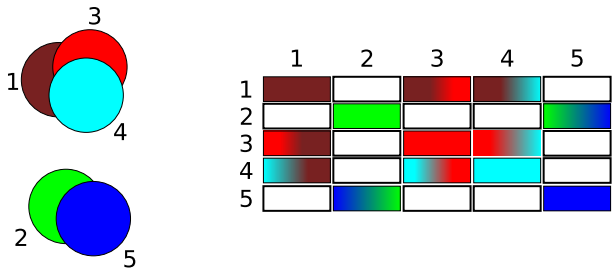


Figure 4.3: *Five particles and the Jacobian matrix that is created from their positions. The matrix is shown on the block level, where each block is colored according to the particles that influenced the value of that block. White blocks indicate independent particles and contain the zero vector.*

where $m$ is the mass carried by each particle, and $\hat{\mathbf{r}}_{ij}$ is the normalized vector pointing from particle $i$ to particle $j$. Also note that the diagonal blocks are the negated sum of all non-diagonal blocks on that row.

In the context of constrained dynamics, the restoring force for constraint violation is in the direction of the gradient of the indicator function. A vanishing gradient for approaching particles is undesirable since it corresponds to a zero restoring force. In the literature it is rather common to use different kernel functions for different purposes, even within the same simulation. For example, Müller et al. [39] applies three different kernel functions, some of which does not satisfy all formal requirements on valid kernel functions [36]. The reason is that the custom kernels results in improved stability. One such kernel, called the *spiky kernel*, brings stability by not having a vanishing derivative when the distance between two particles goes to zero. A vanishing derivative at small distances is hard to understand in a physics perspective, since it means that the restoration force becomes small at short distances. In turn, this results in a collapsing fluid once these short distances are reached. This motivates the changes to the kernel gradient.

Following Müller's example, a custom function for representing the kernel gradient is used here, chosen because of its better stability compared to the formal definition of the kernel gradient.

$$F_{ij} = \frac{945}{32\pi h^8}\left(h^2 - d^2\right)^2 \tag{4.6}$$

In the general case, the matrix $G$ is a $c \times dn$ matrix where $c$ is the number of constraints active in the simulation, $d$ is the number of dimensions in the space where the particles reside and $n$ is the number of particles. In a simple constraint fluid simulation there is one density constraint for each particle and thus $n$ rows in $G$. If the fluid is to be put in some sort of container with a non-penetration constraint for each particle, then $c$ could be as high as $2n$. However, this does not change the content and layout of the $n$ rows that control compressibility. In the final $G$ matrix, each constraint occupies one row and has a non-zero block for each particle it affects, and each particle occupies one block column and has non-zero blocks for each constraint it is affected by.

### 4.2.3 System of equations

To find the Lagrange multipliers $\lambda$, a system of equations must be solved. The system is created from the constraint violations $\mathbf{g}$, the Jacobian matrix $G$, and the current state of the system, including particle velocities $\mathbf{v}$ and external forces $\mathbf{f}$. In addition to this the equation includes

some simulation configuration parameters. The first parameter, $\Sigma$, is a regularization term that improves the condition number of the system matrix and adds a natural elasticity to the constraints. In the simulation this causes the fluid to be slightly compressible, which has relevance to the physics being simulated since even water is actually slightly compressible [1]. The second parameter, $\Upsilon$, controls how fast a violated constraint should be restored.

The equation to solve is the Schur complement form of the SPOOK [34] integration method:

$$\left(\frac{1}{m}GG^T + \Sigma\right)\lambda = (\Upsilon - I)G\mathbf{v} - \frac{4}{\Delta t}\Upsilon\mathbf{g} - \frac{1}{m}G\Delta t\mathbf{f} \tag{4.7}$$

The matrix on the left hand side of Eq. (4.7) is called the Schur complement matrix $S_\varepsilon$. The value of $\Sigma$ and $\Upsilon$ is controlled by two sets of scalar parameters $\tau$ and $\varepsilon$ with the definitions

$$\Upsilon = \mathtt{diag}\left(\frac{1}{1+4\frac{\tau_1}{\Delta t}}, \frac{1}{1+4\frac{\tau_2}{\Delta t}}, \ldots, \frac{1}{1+4\frac{\tau_m}{\Delta t}}\right) \tag{4.8}$$

$$\Sigma = \frac{4}{\Delta t^2}\mathtt{diag}\left(\frac{\varepsilon_1}{1+4\frac{\tau_1}{\Delta t}}, \frac{\varepsilon_2}{1+4\frac{\tau_2}{\Delta t}}, \ldots, \frac{\varepsilon_m}{1+4\frac{\tau_m}{\Delta t}}\right) \tag{4.9}$$

That is, both $\Upsilon$ and $\Sigma$ are diagonal matrices with one element for each constraint in the system and for each constraint there is a $\tau$ and $\varepsilon$ that define the $\Upsilon$ and $\Sigma$ matrices. The two sets of scalar values are the parameters that can be tweaked in order to control the behavior of the simulation. The parameter $\tau$ is the decay rate of constraint violations and can be set to a few multiples of the time step. The other parameter, $\varepsilon$, is the compliance of the constraints, which for the constraint fluids is the compressibility of the fluid.

Since the resulting $\Sigma$ and $\Upsilon$ matrices are diagonal, if the same $\tau$ and $\varepsilon$ is chosen for all constraints this can be seen as a single scalar that is either added or multiplied with either a matrix diagonal or a vector.

## 4.3 State update

Solving Eq. (4.7) yields the vector $\lambda$, which is used to update the state of the simulated system. The update method used is leap frog, which is done in two steps. The first step applies all forces, external and from the constraints, on the simulated particles and the second step uses the new velocities to update the position of each particle. The positions $\mathbf{p}$, velocities $\mathbf{v}$, and external forces $\mathbf{f}$ are vectors of length $dn$. Each block of $d$ consecutive elements in these vectors describes the state of one particle.

The update formula for particle velocities is

$$\mathbf{v}^{(k+1)} = \mathbf{v}^{(k)} + \frac{\Delta t}{m}\mathbf{f} + \frac{1}{m}G^T\lambda \tag{4.10}$$

Here, in the last term, the Lagrange multipliers $\lambda$ are used to calculate the velocity change each particle is subjected to by the constraints. The transpose of the Jacobian matrix, $G^T$, is multiplied with $\lambda$, which in essence means that each column, a record of which constraints affect one of the particles and in which direction, is per-element multiplied with the constraint force magnitudes which is stored in the Lagrange multipliers vector. The sum of these multiplications is the result from one segment of the $G^T\lambda$ multiplication and is the constraint force to apply to one of the particles in the system, scaled to compensate for the length of the time step. The force is divided by particle mass to get acceleration.

The second term is the other forces that act on the particle system. Gravity is an example of one such force. The force is divided by particle mass to get acceleration and then multiplied with

the length of the time step to get this time step's share of that acceleration. The two accelerations are added to the velocity that the particle had at the beginning of the time step, giving the new velocity of the particle.

When the new velocity has been found, the new position can be calculated using the following recurrence:

$$\mathbf{p}^{(k+1)} = \mathbf{p}^{(k)} + \Delta t \mathbf{v}^{(k+1)} \tag{4.11}$$

The velocity of the particle is simply multiplied with the length of the time step to get the movement during the current time step and the result is added to the old position.

# Chapter 5

# Collision Detection

In physics simulations it is important to find all collisions among the simulated group of objects in order to be able to simulate phenomenons such as bounces, stacks, and other types of contact-related situations. A collision can occur in two different forms; intersection and contact. An intersection is a state where a volume in space is occupied by two different objects. In other words, two objects overlap, which is usually not desired in a rigid body simulation. The other variant, a contact, is the case where a single point or an area is occupied by two objects. The real world counterpart to this is two objects touching each other, for example a box resting on a table.

Collision detection is not limited to finding all objects that collide, but may also find the location on each object where the collision occurred, the depth of the intersection and the collision normal.

In general, each object can collide with any other object in the simulation which makes collision detection an $O(n^2)$ problem. Also, since objects tend to move around a lot in simulations the collision detection must be performed each simulation step in order to find the current set of colliding objects. Because of these two properties, collision detection can become the bottle-neck in a simulation when the number of simulated objects increase. A number of techniques have been developed to speed up the collision detection compared to the naive approach of testing all pairs of objects.

## 5.1 Methods for collision detection

Several general methods for collision detection are reviewed. The methods are divided into three different types: bounding volumes, space partitioning, and sweep and prune. In the bounding volume methods, each object is enclosed in one or more bounding volumes to simplify the collision tests. The space partitioning methods divide the space where the objects reside into sections and the number of object pairs that need to be tested is reduced by only testing the object pairs where both objects overlap the same section. Sweep and prune is a single algorithm described last in this section.

### 5.1.1 Bounding volumes

Performing intersection tests between two complex geometries is an expensive operation. To increase the performance of simulations with many objects with complex shapes a pre-test can be performed before the more detailed intersection tests are done. By enclosing each object with

a simple bounding volume some pairs of non-colliding objects can quickly be rejected by testing for intersection between the bounding volumes. If the bounding volumes do not collide, neither can the geometries. A collision between two bounding volumes means that the geometries may collide and the exact collision test is performed.

A number of different shapes for the bounding volume can be used. The choice is a trade-off between how well the shape fits the geometry, how expensive the intersection test is, computational cost to find the bounding volume, and the storage requirement. Two very simple shapes are the sphere and the axis aligned bounding box (AABB). Both of these are easy to create and intersection tests are computationally cheap. A variant to the AABB is the oriented bounding box (OBB). This is a box that is rotated to achieve the best possible fit. These volumes are demonstrated in 2D in Figure 5.1. Even better, with regard to fitness, is the discrete orientation polytopes or k-dop [33]. A k-dop is a bounding volume enclosed by k planes, each with a given orientation. The AABB is a special case of a k-dop with k=6 and the normal of each enclosing plane parallel to the axes of the coordinate system. The k-dop creation process can be seen as placing k planes infinitely far away and moving them towards the object until all planes touch it. A 2D example of this is given in Figure 5.2. Common k-dops in a three-dimensional space are the 14-dop, which is an AABB with cut corners, 18-dop, a box with cut edges, and 26-dop where both the edges and corners are cut.

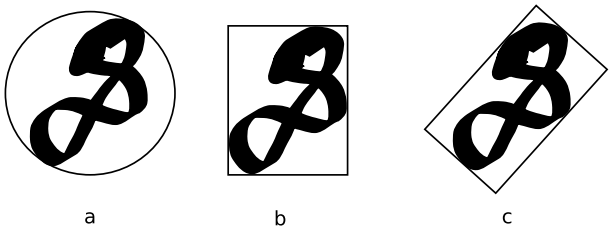


Figure 5.1: *Three bounding volumes demonstrated in two dimensions: a) sphere. b) axis aligned bounding box. c) oriented bounding box.*

To improve the fitness, a hierarchy of bounding volumes can be created [30]. A bounding volume hierarchy is a tree structure where the root node contains the bounding volume for the entire object, and each subsequent level contains finer and finer approximations of the object. Each internal node has a number of children that represent the finer bounding volumes, and the union of these volumes entirely covers the part of the object that was covered by their parent. This ensures that any point $p$ on the object is enclosed by all ancestors of the leaf node that contains $p$. This allows the collision detection algorithm to perform coarse tests on the large bounding volumes at the higher levels of the tree and prune all branches below a bounding volume that failed the intersection test. While the union of the children is required to cover all parts of the object that the parent covered, they are free to extend outside of the parent. However, this coverage is redundant since it will be covered by the parent's sibling and thus its children. Any collision test where the collision happens in this area will fail the collision test with the parent and instead descend into the sibling. On the other hand, if redundant coverage leads to a tighter fit of the area covered by the parent, it may still be worthwhile. The leaf nodes of the tree contains the primitives that build up the object and are used for exact collision tests when a collision has been found with a node in the last layer of internal nodes.

Figure 5.4 shows how a bounding volume tree of spheres can be organized. For each level, the bounding volumes for that level is drawn as a solid line and the bounding volume for the parent, if any, is shown in dotted lines. Since all volumes are spheres, this is an example of a

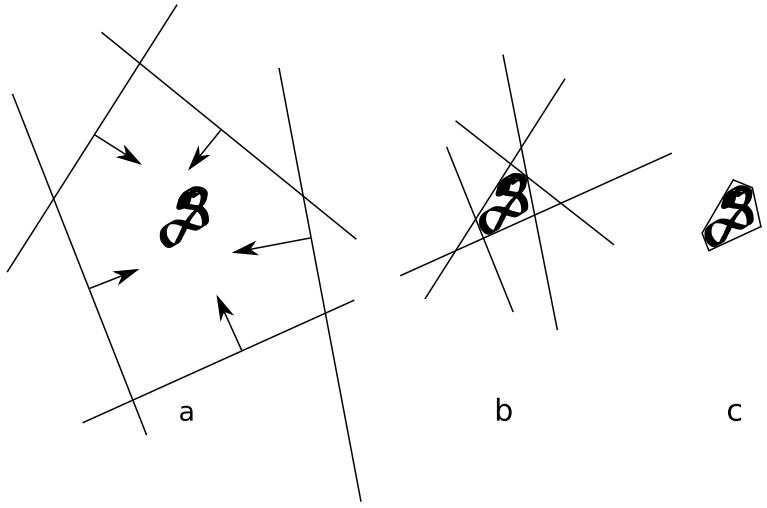


Figure 5.2: *Demonstration of a 2D 5-dop. a) Five lines are moved towards the object. b) Each line stops when it touches the object. c) The complete 5-dop.*

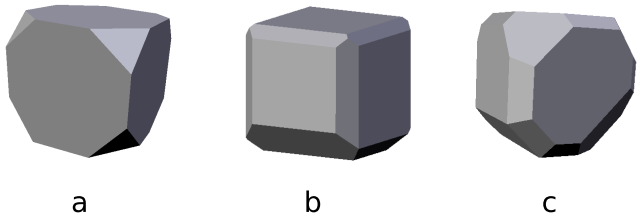


Figure 5.3: *Example k-dops in 3D. a) 14-dop. Box with cut corners. b) 18-dop. Box with cut edges. d) 26-dop. Box where both corners and edges are cut.*

*sphere-tree*. Other volumes are of course possible.

When a collision test between two bounding volume trees is to be performed the root node of each tree is tested against each other and if a collision is found the smaller volume is tested against the other node's children. This continues until two leaf nodes are found and collision tests between the real geometries are performed, or when a collision test between two bounding volumes report no intersection. The rest of that branch is then pruned and one of the siblings is tested next.

### 5.1.2 Space partitioning

Instead of testing each object against every other object, with or without the use of bounding volumes, one can use space partitioning. By splitting the world space into sections and mapping each object to the sections it overlaps, the collision detection algorithm only needs to perform collision tests between objects that share at least one section of the world space. Another way to look at it is to say that collision detection need only be performed for objects in sections to which more than one object has been mapped, and then only between objects that has been mapped to that section.

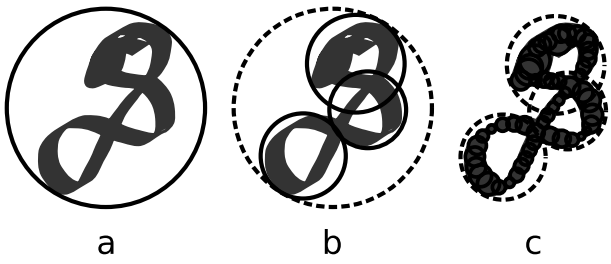


Figure 5.4: *Bounding volume hierarchy of spheres. a) Bounding volume stored in the root node. b) First level of children. Parent volume indicated by dotted lines. c) Final level of internal nodes. It is clear that the union of the closed circles is a better approximation of the object than the full bounding volume depicted in a).*

### Uniform grid

A very important property of a partitioning scheme is the shape and size of the sections that the world is split into. A simple way is to use AABBs in a uniform cell grid [22]. The following expression finds the index, **c**, of the cell where point $\mathbf{p} = [x, y, z]$ resides:

$$c_x = \left\lfloor \frac{p_x - x_0}{S_x} \right\rfloor, c_y = \left\lfloor \frac{p_y - y_0}{S_y} \right\rfloor, c_z = \left\lfloor \frac{p_z - z_0}{S_z} \right\rfloor, \tag{5.1}$$

where $[x_0, y_0, z_0]$ is the coordinate of the base vertex of the first AABB and $[S_x, S_y, S_z]$ is the size of each AABB.

The size of a cell is very important and should be based on the sizes of the simulated objects. The uniform grid works best when the objects are roughly the same size and evenly distributed over the world space. It can be hard to find a good cell size if there are both very large and very small objects. With small cells the large objects will be stored several times and thus increase the memory usage, but larger cells will cause some cells to hold a large number of small particles which will result in an increase in the number of collision tests that must be performed.

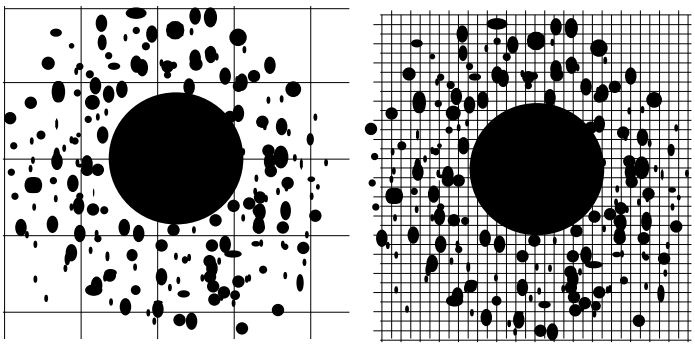

Figure 5.5: *Varying object sizes makes finding a good cell size hard.*

The storage of each cell is another important aspect. A three-dimensional array of cells is a simple approach, but may cost too much in terms of memory consumption if there are many empty cells. A technique to reduce the amount of required memory is to use spatial hashing. Instead of using the calculated cell index as an index into a large array, a hash key is calculated and used to store the object in a hash map. The hash map is a list of buckets where each bucket is associated with a hash key. The hash key for each cell than an object overlaps is calculated and the object is recorded in each associated bucket. Cells that contain no objects use no memory since the objects are stored directly. The next step, after the hash map has been filled, is to traversed the hash map and all objects that have been recorded in the same bucket are tested against each other. An example is shown in Figure 5.6.

The hash key for a cell can, for example, be calculated using the following expression:

$$k = (c_x \cdot p_x + c_y \cdot p_y + c_z \cdot p_z) \bmod n \tag{5.2}$$

where $p_x$, $p_y$, and $p_z$ are large prime numbers. Other hash functions are also possible.

The size of the hash map, $n$, is of vital importance for the performance of the collision detection algorithm. If too small, then many cells will be stored in the same bucket, but a too large size reduces performance because of increased memory usage.

Figure 5.6 shows a situation where five objects are located in a two dimensional cell grid where each cell has been given a hash key between zero and nine. The resulting hash list is shown on the right of the image.

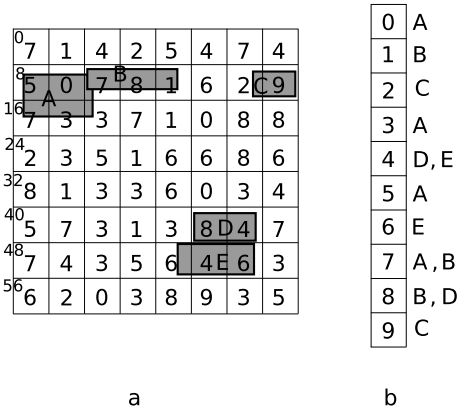


Figure 5.6: *Example scenario in two dimensions and the hash list created by the spatial hashing method. a) Grid layout with objects. The number written in each cell is the hash key for that cell and the cell indices for the first column of cells are shown in smaller font and increases successively to the right (not shown). b) Resulting hash list where the storage location of each object is shown. Collision checks are required for cells given hash keys 4, 7 and 8.*

An alternative to explicitly storing the cells is possible when the difference in the sizes of the objects are small. Each object is given a *home cell* based on the center point of the object and by making the cells larger than the largest object, two objects can not collide unless they have the same or neighboring home cells. Collision detection is done by calculating the home cell of each object and then sorting the list of objects based on the home cell. After sorting, all objects that have the same home cell are stored consecutively in memory and the start and end of each non-empty cell can be found. The only thing that remains is to iterate over the non-empty cells and test all particles in each cell against the other particles in that cell and the particles in the

neighboring cells.

Figure 5.7 shows an example where this method is used. When checking for collisions, the algorithm picks an entry from the list of home cells and tests the found objects against the other objects with the same or neighboring home cell. For example, when inspecting home cell with index 2, the neighboring cells have index 1, 3, 5, 6 and 7. Of these, only the cell with index 3 has an entry in the home cell list, and thus only that cell needs to be tested against the current cell. The union of objects contained in these two cells are objects **b** and **c**, which are tested against each other. To avoid duplicate collisions, each home cell need only test its objects against neighboring cells with higher cell index. Any collision with an object with a home cell with lower index will already have been found when that lower indexed cell was processed. In the example in Figure 5.7, the home cell with index 9 does not need to perform any collisions checks since all non-empty neighboring home cells have a lower cell index.

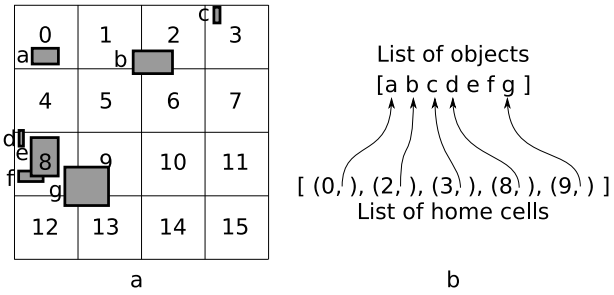


Figure 5.7: *Collision detection using home cells and particle sorting. a) Grid layout with objects. Numbers show cell index and lower-case letters are names on objects. b) Data structures used for collision detection.*

### Recursive space partitioning

A considerable speedup can result if a recursive space partitioning scheme is used when the simulation contains objects of various sizes. The idea is to create the space partitioning based on the shapes and positions of the simulated objects. There are several ways to choose how to perform the splitting and three such ways will be described here. One technique for splitting the space is an octree [49]. An octree is made up of several levels of cubes organized in a tree hierarchy. At the first level, a single node represent the whole simulated space. This node, just as all other non-leaf nodes, has eight children. The children represent the eight subspaces that are created by splitting the space represented by the parent node into eight pieces. The split is always made in three slices, each parallel to one of the coordinate axes, meeting at a single point. This ensures that each piece is a rectangular hexahedron, i.e. a cuboid. For each piece, the process may then be repeated recursively until some stop condition is satisfied, for example that each segment contains a single object. Figure 5.8 shows an example where a two-level octree has been created with the highest division level in one of the top corners.

A similar technique is the kd-tree presented by Bentley [5]. The kd-tree also splits the simulated space into cuboids but splits each node once, making the kd-tree a binary tree. The creation process is similar to the one for octrees, but instead of splitting along every dimension in each step, only one dimension is selected. The choice is arbitrary, where Bentley suggest that the split dimension is chosen in cyclic fashion, starting with dimension 0, then 1 and so on up to dimension k-1, after which dimension 0 is used again. Another suggestion is made by Dikaiakos and Stadel [16], who chose to split each cuboid along its longest dimension.

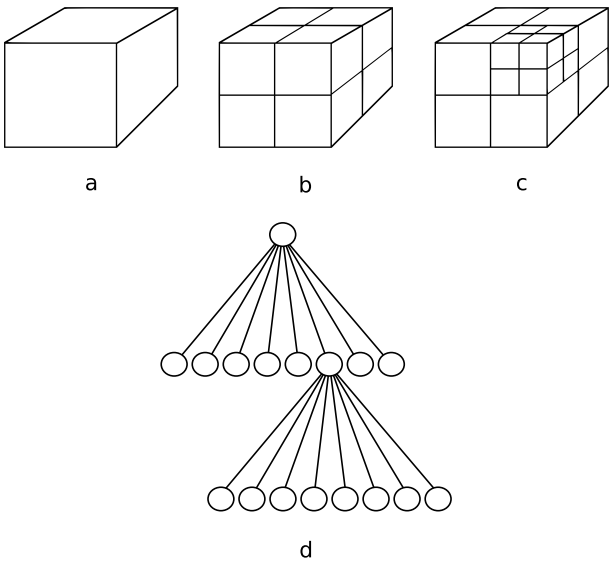


Figure 5.8: *Cubic space subdivided using an octree. a) Original space. b) First division cuts the space in eight pieces. c) The granularity of one of the top corners is increased by dividing that section once more. d) The tree structure created from this set of divisions.*

### 5.1.3 Sweep and prune

This method [11] is also called sort and sweep [22] and builds upon the observation that for two objects to collide, their extent in all dimensions of the simulated space must overlap. If one can find one dimension where the extents of the objects do not overlap, no collision is possible. The algorithm can be explained as follows. Select one dimension of the simulated space, find the beginning and end point along that dimension for each object and insert begin and end markers into a list, eventually containing $2n$ entries where $n$ is the number of simulated objects. The list is sorted in ascending order based on the coordinate stored in the marker and collision detection is performed while traversing the list. Each time a begin marker is found, insert the associated object into a list of active objects and remove the object when its end marker is found. Collision

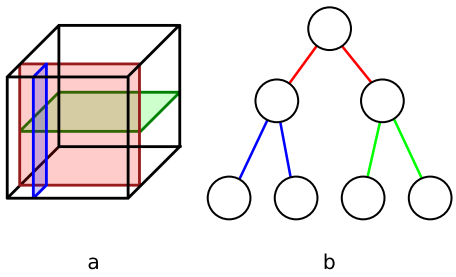


Figure 5.9: *Example of a kd-tree. a) The cubic space is first split along the red plane, creating one large and one thin subsection. The larger subsection is then split by the green plane and the smaller by the blue. b) The tree structure created by this splitting. Each node represents a piece of the simulated space and edges are colored by the edge that split the parent node.*

tests are performed when a new object is inserted into the active list, and then only against the other objects in that list. The choice of which dimension to use is arbitrary, but knowledge about the simulated system can give clues. The best dimension is the one with the smallest amount of overlap of objects projected on that dimension.

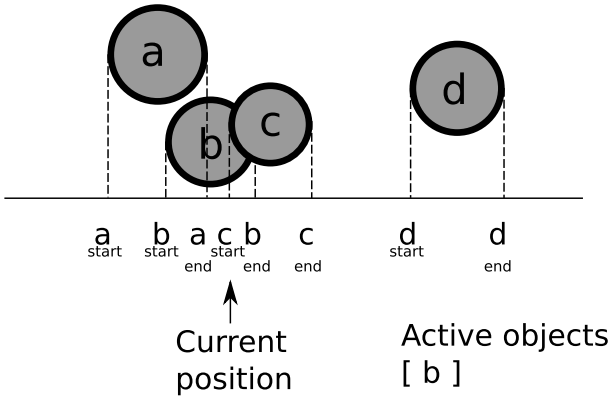


Figure 5.10: *Sweep and prune in action. The first phase of the algorithm has been completed and thus the sorted list of begin- and end markers is available. The figure shows the algorithm in process of traversing the list. Any collision between objects **a** and **b** have already been rejected and now the start marker for object **c** is encountered when the list of active objects only contains **b**. The next thing that will happen is that **c** will be inserted into the active list and tested against **b**, and a collision will be found.*

# Chapter 6

# Methods for Solving Systems of Linear Equations

Many types of numerical methods involve solving a linear system of equations and several methods exist for solving them. A system of linear equations is a set of linear equations that share a set of unknowns and a solution that simultaneously satisfies all the equations. For example, if $\mathbf{x} = [x_1, x_2]^T$ is a vector of unknowns, then

$$5x_1 + 4x_2 = 4$$
$$3x_1 + 7x_2 = 8$$

is an example of a system of linear equations. In matrix form, the system is written as

$$A\mathbf{x} = \mathbf{b} \tag{6.1}$$

where $A$ is called the *system matrix*.

The methods used to find $\mathbf{x}$ can be divided into two major groups: direct solvers and iterative solvers. Direct solvers perform a finite number of operations on the system and at the end, in the absence of round-off errors, produces the desired result, while iterative solvers start with an initial guess $\mathbf{x}^{(0)}$ and iteratively creates a sequence $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(n)}\}$ of (hopefully) ever better approximations [15]. Many iterative methods can be expressed as an iterator function that is applied to the most recent $\mathbf{x}^{(k)}$.

$$\mathbf{x}^{(k+1)} = f(\mathbf{x}^{(k)}) \tag{6.2}$$

The formulation of $f$ is what distinguishes one iterative method from another. Iterative solvers come in two types, stationary and non-stationary. A stationary method is one where the iterator function can be written as $\mathbf{x}^{(k+1)} = \mathbf{c} + T\mathbf{x}^{(k)}$ where $\mathbf{c}$ and $T$ remain unchanged throughout the iterations. When there is no such $T$ and $\mathbf{c}$, the method is a non-stationary one [19]. Any stationary iterative method only converges if the largest eigenvalue of the iteration matrix $T$, the *spectral radius*, is less than one [15].

One drawback of direct methods is their high memory footprint and potentially long execution time, which grows cubically with problem size unless there is much sparsity [32]. If the demand for accuracy in the solution is low, then an iterative method may find a "good enough" solution after a limited number of iterations faster than the time a direct method would require to find a more accurate solution [19]. Another characteristic of direct solvers is the fill-in effect.

The operations that the algorithm performs may write a non-zero value to a location previously occupied by a zero [3]. For matrices stored in a sparse format, this may be a costly operation and increases the amount of required memory. In addition, direct methods, unlike iterative methods, do not map well to the streaming architecture of GPUs.

Iterative methods also have disadvantages. In particular, convergence may be very slow for some problems and the sequence $\{\mathbf{x}^{(n)}\}, n = 1, \ldots$, may even diverge for some very ill-conditioned problems [3], making it impossible to find the desired result. Two very important properties to consider when choosing an iterative solver is the rate of convergence and for which problems the solver is guaranteed to converge. The advantage of iterative solvers is that they consist mostly of matrix-vector operations and do not change the system matrix $A$. This lets the implementation take full advantage of the sparsity of the matrix.

Because high precision is most often not required in physics simulations [19], high performance is the primary focus of this project, and because the matrices used in this project are very sparse and difficult to modify on the highly parallel CUDA platform, only iterative methods will be described here.

## 6.1 The Jacobi method

The Jacobi method is a simple iterative method where each equation is considered independently [3]. In each iteration, equation $i$ is used to update the approximation of $x_i$ using the current approximation for all other unknowns. The update is done by solving for $x_i$ using standard algebraic manipulations.

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right) \tag{6.3}$$

The operation gathers all terms except for the one containing $x_i$ on the right hand side and then divides both sides by the coefficient of the left hand side. The same operation can be expressed using matrix notation and a matrix splitting. The system matrix $A$ is split into two parts

$$A = \Phi + R \tag{6.4}$$

where $\Phi$ contains the diagonal elements of $A$ and $R$ contains the rest of the elements, i.e., off-diagonal elements. All other elements are zero. Eq. (6.1) can now be written as

$$(\Phi + R)\mathbf{x} = \mathbf{b} \tag{6.5}$$

which we rewrite to get an expression closer to the form of Eq. (6.2).

$$\Phi\mathbf{x} + R\mathbf{x} = \mathbf{b} \tag{6.6}$$

$$\Phi\mathbf{x} = \mathbf{b} - R\mathbf{x} \tag{6.7}$$

$$\mathbf{x} = \Phi^{-1}(\mathbf{b} - R\mathbf{x}) \tag{6.8}$$

Since $\Phi$ is a diagonal matrix, it is easy to invert and is reduced to the $\frac{1}{a_{ii}}$ seen in Eq. (6.3). Similarly, since the matrix $R$ contains the elements of $A$ except for the diagonal elements $a_{ii}$, $\mathbf{b} - R\mathbf{x}$ is equivalent to the parenthesized expression in Eq. (6.3).

When iterating, Eq. (6.8) is evaluated again and again, and for each iteration, the approximation of the solution $\mathbf{x}$ calculated in the previous iteration is used in the right hand side. Using the notation for iteration indexing, the update equation is written as

$$\mathbf{x}^{(k+1)} = \Phi^{-1}(\mathbf{b} - R\mathbf{x}^{(k)}) \tag{6.9}$$

This equation is in the form of Eq. (6.2).

## 6.2 The Gauss-Seidel method

By making only a slight variation to the Jacobi algorithm one can derive the Gauss-Seidel iterative method. The fundamental difference between Gauss-Seidel and Jacobi is that Gauss-Seidel does not perform the updates to the unknown vector $\mathbf{x}$ independently for each element. When the first element in $\mathbf{x}$ has been updated, all subsequent updates use the new value in their calculations. This has the effect that the ordering of updates, which was irrelevant when using Jacobi, now becomes important [15]. If an arbitrarily selected equation $i$ is updated first, then it will use only old elements from $\mathbf{x}$ in the update, while if the same equation is updated last, it will necessarily only use the newer values of $\mathbf{x}$. This will produce two different, but both valid, updates to $x_i$. Any equation updated anywhere else in the ordering may use a mix of old and new elements.

The update rule from the Jacobi method is changed to take advantage of the new values of the vector $\mathbf{x}$:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j=0}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n-1} a_{ij} x_j^{(k)} \right) \tag{6.10}$$

The equivalent matrix representation of Eq. (6.10) requires that we split the system matrix $A$ slightly differently than done for the Jacobi method. $A$ is now rewritten as

$$A = \Phi + L + U \tag{6.11}$$

where, again, $\Phi$ contains the diagonal part of $A$. The other elements, however, are split in an upper triangular part $U$ and a lower triangular part $L$. When the equations are processed top-to-bottom and currently working on equation $i$, row $i$ of $U$ contains all the non-zero elements that should be multiplied with the old elements of $\mathbf{x}$ and $L$ contains the non-zero elements that should be multiplied with the newly updated elements of $\mathbf{x}$.

The Gauss-Seidel method is derived in matrix notation as follows:

$$(\Phi + L + U)\mathbf{x} = \mathbf{b} \tag{6.12}$$

$$\Phi\mathbf{x} + L\mathbf{x} + U\mathbf{x} = \mathbf{b} \tag{6.13}$$

$$\Phi\mathbf{x} = b - L\mathbf{x} - U\mathbf{x} \tag{6.14}$$

$$\mathbf{x} = \Phi^{-1}(\mathbf{b} - L\mathbf{x} - U\mathbf{x}) \tag{6.15}$$

The differentiation between different steps in the successive approximation refinement is done as follows.

$$\mathbf{x}^{(k+1)} = \Phi^{-1}(\mathbf{b} - L\mathbf{x}^{(k+1)} - U\mathbf{x}^{(k)}) \tag{6.16}$$

Gauss-Seidel is sequential in nature since each update to an element $\mathbf{x}_i$ requires that the previous elements that equation $i$ depends on have been updated already [19]. This makes it hard to parallelize since sets of equations that are internally independent with respect to a set of unknowns must be found and the amount of concurrency available is limited to the number of

equations per set. The independent sets are easily visualized, as in Figure 6.1, by reordering the equations and unknowns in such a way that the equations within a set are numbered consecutively, and similar for the unknowns. The result is that the new system matrix will have a clear block structure, where the square diagonal blocks are diagonal matrices. All unknowns of one such diagonal block can be updated in parallel, since they are all independent of each other. The blocks, however, must be processed serially [45].

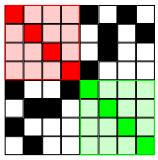

Figure 6.1: *System matrix for two sets of independent equations, colored in red and green. They are independent because each unknown within each diagonal block only appear in a single equation within that block.*

## 6.3 The conjugate gradients method

The conjugate gradient method [48, 19], or CG for short, is different from the Jacobi and Gauss-Seidel methods in that it is not based on a recurrent multiplication between an iteration matrix created from the system matrix and the current approximation. Instead, CG should be viewed as a minimization method that tries to find the minimum on a hyper-surface defined by the following scalar function, called a *quadratic form*:

$$f(\mathbf{x}_a) = \frac{1}{2}\mathbf{x}_a^T A\mathbf{x}_a - \mathbf{b}^T\mathbf{x}_a + c \tag{6.17}$$

where $c$ is any scalar constant and $\mathbf{x}_a$ is any approximation of $\mathbf{x}$. Figure 6.2 shows a graph of Eq. (6.17) for a $2 \times 2$ matrix.

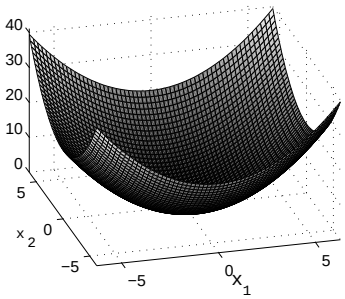


Figure 6.2: *Graph of the quadratic form for a* $2 \times 2$ *system matrix.*

For any positive-definite matrix $A$, the shape of the quadratic form is a paraboloid bowl and thus has a well defined global minimum. The gradient of Eq. (6.17) is

$$f'(\mathbf{x}_a) = \frac{1}{2}A^T\mathbf{x}_a + \frac{1}{2}A\mathbf{x}_a - \mathbf{b} \tag{6.18}$$

which, for symmetric matrices, reduces to

$$f'(\mathbf{x}_a) = A\mathbf{x}_a - \mathbf{b} \tag{6.19}$$

The gradient is a vector field where each vector points in the direction of greatest increase. By setting $f'(\mathbf{x}_a)$ to $\mathbf{0}$, we arrive at the original system that we are trying to solve. This means that the bottom of the parabolic bowl will, for any symmetric positive-definite matrix, coincide with solution of $A\mathbf{x} = \mathbf{b}$. For non-symmetric matrices, the simplification from Eq. (6.18) to Eq. (6.19) is invalid and thus the local minimum may lie elsewhere than on the solution to the original system. CG will then instead find the solution to the system $\frac{1}{2}(A^T + A)\mathbf{x} = \mathbf{b}$. For matrices that are not positive-definite, the shape of the hyper-surface is not a bowl and it will be impossible to find a global minimum, causing CG to fail. Figure 6.3 shows what the quadratic form looks like for an indefinite matrix.

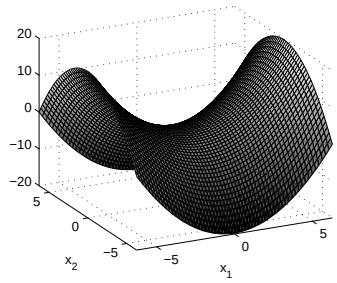


Figure 6.3: *Quadratic form for an indefinite matrix.*

### 6.3.1 Search directions

Arguably, the most obvious direction to move an approximation $\mathbf{x}^{(k)}$ is in the direction of steepest descent. That is, to move $\mathbf{x}^{(k)}$ in the direction of the negative gradient. The iteration function for this method, called *gradient descent*, or just *steepest descent*, is

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha f'(\mathbf{x}^{(k)}) \tag{6.20}$$

which, by using Eq. (6.19), can be formulated as

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha(\mathbf{b} - A\mathbf{x}^{(k)}) \tag{6.21}$$

The step length $\alpha$ is chosen such that the gradient of the new approximation, $f'(\mathbf{x}^{(k+1)})$, is orthogonal to that of the old approximation, $f'(\mathbf{x}^{(k)})$. This ensures that $\mathbf{x}^{(k+1)}$ lies on the lowest possible location along the line defined by the steepest descent of position $\mathbf{x}^{(k)}$.

Steepest descent has the disadvantage that it tends to repeat steps in similar directions. This can be clearly seen in Figure 6.5 where a large number of steps is required since each new solution is thrown back and forth between the two sides of the long valley, only slowly moving towards the minimum point.

A much better strategy would be to choose search directions that do not repeat a previously used one and makes sure that the step taken in each direction makes any more movements in that direction unnecessary. By using orthogonal directions, $\{\mathbf{d}^{(0)}, \mathbf{d}^{(1)}, \ldots, \mathbf{d}^{(n-1)}\}$, there can be no repetition of any direction and the correct answer is found in $n$ steps, where $n$ is the number of unknowns. For each step taken, the length of the step should be chosen so that the error vector

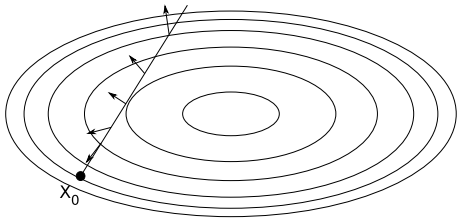


Figure 6.4: *The search line and the gradient at several points along the line. The minimum along the line is found when the gradient is orthogonal to the search line.*

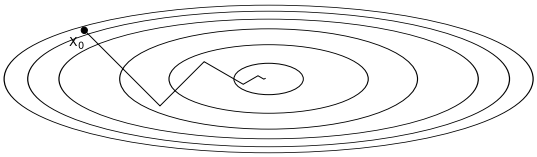


Figure 6.5: *Convergence is slow when similar directions are used several times.*

of the new approximation is orthogonal to the search direction, i.e. $\mathbf{e}^T\mathbf{d}^{(k)} = 0$. This will bring the new approximation as close as possible, along the search direction, to the exact solution. Unfortunately, since knowing the error would be the same as knowing the exact answer, this method can not be used directly.

Instead of requiring orthogonality between the search directions, a practical formulation is obtained by requiring *A-orhogonality*. Two search directions $\mathbf{d}^{(k)}$and $\mathbf{d}^{(l)}$ are $A$-orthogonal if

$$(\mathbf{d}^{(k)})^T A\mathbf{d}^{(l)} = 0 \tag{6.22}$$

This in turn changes the requirements on the step length, which should be such that the new error is $A$-orthogonal to the search direction:

$$(\mathbf{d}^{(k)})^T A\mathbf{e}^{(k+1)} = 0 \tag{6.23}$$

This can be used to find the step length $\alpha^{(k)}$:

$$(\mathbf{d}^{(k)})^T A\mathbf{e}^{(k+1)} = 0 \tag{6.24}$$

$$(\mathbf{d}^{(k)})^T A(\mathbf{e}^{(k)} + \alpha^{(k)}\mathbf{d}^{(k)}) = 0 \tag{6.25}$$

$$\alpha^{(k)} = -\frac{(\mathbf{d}^{(k)})^T A\mathbf{e}^{(k)}}{(\mathbf{d}^{(k)})^T A\mathbf{d}^{(k)}} \tag{6.26}$$

$$\alpha^{(k)} = \frac{(\mathbf{d}^{(k)})^T \mathbf{r}^{(k)}}{(\mathbf{d}^{(k)})^T A\mathbf{d}^{(k)}} \tag{6.27}$$

$$\tag{6.28}$$

The search directions are built from the residuals at each iteration, where the initial search direction is the residual of the initial guess $\mathbf{x}^{(0)}$. For each iteration, the search direction to use in the next iteration is a linear combination of the residual and the current search direction according to the following:

$$\mathbf{d}^{(k+1)} = \mathbf{r}^{(k+1)} + \beta^{(k+1)}\mathbf{d}^{(k)} \tag{6.29}$$

where

$$\beta^{(k+1)} = \frac{(\mathbf{r}^{(k+1)})^T\mathbf{r}^{(k+1)}}{(\mathbf{r}^{(k)})^T\mathbf{r}^{(k)}} \tag{6.30}$$

and

$$\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)} \tag{6.31}$$

$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha^{(k)}A\mathbf{d}^{(k)} \tag{6.32}$$

### 6.3.2 Algorithm

The complete CG algorithm is given in Algorithm 6.3.1.

**Algorithm 6.3.1** The Conjugate Gradient algorithm

Given $A, \mathbf{b}, \mathbf{x}^{(0)}$
$\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$
$\mathbf{d}^{(0)} = \mathbf{r}^{(0)}$
$k = 0$
**repeat**
  $\alpha^{(k)} = (\mathbf{r}^{(k)})^T\mathbf{r}^{(k)} / (\mathbf{d}^{(k)})^T A\mathbf{d}^{(k)}$
  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{d}^{(k)}$
  $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha^{(k)}A\mathbf{d}^{(k)}$
  $\beta^{(k+1)} = (\mathbf{r}^{(k+1)})^T\mathbf{r}^{(k+1)} / (\mathbf{r}^{(k)})^T\mathbf{r}^{(k)}$
  $\mathbf{d}^{(k+1)} = \mathbf{r}^{(k+1)} + \beta^{(k+1)}\mathbf{d}^{(k)}$
  $k = k + 1$
**until** $||\mathbf{r}^{(k+1)}||$ is small enough

Unlike Jacobi and Gauss-Seidel, CG is not a *smoothing* method, meaning that some components of the error term might increase after an early iteration. This manifests itself as jitter in a plot of the error and because of this, CG may return very inaccurate results if too few iterations are performed.

### 6.3.3 Preconditioning

Preconditioning is a technique to increase the rate of convergence of CG. Instead of solving $A\mathbf{x} = \mathbf{b}$ directly, one can solve $M^{-1}A\mathbf{x} = M^{-1}\mathbf{b}$ where $M^{-1}A$ has a lower condition number than $A$ and $M$ is called the preconditioner. There are many ways to create $M$, but a simple method is to us a diagonal matrix whose elements are the same as the diagonal elements of $A$.

The algorithm is only slightly changed to make use of the prconditioner, inserting it at appropriate places. It is listed in Algorithm 6.3.2.

**Algorithm 6.3.2** The preconditioned Conjugate Gradient algorithm

Given $A, \mathbf{b}, \mathbf{x}^{(0)}$
$\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$
$\mathbf{d}^{(0)} = M^{-1}\mathbf{r}^{(0)}$
$k = 0$
**repeat**
  $\alpha^{(k)} = (\mathbf{r}^{(k)})^T M^{-1}\mathbf{r}^{(k)} / (\mathbf{d}^{(k)})^T A\mathbf{d}^{(k)}$
  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{d}^{(k)}$
  $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha^{(k)} A\mathbf{d}^{(k)}$
  $\beta^{(k+1)} = (\mathbf{r}^{(k+1)})^T M^{-1}\mathbf{r}^{(k+1)} / (\mathbf{r}^{(k)})^T M^{-1}\mathbf{r}^{(k)}$
  $\mathbf{d}^{(k+1)} = M^{-1}\mathbf{r}^{(k+1)} + \beta^{(k+1)}\mathbf{d}^{(k)}$
  $k = k + 1$
**until** $||\mathbf{r}^{(k+1)}||$ is small enough

## 6.4 Convergence

The Jacobi and Gauss-Seidel methods are similar and have similar convergence criteria. For both methods, strict row diagonal dominance is sufficient for convergence [15]. That is, for every system matrix $A$ where

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \tag{6.33}$$

is true for every $i$, i.e., every row, Jacobi and Gauss-Seidel both converge. In addition, Gauss-Seidel is guaranteed to converge to the solution even for systems that lack strict row dominance if the system matrix is positive-definite. However, this is not necessarily true for Jacobi and because of this, some problems that Gauss-Seidel can solve causes Jacobi to diverge.

An example of one such matrix is

$$A = \begin{bmatrix} 3 & 2 & 2 \\ 2 & 3 & 2 \\ 2 & 2 & 3 \end{bmatrix} \tag{6.34}$$

which is both symmetric and positive-definite but not strictly row diagonally dominant. The Jacobi iteration matrix for $A$ is

$$T = \begin{bmatrix} 0 & -2/3 & -2/3 \\ -2/3 & 0 & -2/3 \\ -2/3 & -2/3 & 0 \end{bmatrix} \tag{6.35}$$

To determine if a recurrent application of $T$ to a vector will converge or diverge, the eigenvalues of $T$ must be inspected. If the aboslute value of any eigenvalue is greater than one, then Jacobi fails. The eigenvalues are found by finding the roots of the characteristic polynomial

$$\mathtt{det}(\lambda I - T) = \lambda^3 - \frac{4}{3}\lambda + 16/27 \tag{6.36}$$

which are 2/3 and -4/3. The spectral radius of $T$ is therefore $4/3$, which is larger than one and causes Jacobi to diverge.

For CG, a symmetric positive-definite system matrix $A$ is both required and sufficient for convergence [48].

# Chapter 7

# Implementation

In this chapter the implementation of the four main parts of the system is described. The first section describes the bulk of the fluid simulation, which includes data structures, the ordering of events, state updates, and how to include a fluid simulation into an application. The second section concerns the collision detection part and include motivations for why a particular method was chosen and an in-depth description of how it works. The next section describes the solver and the final section the demonstrator, including fluid setup, user interactions, and rendering.

## 7.1 Constraint fluid

The constraint fluid library consists of a stack of classes and functions as shown in Figure 7.1, and at the highest level is the `ConstraintFluid` class. A fluid simulator is created by creating a new instance of this class and it is the interface that the application uses to communicate with the fluid simulator. The second layer of the stack is the helper classes that the top level class uses to perform the different parts of the fluid simulator. Two of these helpers, the `CollisionDetector` and the different types of solvers, are described in coming sections and the last will be given a shorter description in this section. It is the helper objects that launches the CUDA kernels, which constitutes the last two layers of the stack. The helper objects call kernel wrappers which perform texture bindings, validity checks, grid block setups, and finally launches an actual CUDA kernel, which is the last layer of the stack.

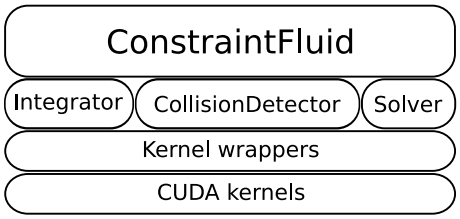


Figure 7.1: *The different parts of the library.*

### 7.1.1 Application integration

This section contains a brief description of what is required to insert a fluid simulation into an application. The text is not to be considered exhaustive and an example application is supplied

together with the rest of the source code.

All the definitions an application needs in order to use the constraint fluid is supplied in the header file `constraintFluid.h`. In this file, two definitions are of utmost importance to the user. One is the definition of the `ConstraintFluid` class itself, but equally important is a structure named `ConstraintFluidParameters`. This structure is used to control the behavior of the fluid and is passed to the `ConstraintFluid` constructor. It includes parameters such as the number of particles to be simulated, the size of the container, all parameters used in Eq. (4.7) and the strength of the gravity force. When the structure has been filled, the application can continue to create the `ConstraintFluid` object. In addition to the parameters structure, a solver must be specified. The `ConstraintFluid` class contains a public `enum` that defines the supported solvers and one of these should be passed as the second argument to the constructor.

When a fluid simulator has been created, only two method calls are required to finish the integration with the application. The first is `stepSimulation`, which does a complete simulation step, and the second is `getParticlePositionsVBO`, which returns the identifier of the OpenGL vertex buffer where the current particle positions are stored. It is up to the application to render the particles in an appropriate manner. There is currently no support for DirectX integration, but the development effort required is limited to implementing a subclass of `DenseBlockVector` that uses a DirectX vertex buffer for storage and replace the use of `DenseBlockOpenGLVector` with the new vector type.

The simulator supplies a number of additional methods that can be used to inspect the state of the simulated fluid, for example the density violations of each particle, the distribution of the number of neighbors among the particles and the number of simulation steps performed. In addition, some of the aspects of the simulation can be changed during runtime. This is currently limited to the gravity vector and the size and position of the container.

### 7.1.2 Data structures

Two types of data structures are fundamental to the implementation of the fluid simulator: a vector and a sparse matrix. Vectors are used for storage of, for example, particle positions and velocities, constraint violations, and the calculated Lagrange multipliers. It is important to differentiate between vectors as the CUDA data type and the different vector classes, which may be a vector of for example `float4`s. The sparse matrix is used only to represent the Jacobian matrix.

#### Vectors

Vectors are the main data holders of the fluid simulation library. All data related to a single particle or a single constraint are stored in a vector. Vectors can be of two types, element vectors and block vectors, and be stored in two different ways: in memory allocated by CUDA allocations or as OpenGL vertex buffers. There is currently no support for Direct3D integration. OpenGL buffers are used for particle positions and colors since this allows an application that uses the library to render particles efficiently through the OpenGL calls `glVertexPointer`, `glColorPointer` and `glDrawArrays`. Abstractions has been made though object oriented design and inheritance to allow any non-OpenGL code to use these vectors without knowing where the actual data is stored.

The difference between an element vector and a block vector is that the element vector is a vector of `float`s and the block vector is a vector of `float4`s. The use of block vectors makes it easier to find for example the position of a given particle and memory read and writes are easy to coalesce since coalescing is achieved when each thread of a warp reads from or writes

to the index dictated by its thread ID. If simple `float` arrays were used, the threads of a thread block would have to cooperatively read all required data to shared memory, synchronize and then distribute the data from shared memory to each of the threads. Since shared memory is a limited resource, this approach may limit the number of blocks that can reside on an SM and thus hamper the hardware's ability to hide memory latency.

The vector class contains methods to fill the device memory area with data from host memory, read the content of a vector from device memory to host memory, and print the content of a vector to screen or disk. It can also supply a raw pointer in CUDA address space that can be used in CUDA kernels to read from and write to the memory allocated for a vector. The OpenGL vectors can also supply the `GLuint` that identifies the buffer object and is a required argument to OpenGL calls that use the buffer.

### Sparse matrix

The sparse matrix class developed for this project is tailored to the requirements of the fluid simulation, designed specifically to hold the Jacobian matrix $G$ for the particular set of constraints used in this fluid simulation and optimized for matrix vector and transposed matrix vector multiplication, which are the two most important matrix operations used.

The layout of the matrix we wish to represent is as follows. The matrix contains one row for each constraint in the simulation and each row contains one block of elements for each particle, as described in Chapter 4 where the constraint fluid algorithm is presented. The particles in the simulation can produce two constraints each, one density constraint and one non-penetration constraint, and the matrix can therefore contain up to $2n$ rows. The density constraints are multi-body constraints and the rows created by them may therefore contain several non-zero blocks, but the rows created by non-penetration constraints only contain one non-zero block. The ordering of the rows is arbitrary in general, but the implementation always places the density constraints at the top and the non-penetration constraints in the bottom rows. Figure 7.2a shows the layout of non-zero blocks.

The implementation makes some assumptions about and impose some limitations on the data stored in the matrix. First, the density constraint part of the matrix must be skew symmetric, and second, that no row in the density constraint part contains more than a preconfigured maximum number of non-zero blocks. The assumption on skew-symmetry is no limitation in the simulation since it holds true by the definition of Eq. (4.5), which defines the matrix, but the limitation on blocks per row in density constraints can be a problem. In essence it means that each particle can have only a limited number of neighbors and any neighbor found after that limit has been reached is ignored. Because of the chosen collision detection algorithm and hashing function, the result is that particles with high density, which happens to particles with many neighbors, are forced upwards. This is because neighbors are inserted in hash key order and lower particles have a lower hash key except for at the cell grid roof. The actual limit is configurable and is currently set to 32. Experimentation has shown that the maximum number of neighbors in a typical simulation tends to stay around the low twenties and rarely approaches thirty.

The storage of the matrix is split into three parts: one for the non-penetration constraints, one for the diagonal elements in the density rows, and one for the off-diagonal elements from the same rows. The non-penetration and diagonal blocks are stored as `float4` arrays allocated large enough to handle the worst case scenario where all constraints are active at once. The off-diagonal blocks are stored in a format called "simplified jagged diagonal storage" [17]. The format is a packed column major format where the first non-zero block from each row is stored first in memory, allocated as a single long `float4` array. Then the second non-zero block from

each row is stored and so on until the selected maximum number of blocks has been reached. Zeros are explicitly stored when a row runs out of non-zero blocks. Paired with the non-zero blocks is an array of integers that specifies the column index of the stored block. The layout of these integers in memory is exactly the same as the layout of the blocks they belong to. Also, the sparse matrix class stores an array that contains the number of blocks in each row. By inspecting this value one does not need to read and operate on the extra zeros that were stored when a row contained less than the maximum number of blocks.

One can view the storage format as a list of segments in memory where each row owns one slot in each segment and there are as many segments as the maximum number of blocks per row. When creating the matrix each row iterates over the non-zero blocks it has and places each block in the next free slot owned by that row. The non-zero block at location $(i, j)$ is written to memory location

$$k = i + nc \tag{7.1}$$

where $c$ is the number of non-zero blocks to the left of the current block and $n$ is the number of particles. The value $j$ is written to the same location, $k$, in the column index array.

The reason for choosing the simplified storage version which stores these extra zero blocks is to simplify the creation process. Many other sparse storage schemes requires knowledge about the whole matrix before any elements can be written, but any row can be created independently of the others when using the simplified jagged diagonal storage format. This fits well with the collision detection algorithm, which handles the creation of the matrix, described in Section 7.2.

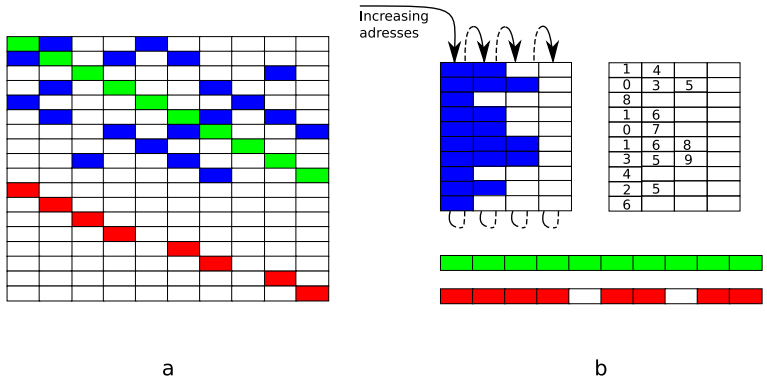


Figure 7.2: *An example matrix and how it is stored in memory. a) The full matrix. White blocks are zero elements, red blocks are non-penetration blocks, green ones are the diagonal of the density section of the matrix and finally the blue blocks are the off-diagonal blocks of the same section. b) The diagonal- and non-penetration blocks are stored as simple block vectors. The off-diagonal blocks are stored in a column major format in a single long array. Paired with the array is another array that, for each non-zero off-diagonal block, stores the column index in the original matrix where that block belongs.*

### 7.1.3 Arithmetic operations

All the algorithms used throughout the library uses a multitude of operations on the vectors such as addition, scaling, and subtraction. Most of these are trivial but the operations that involve the sparse matrix are not. The two operations used are matrix-vector multiplication and transposed

matrix-vector multiplication. In this subsection these two operations will be described along with how they are implemented using the data structures introduced in the preceding subsection.

### Matrix-vector multiplication

The matrix-vector multiplication performs the operation $\mathbf{y} \leftarrow G\mathbf{x}$. It uses a matrix $G$ of size $c \times n$ and a vector $\mathbf{x}$ of length $n$ to produce a new vector $\mathbf{y}$ of length $c$. Each element in $\mathbf{y}$ contains the dot product between the input vector and one of the rows of the input matrix. The $i$th element of $\mathbf{y}$ is the dot product of $\mathbf{x}$ and the $i$th row of $G$. The size of the matrix is dictated by the number of simulated particles $n$ and is always $2n \times n$ blocks. The multiplication operator is only defined if the input vector has the same length as each row in $G$ and thus the input vector must be a block vector with $n$ blocks. The result will be an element vector of length $2n$. The algorithm for performing the matrix-vector product is given below where `G_d` is the diagonal blocks of the density constraints part of the matrix, `G_o` is the off-diagonal blocks and `G_n` is the non-penetration blocks. The data type `block` used in the listing is actually one of the primitive CUDA vectors, but renamed here for simplicity. Each CUDA thread computes one element of $\mathbf{y}$.

Figure 7.3 illustrates the matrix-vector multiplication operation.

Listing 7.1: $\mathbf{y} \leftarrow G\mathbf{x}$

```
with one thread per particle {
  //Indices are on the block level

  int i = thread_id;

  //Start with the diagonal block
  block accumulator = G_d[i] * x[i];  //Per element multiply

  //Then loop over the remaining blocks
  int rowLength = rowLengths[i];
  for(uint itr = 0; itr < rowLength; ++itr) {
    int index = itr*numParticles + i;
    int column = columns[index];

    block G_block = G_o[index];
    block x_block = x[column];
    accumulator += G_block * x_block;  //Per element multiply-add
  }

  //Write the result
  y[i] = sumElements(accumulator);

  //Finish with the non-penetration block
  y[i+num_particles] = sumElements( G_n[i]*x[i] );
}
```

### Transposed matrix-vector multiplication

The transposed matrix-vector product $\mathbf{y} \leftarrow G^T\mathbf{x}$ is similar to the normal matrix-vector product with the difference that each element in the output vector $\mathbf{y}$ is created from columns of the input matrix $G$ instead of rows. Because of the definition of matrix-vector multiplication, the input vector $\mathbf{x}$ must be an element vector of length $2n$ and the result vector $\mathbf{y}$ is a block vector containing $n$ blocks.

Because of the block skew-symmetric property of the top half of the matrix, the content of the columns can be found by iterating over the rows and negating all non-diagonal blocks. For the lower half, the non-penetration block can be found, just as done in the normal matrix-vector

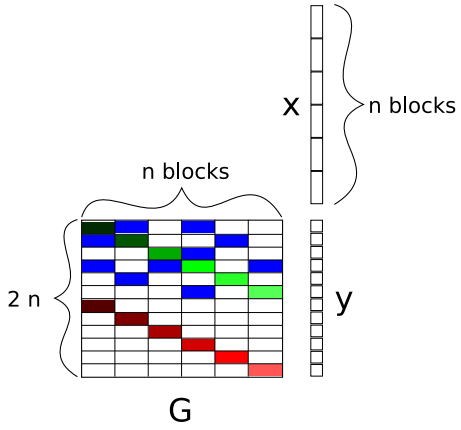


Figure 7.3: *Example of a matrix-vector multiplication. Blocks are colored using the pattern as described in Figure 7.2 with color gradients to simplify comparison with Figure 7.4*

product, by reading a block from the non-penetration array at the index dictated by the thread ID.

Each time a row block is read, data from three columns of the matrix $G$ is collected and three components of the result vector $y$ can be updated.

Figure 7.4 illustrates the transposed matrix-vector operation and each CUDA thread executes Listing 7.2, calculating one element of the output vector $\mathbf{y}$.

Listing 7.2: $\mathbf{y} \leftarrow G^T\mathbf{x}$

```
with one thread per particle {

  int i = thread_id;

  //Start with the diagonal block
  block accumulator = x[i] * G_d[i]; //Scale the block with the scalar found in x

  //Then add the non-penetration block
  accumulator += x[i] * G_n[i];

  //Finally loop over the remaining elements, negating them to get a column instead of a row
  int rowLength = rowLengths[i];
  for(int itr = 0; itr < rowLength; ++itr) {
    int index = itr*numParticles + i;
    int column = columns[index];

    float x_value = x[column];
    block G_block = G_o[index];
    G_block = -G_block;

    accumulator += x_value * G_block;  //Scale the block by the value found in x
  }

  y[i] = accumulator; //Write the complete block to y
}
```

### 7.1.4 ConstraintFluid class

The `ConstraintFluid` class does little work of its own. It is responsible for allocating and initializing the data structures that are passed between helper objects, such as particle positions and velocities, and creating the helper objects themselves that it uses for the bulk of the work.

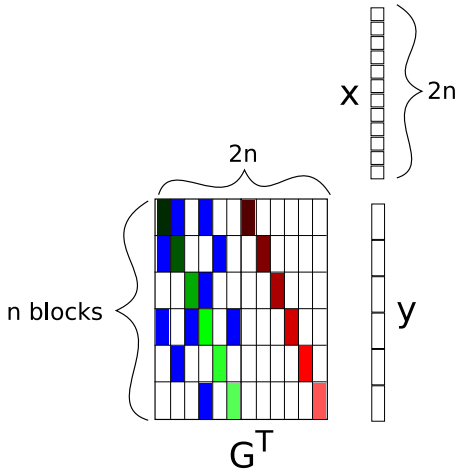


Figure 7.4: *Example transposed matrix-vector multiplication. Blocks are colored in the same way as in Figure 7.3.*

When a simulation step is performed, the first step is to let the collision detector process all collisions in the scene. This fills the constraint violations vector **g**, the inverse of the diagonal of the Schur complement matrix, $\Phi^{-1}$, and the Jacobian matrix $G$. In the process, it reorders the particle positions and velocities as described in Section 7.2. After the collision detection step is complete, the elements of the $\lambda$ vector may be reordered to match the new ordering of the particles. This is required for warm starting, which currently is done only for the Jacobi solver.

Next the solver is called and depending on which solver is used, different parts of the simulation is performed. The simplest solver, Jacobi, simply calculates and returns the Lagrange multipliers $\lambda$. The other solvers, CG and preconditioned CG, include state updates in their algorithm. So when Jacobi is used, the solver call must be followed by a call to the `Integrator` helper class. The `Integrator` uses the Jacobian matrix $G$, the Lagrange multipliers $\lambda$, and the current system state to move each particle into its next state.

### 7.1.5 Helper objects

The only class described here is the `Integrator`. The larger helper objects, the collision detector and the solvers, are described in their respective sections later in this chapter.

#### Integrator

The `Integrator` is a simple class with a single purpose. Each simulation step, unless it has already been done by the solver, the `Integrator` uses Eq. (4.10) to apply all forces, external and from the constraints, to each particle and updates the particle position according to the new velocity using Eq. (4.11). These are implemented in a single kernel, except for the transposed matrix-vector product $G^T\lambda$, which is done first.

### 7.1.6 Data representation

Since a fluid simulation is a very dynamic process where particles move around unpredictably, bouncing off walls and constantly interacting with new neighbors, many of the data structures

used in the simulation must be updated every time step. The seven pieces of data that are most important are the particle states, including positions **p**, velocities **v**, and external forces **f**, the Jacobian matrix $G$, the inverse diagonal of the Schur complement matrix $S_\varepsilon$, called $\Phi^{-1}$, the constraint violations **g**, and the Lagrange multipliers $\lambda$. The particle state is the easiest to represent. Each of the three quantities is given a block vector where the state of particle $i$ is stored in index $i$ of the vector. The other data structures are not so trivially handled. They all contain one entry, block or single element, per constraint currently active and the number of active constraints varies over time. However, there is a maximum number of possible constraints, $2n$, and the implementation prepares for the worst case scenario by allocating enough space to handle all of them. Each such data structure is logically divided into two segments where the first $n$ positions contain data related to the density constraints and the remaining $n$ positions contain entries for the non-penetration constraints. Also, within each segment, the constraints are sorted based on the particle that produced the constraint. That is, position $i$ in the structures contains information about the constraint concerning the density of particle $i$ and position $n+i$ contains information about the non-penetration constraint for the same particle.

Because the structures contain entries for all possible constraints, they may contain "ghost" entries that are not really there. This happens when a particle is free from all walls, in which case it has no non-penetration constraint, and when a particle have no neighbors, in which case it has no density constraint. In the matrix $G$ these entries show up as zero-only rows and in the constraint violations vector **g** they are zero elements. In $\Phi^{-1}$, ghost entries are marked with `inf`. These `inf` markers are detected by the solvers and the corresponding element in $\lambda$ is set to zero when one is found. In this way, the constraint forces created by the ghost constraints are also forced to zero and they have no effect on the simulation.

The simulation parameters $\Sigma$ and $\Upsilon$ described in Section 4.2.3 are treated as matrices in the mathematical discussions throughout this text, but since they are diagonal matrices containing only a limited number of distinct values, they are stored as scalar constants. $\Upsilon$ has the same scalar value everywhere and can therefore be stored as a single scalar, but $\Sigma$ can be different for both types of constraints and therefore requires two scalars for this implementation.

Table 7.1 summarizes the data structures described in this section.

| Name | Symbol | Type | Size |
|---|---|---|---|
| Positions | **p** | Block vector | $n$ |
| Velocities | **v** | Block vector | $n$ |
| External forces | **f** | Block vector | $n$ |
| Jacobian matrix | $G$ | Sparse block matrix | $2n \times n$ |
| Inverse diagonal | $\Phi^{-1}$ | Element vector | $2n$ |
| Constraint violations | **g** | Element vector | $2n$ |
| Lagrange multipliers | $\lambda$ | Element vector | $2n$ |
| Constraint compliance | $\Sigma$ | Two scalars | 2 |
| Constraint decay rate | $\Upsilon$ | Scalar | 1 |

Table 7.1: *List of the data organized by the highest layer in the library stack*

## 7.2 Collision detection

This section describes how the collision detection algorithm works and how the CUDA implementation is partitioned into kernels. Parts of the implementation is based on the "particles" example supplied with the CUDA SDK.

A number of simplifications have been done for the collision detection in this project. First, objects are spheres and there is therefore no need for any bounding volumes. Also, all objects have the same size which is the perfect situation for a space partitioning scheme with a uniform grid. The memory requirements have been reduced by using spatial hashing and the particle sorting technique introduced in Section 5.1.2. Because of the formulation of SPH, described in Section 4.1, the collision detection should not return objects that overlap, but instead report collisions where one object overlaps the *center* of another object. This is achieved by reporting only half the particle diameter to the collision detection algorithm. When two half-sized spheres touch, two full sized spheres with the same positions will touch each others centers. Thus, a good cell size is the particle radius, $h$.

### 7.2.1 Algorithm

The input to the collision detection algorithm is a list of particle positions, $\mathbf{p}$, the radius of the particles, $h$, and the size of the spatial grid, $\mathbf{w}$. Output from the algorithm is a set of collisions, but the collisions are not explicitly stored anywhere. They are instead used directly to create the data structures required at a later stage of the simulation: the Jacobian matrix $G$, the constraint violations $\mathbf{g}$, and $\Phi^{-1}$, the inverse diagonal part of the Schur complement matrix $S_\varepsilon$.

The algorithm is based on a spatial hashing technique where the centroid of each particle is used to calculate a cell hash key for the particle. The list of particles is then sorted based on this hash, resulting in a list where particles residing in the same cell are stored consecutively in memory. The start- and end indices of each cell in the list of particles is found during the sorting process. Next the algorithm picks a particle and finds the hashes of the home- and neighboring cells, as well as the start- and end indices into the sorted list of particles for those cells. The final step of the algorithm is to test the picked particle against the other particles found in the index ranges defined by the start- and end indices and record any collisions in the output data structures.

Figure 7.5 shows the different lists created and how they are related to each other and Figure 7.6 shows how the sorted hash list is used to reorder the list of particles.

### 7.2.2 Data representation

All data structures in the collision detection algorithm are implemented as arrays of one of the primitive data types `uint` or `float`, or their corresponding vector variants `uint2` and `float4`.

Table 7.2 show the arrays required, their sizes, types, and what they represent.

The reason for using `float4` instead of `float3` for particle positions is that memory transactions to and from global memory can be performed much more efficiently when using using coalesced memory operations that are possible only if each element accessed is aligned to 32, 64 or 128 bits, which a `float4` is.

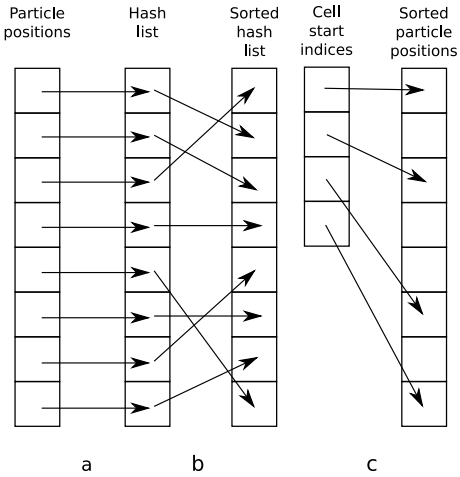


Figure 7.5: *The different lists created during the collision detection algorithm. a) A cell hash/particle index pair is created for each particle. b) The cell hash list is sorted on hash key and the start of each section of equal hash keys are found. c) The list of particles is reordered to match the ordering of the hash list.*

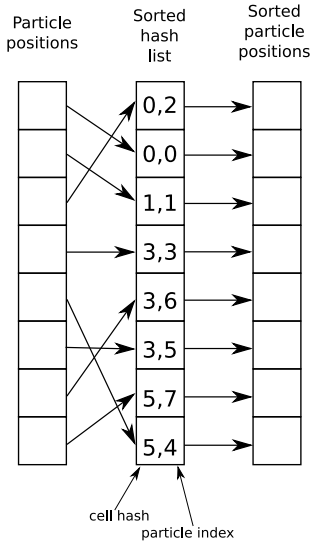


Figure 7.6: *When reordering the particles, an entry from the hash list is inspected to find the index from which a particle should be read, which is shown in the right column of numbers in the figure. Then the particle data is read from that index and written to the same index as where the hash list entry was found.*

| Data | Symbol | Type | Number of elements |
|---|---|---|---|
| Particle positions | **p** | float4 | *numParticles* |
| Sorted particle positions | **q** | float4 | *numParticles* |
| Hash list | **h** | uint2 | *numParticles* |
| Start indices | **s** | uint | *numCells* |
| End indices | **e** | uint | *numCells* |

Table 7.2: *The arrays used by the collision detection implementation, along with their type and number of elements.*

### 7.2.3 Kernels

A high level algorithm description is given in Algorithm 7.2.1 and the rest of this subsection gives a short description of how each of the steps listed in Algorithm 7.2.1 is implemented.

**Algorithm 7.2.1** High level description of the collision detection algorithm.

Calculate the cell hash keys, **h**, for the particles.
Sort **h** on hash key.
Reorder **p**, into **q**, to match the ordering of **h** and find the start and end of each segment with the same hash key.
Perform the narrow phase, creating $G$, **g**, and $\Phi^{-1}$.

#### Hash key calculation

The hash key, $k$, of each particle is calculated from the cell index, **c**, of Eq. (5.1) with the hash function:

$$k = (c_x \bmod w_x) + (c_y \bmod w_y)w_x + (c_z \bmod w_z)w_x w_y \tag{7.2}$$

where $[w_x, w_y, w_z]$ is the number of cells in each grid dimension.

Since the cell index is wrapped if it is outside the cell grid, the hash values generated by this hash function are unique for each cell within the cell grid, starting at 0 and increases up to $num\_cells - 1$. The cell hash function can be seen as a three dimensional stamp that is repeatedly stamped over the simulated world with the first stamp's lower corner at the $[x_0, y_0, z_0]$ point used in Eq. (5.1).

The cell hash, along with the index of the particle, is stored as a pair in a new list, **h**, called the hash list, that is created such that it is initially sorted on particle index.

**Algorithm 7.2.2** Cell hash generation kernel.

Given particle positions **p** and cell grid dimensions **w**.
$i = thread\_id$
$\mathbf{c} = [\left\lfloor \frac{p_x - x_0}{S_x} \right\rfloor, \left\lfloor \frac{p_y - y_0}{S_y} \right\rfloor, \left\lfloor \frac{p_z - z_0}{S_z} \right\rfloor]$ // Eq. (5.1)
$k = (c_x \bmod w_x) + (c_y \bmod w_y)w_x + (c_z \bmod w_z)w_x w_y$ // Eq. (7.2)
$\mathbf{h}_i = [k, i]$

### Hash list sorting

Sorting the hash list is actually done with three kernels that together implement the radix sort algorithm. The source code for this was supplied with the CUDA SDK and is used with permission as stated in the CUDA SDK license agreement. A detailed description of the algorithm is available in GPU Gems 3 [22].

Radix sort is done by counting the number of occurrences of each possible value for a given radix in the input list and then reorder the list, using these counters, so the list becomes sorted with respect to that radix. The process is repeated for each radix and at the end a fully sorted list is produced.

The CUDA implementation is divided into three phases. The first phase performs the radix counting, each thread block counts its own segment of the input list. The second sums these to find reordering offsets for each entry in the list and the third performs the actual reordering, reading an entry from the list, inspecting the value of the current radix and storing the entry at the location indicated by the radix counter. These three phases are repeated as many times as there are radices, four in this case.

### Particle reordering

This kernel has two responsibilities. It both reorders the list of particle positions according to the hash list and finds the indices into that list where each cell starts and ends. Each thread requires access to the hash list entry read by the neighboring thread. The entries are passed between threads through the shared memory, labeled `mem` in Algorithm 7.2.3.

**Algorithm 7.2.3** Particle reordering kernel.

Given the particle positions **p** and the hash list **h**.
$i = thread\_id$
$[k_1, j] = \mathbf{h}_i$
$\texttt{mem}[i+1] = k_1$
Synchronize.
$k_2 = \texttt{mem}[i]$
**if** $k_1 \neq k_2$ **then**
    $s_{k_1} = i$
    $e_{k_2} = i$
**end if**
$\mathbf{q}_i = \mathbf{p}_j$

### Narrow phase

The final step of the algorithm is to find and record all collisions. For each particle, the kernel finds the home cell and all neighboring cells and calculates their hash keys. Then the hash keys are used to index into the start and end indices lists to find the set of indices where particles that the current particle may collide with are stored. For each such set, for each index in the set, the kernel looks up the position of the other particle and performs an exact collision test.

When a collision has been found, the density of the current particle is updated according to Eq. (4.1) and Eq. (4.2), and one block, **b**, in the Jacobian matrix $G$ is created using the upper part of Eq. (4.5). Also, the diagonal block, **d**, is updated by adding one term of the sum in the lower part of Eq. (4.5) and an update to the inverse Schur complement diagonal $\Phi^{-1}$ is made.

When all collisions have been found, the final constraint violation is calculated according to Eq. (4.4).

**Algorithm 7.2.4** Narrow phase kernel

Given the sorted particle positions **q** and the start, **s**, and end, **e**, indices for each cell.
$i = thread\_id$
$\Phi_i^{-1} = 0, \mathbf{b} = \mathbf{0}$
$\rho = mW(0,h)$
**for all** neighboring cells **c** **do**
  $k = (c_x \bmod w_x) + (c_y \bmod w_y)w_x + (c_z \bmod w_z)w_x w_y$ // Eq. (7.2)
  **for all** indices $j \neq i$ between $s_k$ and $e_k$ **do**
    $d = \text{distance}(\mathbf{q}_i, \mathbf{q}_j)$
    **if** $d < h$ **then**
      $\rho = \rho + mW(d,h)$
      $\mathbf{b} = -\frac{mF_{ij}\hat{\mathbf{r}}_{ij}^T}{\rho_0}$ // Eq. (4.5)
      $\mathbf{d} = \mathbf{d} - \mathbf{b}$
      $\Phi_i^{-1} = \Phi_i^{-1} + \mathbf{b} \cdot \mathbf{b}$
      Write **b** to $G$
    **end if**
  **end for**
**end for**
**if** $\Phi_i^{-1} \neq 0$ **then**
  $\Phi_i^{-1} = \frac{\Phi_i^{-1} + \mathbf{d}\cdot\mathbf{d}}{m} + \Sigma_i$
  $\Phi_i^{-1} = \frac{1}{\Phi_i^{-1}}$
**else**
  $\Phi_i^{-1} =$ `inf`
**end if**
$g_i = \frac{\rho}{\rho_0} - 1$
Write **d** to $G$

## 7.3 Solvers

Three iterative solvers have been implemented for this project. The first, and primary, solver is the simple Jacobi method and the others are two versions of the Conjugate Gradient (CG) method.

The equation to solve, described in Chapter 4, is formulated as

$$\left(\frac{1}{m}GG^T + \Sigma\right)\lambda = (\Upsilon - I)G\mathbf{v} - \frac{4}{\Delta t}\Upsilon\mathbf{g} - \frac{\Delta t}{m}G\mathbf{f} \tag{7.3}$$

and should be solved for $\lambda$. The right hand side can be pre-computed and is for the sake of clarity henceforth denoted by **b**. The equation to solve is thus

$$\left(\frac{1}{m}GG^T + \Sigma\right)\lambda = \mathbf{b} \tag{7.4}$$

where $G$ is a sparse, up to $2n \times 3n$ matrix, $m$ is a scalar constant and $\Sigma$ a dense diagonal $2n \times 2n$ matrix. Both $\lambda$ and **b** are dense vectors.

What complicates matters slightly is the fact that $GG^T$ is significantly less sparse than $G$. That, in conjunction with the arithmetic cost, makes it undesirable to explicitly create the complete system matrix $\frac{1}{m}GG^T + \Sigma$. Instead, the full expression is used and algebraic manipulations are done to reduce the number of operations required for each iteration.

As noted in Section 7.1.6, the system may contain invalid elements, called "ghost" entries. It is the responsibility of the solver to ensure that any element in $\lambda$ dependent on such ghost entries are set to zero.

### 7.3.1 Jacobi

The main reason for choosing the Jacobi solver is that it is highly parallel and therefore fits well with the CUDA architecture.

The definition of the Jacobi method, as described in Section 6.1, is

$$\lambda^{(k+1)} = \Phi^{-1}(\mathbf{b} - R\lambda^{(k)}) \tag{7.5}$$

where $\Phi$ is the diagonal part of the system matrix and $R$ contains the rest of the elements. Since the full system matrix is inaccessible, Eq. (7.5) must be rewritten using the components from Eq. (7.4). First, observe that

$$\frac{1}{m}GG^T + \Sigma = \underbrace{\Sigma + \frac{1}{m}\mathtt{diag}(GG^T)}_{\Phi} + \underbrace{\frac{1}{m}GG^T - \frac{1}{m}\mathtt{diag}(GG^T)}_{R} \tag{7.6}$$

where `diag` is a function that returns the diagonal elements of a matrix. Using the definitions from Eq. (7.6), the Jacobi iteration function is found to be

$$\lambda^{(k+1)} = \left(\Sigma + \frac{1}{m}\mathtt{diag}(GG^T)\right)^{-1}\left(\mathbf{b} - \left(\frac{1}{m}GG^T - \frac{1}{m}\mathtt{diag}(GG^T)\right)\lambda^{(k)}\right). \tag{7.7}$$

Since

$$-\frac{1}{m}\mathtt{diag}(GG^T) = \Sigma - \Phi, \tag{7.8}$$

we can rewrite Eq. (7.7) as

$$\lambda^{(k+1)} = \left(\Sigma + \frac{1}{m}\mathtt{diag}(GG^T)\right)^{-1}\left(\mathbf{b} - (\frac{1}{m}GG^T + \Sigma - \Phi)\lambda^{(k)}\right) \tag{7.9}$$

Restoring all $\Phi$s gives

$$\lambda^{(k+1)} = \Phi^{-1}\left(\mathbf{b} - (\frac{1}{m}GG^T + \Sigma - \Phi)\lambda^{(k)}\right) \tag{7.10}$$

which can be rewritten as

$$\lambda^{(k+1)} = \lambda^{(k)} + \Phi^{-1}\left(\mathbf{b} - \Sigma\lambda^{(k)} - \frac{1}{m}GG^T\lambda^{(k)}\right) \tag{7.11}$$

In the end, what is needed is the inverse of the diagonal part of the system matrix, $\Phi^{-1}$, which stays constant throughout the solver process, and the ability to create the product $\frac{1}{m}GG^T\lambda$. The diagonal matrix $\Phi^{-1}$ is created by the collision detector and supplied to the solver as an element

vector and $\frac{1}{m}GG^T\lambda$ is created in several steps, first forming $\mathbf{x} \leftarrow G^T\lambda$ and then $\mathbf{y} \leftarrow G\mathbf{x}$ using the matrix-vector operations described in Section 7.1.3 and finally scaling $\mathbf{y}$ with the scalar $\frac{1}{m}$.

To improve stability of the simulation, only half of the update of Eq. (7.11) is added, resulting in a relaxed Jacobi implementation.

#### Algorithm

The input to the solver is the Jacobian matrix $G$, the constraint violations $\mathbf{g}$, and the inverse Schur diagonal $\Phi^{-1}$, all created by the collision detector, as well as the particles' current velocities, the external forces, and the simulation parameters described in Chapter 4. The solver starts by creating the right hand side and then enters the iteration loop, which it runs a fixed number of iterations. A discussion on the number of required iterations is given in Section 8.3. In the loop, the matrix-vector operations are performed and the result is stored in temporary vectors and then the iteration kernel is launched. The kernel performs the evaluation of Eq. (7.11) that remains, i.e., the per-element additions, subtractions, and multiplications that are required to finish the update of $\lambda$.

**Algorithm 7.3.1** Jacobi solver

Given $G, \mathbf{g}, \mathbf{v}, \mathbf{f}, m, \Delta t, \Upsilon, \Sigma, \lambda^{(0)}$
$\mathbf{t}_1 = G\mathbf{v}$
$\mathbf{t}_2 = G\mathbf{f}$
$\mathbf{b} = (\Upsilon - I)\mathbf{t}_1 - \frac{4}{\Delta t}\Upsilon\mathbf{g} - \frac{\Delta t}{m}\mathbf{t}_2$
$iters = 0$
**repeat**
  $iters = iters + 1$
  $\mathbf{t}_1 = G^T\lambda$
  $\mathbf{t}_2 = G\mathbf{t}_1$
  $\lambda = \lambda + 0.5(\Phi^{-1}(\mathbf{b} - \Sigma\lambda - \frac{1}{m}\mathbf{t}_2))$
**until** *iters* exceeds a maximum limit

The iteration kernel is launched with one thread per constraint, which is the same as the number of elements in the vectors of Eq. (7.11). Thus, each thread reads one element from each vector, performs a series of scalar multiplications, additions, and subtractions to finally produce a piece of the new solution for the Lagrange multipliers $\lambda$. To handle the ghost constraints, the kernel first reads the element in $\Phi^{-1}$ and checks for the `inf` marker. If found, the kernel sets its element in the new $\lambda$ to zero and returns immediately.

### 7.3.2 Conjugate Gradient

To increase development speed and make comparisons with the reference Matlab implementation easier, the CG solver was implemented using as many BLAS calls as possible. BLAS, Basic Linear Algebra Subprograms, is a standardized set of operations on matrices and vectors and a CUDA implementation is supplied with the CUDA SDK. The implementation can of course not work on the application specific sparse matrix format, so matrix-vector operations must use the operations described in Section 7.1.3.

The listing in Algorithm 7.3.2 details the steps of the algorithm.

**Algorithm 7.3.2** The implemented Conjugate Gradient algorithm

Given $G, \mathbf{g}, \mathbf{v}, \mathbf{f}, m, \Delta t, \Upsilon, \Sigma, \lambda^{(0)}$
$RHS_b = m\mathbf{v} + \Delta t\mathbf{f}$
$RHS_c = -\frac{4}{\Delta t}\Upsilon\mathbf{g} + \Upsilon G\mathbf{v}$
$\mathbf{v} = \frac{1}{m}(RHS_b + G^T\lambda)$
$\lambda_{dir} = \Sigma\lambda + G\mathbf{v} - RHS_c$
$\mathbf{r} = -\lambda_{dir}$
$w = \mathbf{r} \cdot \mathbf{r}$
$iters = 0$
**repeat**
  $\mathbf{f}_{dir} = -G^T\lambda_{dir}$
  $\mathbf{v}_{dir} = \frac{1}{m}\mathbf{f}_{dir}$
  $\alpha = \frac{w}{\mathbf{f}_{dir} \cdot \mathbf{v}_{dir} + \lambda_{dir} \cdot \Sigma\lambda_{dir}}$
  $\lambda = \lambda + \alpha\lambda_{dir}$
  $\mathbf{v} = \mathbf{v} + \alpha\mathbf{v}_{dir}$
  $\mathbf{r} = RHS_c - G\mathbf{v} - \Sigma\lambda$
  $w_1 = \mathbf{r} \cdot \mathbf{r}$
  $\beta = \frac{w_1}{w}$
  $\lambda_{dir} = -\mathbf{r} + \beta\lambda_{dir}$
  $w = w_1$
**until** $w$ is small or $iters$ exceeds a maximum limit

### 7.3.3 Preconditioned Conjugate Gradient

The preconditioned version of the conjugate gradient solver is very similar to the plain CG solver described in the preceding subsection. The difference is the inclusion of $\Phi^{-1}$ as a preconditioner. The diagonal matrix $\Phi^{-1}$ contains special markers for the so called ghost entries and these must be set to zero before they are used in any BLAS call. Setting them to zero ensures that the final solution for $\lambda$ will contain zeros at the same locations.

The implemented algorithm is listed below.

## 7.4 Demonstrator

The demonstrator is implemented as a separate application that includes the constraint fluid library and uses it to run a fluid simulation that is rendered on the screen. It uses OpenGL for the rendering and GLUT for user interactions. It is implemented as a startup function that creates the `ConstraintFluid` object, a set of callbacks required by the rendering system and a small set of application specific classes that control the fluid and makes the render calls to OpenGL. The startup function also reads the given command line arguments by which the user can control some aspects of the simulation, such as the number of particles, the size of the container, the simulation frequency, and more.

One of the classes is a simple wrapper over the fluid simulation. Its purpose is to control if the fluid simulation should be updated or not for each frame and if so, make the `stepSimulation` call to the fluid. The other class handles rendering. Each frame, after the fluid wrapper has had its chance to run a simulation step, the renderer fetches the vertex buffer object identifier associated with the buffer storing particle positions and the same for the buffer storing particle colors. It then renders the particles using a shader program supplied with the CUDA SDK and

**Algorithm 7.3.3** The implemented preconditioned Conjugate Gradient algorithm

$RHS_b = m\mathbf{v} + \Delta t\mathbf{f}$
$RHS_c = -\frac{4}{\Delta t}\Upsilon\mathbf{g} + \Upsilon G\mathbf{v}$
$\lambda, \mathbf{e}, \mathbf{e}_{pre}, \mathbf{d}, \mathbf{r}, \mathbf{v}, \mathbf{v}_{tmp} = \mathbf{0}$
$\mathbf{v}_{tmp} = \frac{1}{m}(G^T\lambda + RHS_b) - \mathbf{v}$
$\mathbf{r} = G\mathbf{v}_{tmp} + \Sigma\lambda$
$\mathbf{e} = RHS_c - \mathbf{r}$
$\mathbf{e}_{pre} = \Phi^{-1}\mathbf{e}$
$d_1 = \mathbf{e}_{pre} \cdot \mathbf{e}$
$d_2 = d_1$
$iters = 0$
**repeat**
  $iters = iters + 1$
  $\beta = d_2/d_1$
  $d_1 = d_2$
  $\mathbf{d} = \mathbf{e}_{pre} + \beta\mathbf{d}$
  $\mathbf{v}_{tmp} = \frac{1}{m}G^T\mathbf{d}$
  $\mathbf{r} = G\mathbf{v}_{tmp} + \Sigma\mathbf{d}$
  $\gamma = \mathbf{d} \cdot \mathbf{r}$
  $\alpha = \frac{d_1}{\gamma}$
  $\lambda = \lambda + \alpha\mathbf{d}$
  $\mathbf{e} = \mathbf{e} - \alpha\mathbf{r}$
  $\mathbf{e}_{pre} = \Phi^{-1}\mathbf{e}$
  $d_2 = \mathbf{e}_{pre} \cdot \mathbf{e}$
**until** $d_2$ is small or *iters* exceeds a maximum limit

used with permission as stated in the CUDA SDK end user licence agreement. The end result is that particles are rendered as spheres colored by the current density. By default, the radius of the rendered spheres are set to half the influence radius. This means that spheres that barely touch each other represent particles that have a very small influence on each other, but also that overlaps between particles are larger than they appear. This is most apparent in the splashes of the screen captures in Section 8.4.

# Chapter 8

# Results

The focus of this project is performance and the goal is to simulate hundreds of thousands of particles at interactive rates. However, no specifics was given in the specification on the number of frames per second that is considered interactive rates. For the sake of this discussion, anything more than ten frames per second will be regarded as interactive. The following two sections discusses how well this goal has been met, and also some scalability limitations on the number of particles that can be simulated. In particular, it is the memory requirements that poses a hard limit on the size of the simulated system. In the last section, the demonstrator application is evaluated and compared to the goal formulations that mention it.

## 8.1 Performance

A series of tests have been conducted and the time required for the different parts of the library measured using the timing capabilities of the CUDA device. All tests were run on an NVIDIA GTX 280 graphics card hosted by an Intel Core i7 processor. The timings only include the simulation itself and not other aspects of the application such as rendering and user interaction.

The tests were run with an increasing number of particles placed inside a container with a square bottom large enough to create a fluid between twenty and thirty particles deep, if the test contained enough particles to do that. The Jacobi solver was used exclusively for this test, see Section 8.3 below for a performance comparison between the different solvers, using twenty iterations per time step. The simulation frequency was three hundred time steps per second and the simulation parameters set according to Table 8.1. Table 8.2 lists the test configurations that were run. Each simulation was run for several hundred time steps until the fluid had calmed down and ten frames after that were timed. All ten data points are plotted in the the following graphs.

The series of figures that follow illustrates the results of these tests. First, Figure 8.1 shows the time required for the whole simulation update, counted from the beginning of the call to `ConstraintFluid::stepSimulation()` until the return to the application. The graph shows the total time, as well as the time required for the individual parts. It is clear that the solver requires the most time, followed by the collision detection. Data reordering and state update are negligible. The figure also indicates a clear linear relationship between simulation time and the number of particles. Figure 8.2 shows the total time per simulated particle. It shows that more particles let the application utilize the parallel hardware more efficiently and that at least 25,000 particles should be simulated for high efficiency.

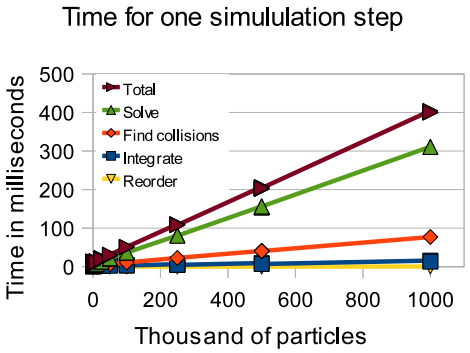


Figure 8.1: *The time required to perform a state update for an increasing number of particles. The time for the four components of a state update is also shown.*

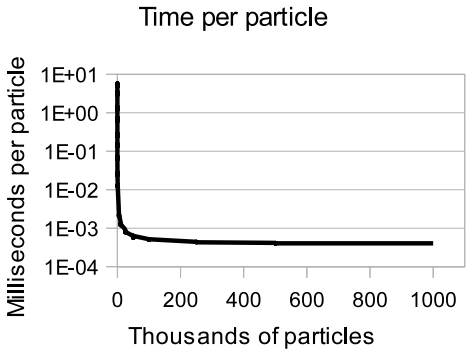


Figure 8.2: *The time required to perform a complete update divided by the number of simulated particles.*

The remaining figures divides the data sets of Figure 8.1 into their respective components. Figure 8.3 shows how the time spent in the collision detector is distributed. The most costly operation for the collision detector is the narrow phase, where each particle is tested against its neighbors and the output data structures are created. Due to implementation details in the timer framework created for this project, the timings for the solver can not produce timings for each individual solver iteration. Instead, the time for all solver iterations is returned and compared with the time to create the right hand side, which is shown in Figure 8.4. Figure 8.5 compares the cost of one solver iteration with the other solver-related operations by dividing the recorded time for the solver iterations by the number of iterations. Section 8.3.1 contains a more detailed discussion about the time requirements of the solvers. Finally, Figure 8.6 shows timings for the operations performed by the integrator.

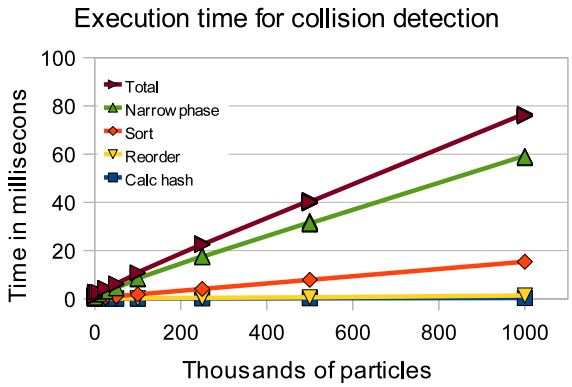


Figure 8.3: *Timings for the collision detection. The narrow phase, where the output data is generated, is by a large margin the most costly operation.*

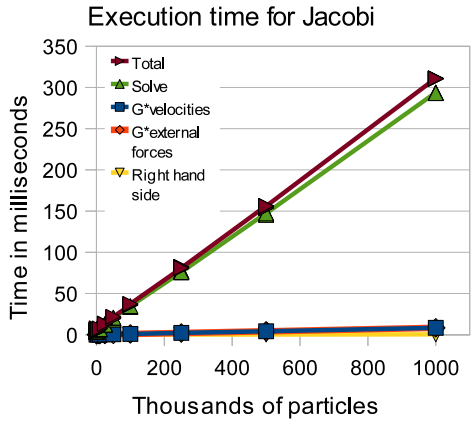


Figure 8.4: *Timings for 20 iterations of the Jacobi solver. The iterations are significantly more expensive than the creation of the right hand side.*

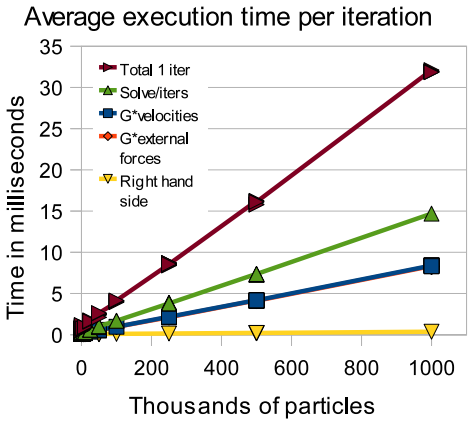


Figure 8.5: *In this graph, the time for the solver iterations has been divided by the number of iterations to make a comparison between a single iteration and other parts of the library possible.*

| | |
|---|---|
| $\varepsilon$ | 0.0001 |
| $\tau$ | 4 |
| Frequency | 300 Hz |
| Solver iterations | 20 |

Table 8.1: *Simulation parameters for the performance tests.*

To conclude the performance section, the following was stated about performance in the Problem statement section of Chapter 2.

*The primary goal of the project is to have a working demonstrator for the fluid that can simulate hundreds of thousands of particles at interactive rates and even larger systems at non-interactive rates.*

The tests have shown that 100,000 particles can be simulated at about 20 frames per second and 250,000 particles at about 10 frames per second. While far from real time, this is still considered to be interactive. The test containing one million particles is an example of what is referred to as an “even larger system” in the problem description and this scenario can be simulated at 2.5 frames per second.

### 8.1.1 CPU comparison

A performance comparison has been made with a CPU implementation available at Algoryx. The CPU implementation uses a Gauss-Seidel solver, performing five iterations per time step, and was run on a 2.8 GHz Xeon processor. Prior usage of the CPU based simulator has shown that five iterations are sufficient for stable simulation when using a Gauss-Seidel solver and five iterations are therefore used here as well. The results from these tests are shown in Figures 8.7 and 8.8.

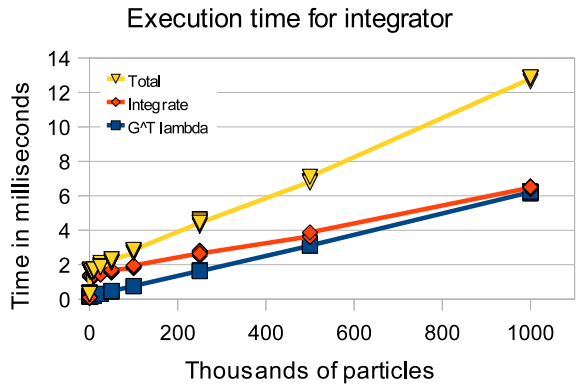


Figure 8.6: *Timings for state updates.*

| n | Container size (m) | n | Container size (m) |
|---|---|---|---|
| 1 | 0.03 | 1,000 | 0.15 |
| 2 | 0.03 | 5,000 | 0.3 |
| 5 | 0.06 | 10,000 | 0.45 |
| 10 | 0.06 | 25,000 | 0.6 |
| 25 | 0.09 | 50,000 | 0.9 |
| 50 | 0.09 | 100,000 | 1.2 |
| 100 | 0.09 | 250,000 | 2.1 |
| 250 | 0.09 | 500,000 | 2.7 |
| 500 | 0.09 | 1,000,000 | 4.2 |

Table 8.2: *Number of particles and container sizes for each performance test.*

Tests were run with up to 250,000 particles and it is clear that the GPU implementation is faster, even with the difference in the number of solver iterations performed. Figure 8.8 shows the speedup that the GPU implementation gives compared to the CPU implementation. As the size of the system increases, so does the speedup and for the largest system, 250,000 particles, a speedup of 50 was attained.

For comparison, the execution time of one solver iteration of the CPU Gauss-Seidel solver is graphed together with the execution time of one Jacobi iteration for the GPU implementation in Figures 8.9 and 8.10. In this case, the speedup was even larger and approaches 100 for the systems containing 100 thousand particles or more.

## 8.2 Memory usage

Figure 8.11 shows how the memory consumption increases for the performed tests. Clearly the Jacobian matrix requires the most amount of space at almost 650MiB for one million particles. Second largest is color data at about 45MiB and after that no data buffer is much larger than 15MiB.

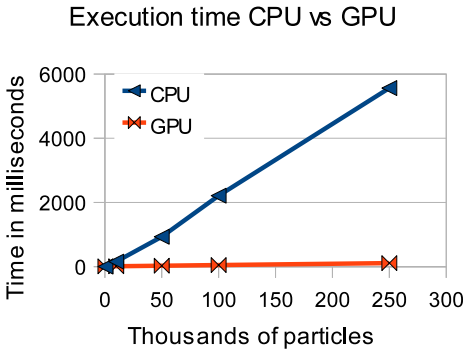


Figure 8.7: *Simulation time for the CPU and the GPU implementations. The GPU version is significantly faster.*

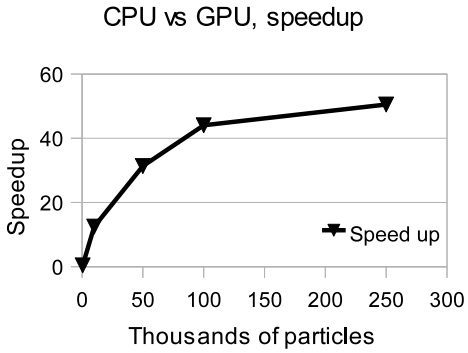


Figure 8.8: *The speedup achieved for the CPU comparison tests.*

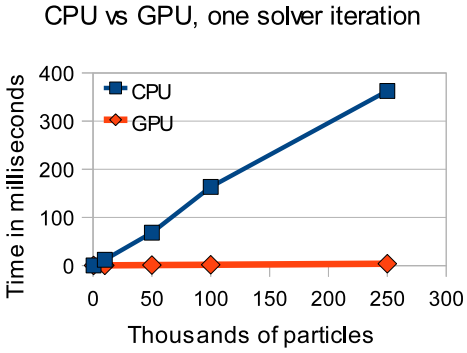


Figure 8.9: *Execution time for one solver iteration of the CPU and the GPU implementations.*

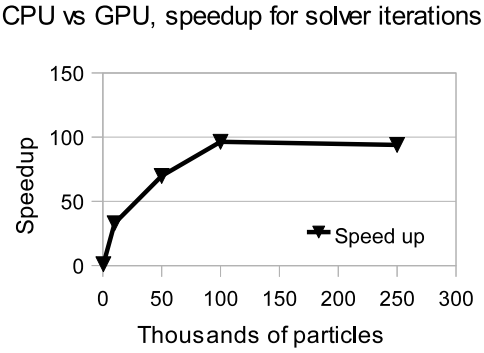


Figure 8.10: *Speedup for the solver iterations.*

## 8.3 Solver comparison

This section compares the performance of the different solvers that has been implemented, and also makes a qualitative comparison between them to motivate why Jacobi was chosen as the primary solver.

### 8.3.1 Performance

Figure 8.12 shows the cost, in milliseconds, to run the solvers on the one million particles performance test with an increasing number of iterations. While Jacobi is faster than the other two, the difference is not significant. For a given number of iterations, Jacobi requires about 20% less time to complete. To show the cost per iteration, Figure 8.13 shows the average time per iteration for the same scenario. The reason for the decrease in time per iteration is because the one-time cost of creating the right hand side is included, a cost that diminishes with increasing number of iterations. A profile of a solver iteration in a simulation involving one million particles is shown in Figure 8.14, where Jacobi and CG are compared. They both

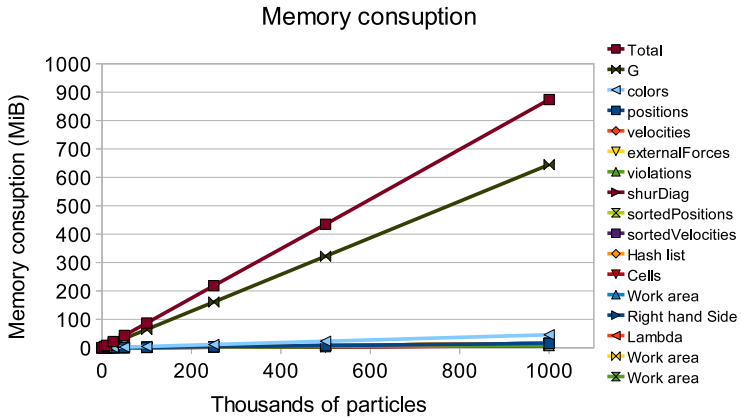


Figure 8.11: *Memory usage for the different buffers.*

contain one set of matrix-vector and transposed matrix-vector multiplication and it is clear that it is these two operations that dominate both solvers.

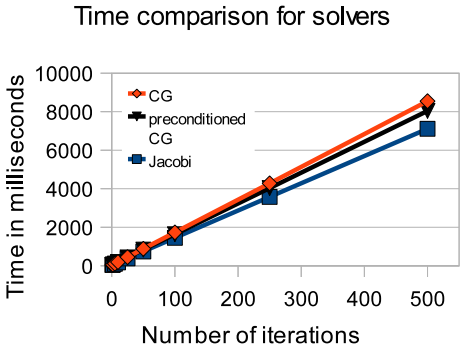


Figure 8.12: *Timings for an increasing number of solver iterations.*

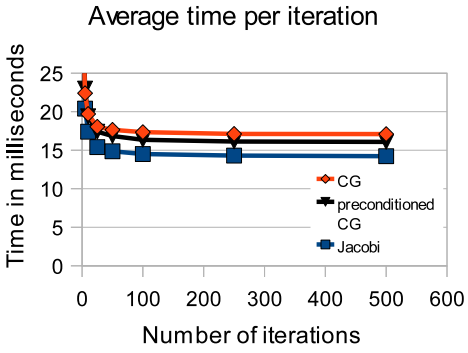


Figure 8.13: *Average time for each solver iteration. Timings include startup overhead, which produces the decreasing graphs shown.*

### 8.3.2 Convergence

The first solver to be implemented was Jacobi, since it is the simplest solver and the one easiest to map to the highly parallel CUDA platform. However, it proved more difficult than expected to get stable simulations and one of the aspects that was inspected in order to find the cause of the instabilities was the solver. A set of Matlab programs had already been written to verify that the CUDA Jacobi solver produced the same results as a Jacobi solver that had been implemented in Matlab and these Matlab programs were extended to include other solvers as well. Two solver methods were chosen for the comparison. The first was Gauss-Seidel, since successful constraint fluid simulations has used this solver. The second was Conjugate Gradient, CG, since it has been implemented for the CUDA platform by others, as mentioned in

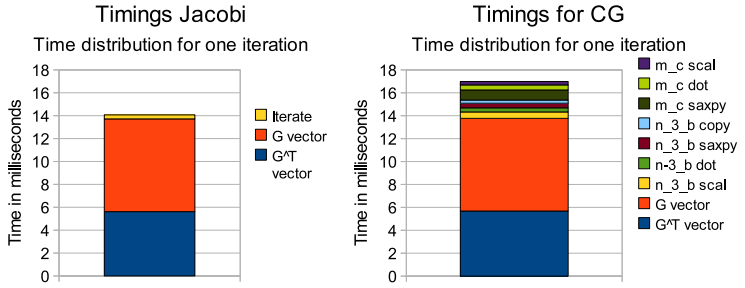


Figure 8.14: *Profile for one solver iteration of Jacobi and CG. The matrix operations are the most costly operations.*

Section 2.5. The actual Matlab implementations used were found in the Scientific/Educational Matlab Database [13].

The errors discussed in the following text and shown in the figures below have been created by importing all data generated from the application into Matlab and there calculated as

$$||S_\varepsilon \lambda - \mathbf{b}||_2 \tag{8.1}$$

This expression is evaluated once for each solver and iteration.

Prior testing had shown that fluid instabilities always occurred in areas of high density and the comparison with the Matlab solvers were therefore done using a small scene with a successively denser fluid. The result from the first run is shown in Figure 8.15. The setup for this test was a normal fluid with the highest density of any particle being 3% too high. The figure show that Jacobi and Gauss-Seidel perform very similar and that CG, after some fluctuations, produces a very accurate result. As the density of the fluid is increased, Jacobi produces less and less accurate results, as shown in Figure 8.16 where the average fluid density was 188% too high. All three solvers converged, but Jacobi converged slower than the others. This is illustrated more clearly in Figure 8.17 and 8.18 where the errors for Jacobi and Gauss-Seidel are shown with a shorter span for the vertical axis. The difference grows as the number of iterations increases, as seen in Figure 8.19. When the fluid is further compressed, as in Figure 8.20, Jacobi starts to fail. Both Gauss-Seidel and CG can produce the desired result, but the Jacobi solver bounces between two very inaccurate solutions. As the density is increased a little more, to 217% shown in Figure 8.21, Jacobi starts to diverge.

Unfortunately, the fast convergence of CG is not maintained when the number of particles is increased and running any decently sized simulation using the CG solver is painstakingly slow, requiring several hundreds, up to thousands, of iterations for a thousand particles. A preconditioned version of CG has been implemented to speed up the convergence, but lingering bugs makes it unusable at this point.

The reason for the instabilities in a normal fluid simulation using the Jacobi solver appears to be caused by the degrading accuracy of the Lagrange multipliers $\lambda$. As the fluid begins to compress against the floor due to gravity, Jacobi starts to produce inaccurate solutions which fail to properly counteract the compression, resulting in even higher densities in the next time step and thus even less accurate constraint forces. As the density increases, $S_\varepsilon$ begins to loose its diagonal dominance which eventually leads to divergence.

In addition, the low number of solver iterations limits the rate of pressure propagation in the fluid as each Jacobi solver iteration only propagates corrections to the solution a short distance in the system. Consider a scenario where a rectangular block of fluid is falling towards the floor.

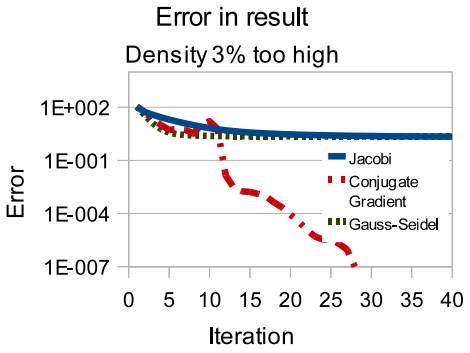


Figure 8.15: *Rate of convergence for a nice fluid. Jacobi and Gauss-Seidel produces similar results, but CG is considerably better.*

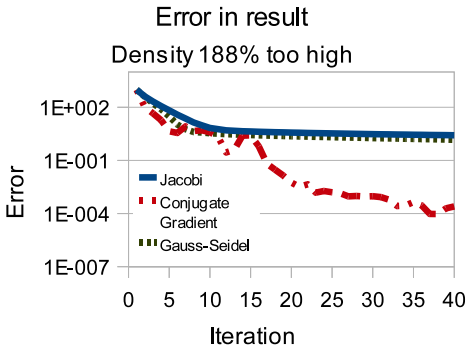


Figure 8.16: *Solver convergence for a compressed fluid. Jacobi produces less accurate results compared to Gauss-Seidel in this scenario.*

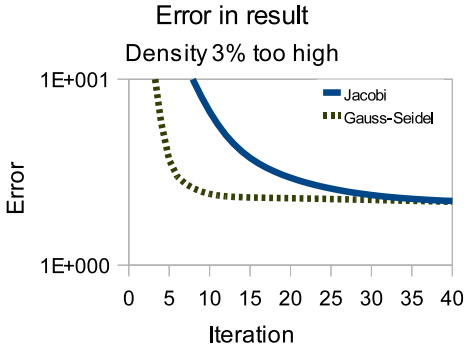


Figure 8.17: *Comparison between Jacobi and Gauss-Seidel for a nice fluid. They both reach a similar result, but Gauss-Seidel reaches it with fewer iterations.*

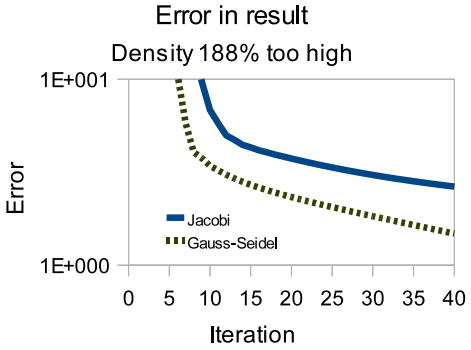


Figure 8.18: *Comparison between Jacobi and Gauss-Seidel for a compressed fluid. Jacobi does not manage to achieve the accuracy Gauss-Seidel gives.*

At the time step where the lowest layer of particles hits the floor, the Jacobi solver propagates the effect of the collision two particle layers up for each iteration. If the number of performed iterations is too low, none of the topmost particles become aware of the event and continue falling undisturbed. This further increases the density and leads to a spongy fluid.

The number of Jacobi iterations required for a stable simulation depends on a number of properties of the simulated fluid. As hinted above, a thick layer of particles requires more iterations to propagate events though the whole fluid, and high relative velocities between particles also increases the need for more iterations, since a higher velocity gives deeper penetrations when two particles meet and consequently higher densities. Both of these problems can be countered by reducing the length of the time step. Shorter time steps let the simulator restore violated constraints earlier, and any constraint violation will be smaller since particles can move shorter distances between time steps. Also, the regularization parameter $\varepsilon$ can be increased to increase stability in a fluid. The simulator becomes less forceful in restoring constraint violations when $\varepsilon$ has been increased, giving a system matrix that is more well-conditioned. The draw-

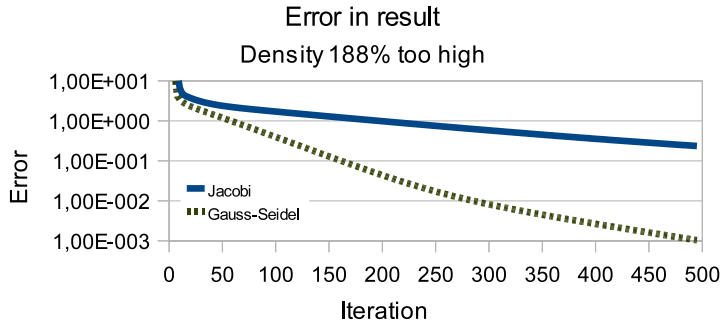


Figure 8.19: *The difference between Jacobi and Gauss-Seidel increases as more iterations are performed.*

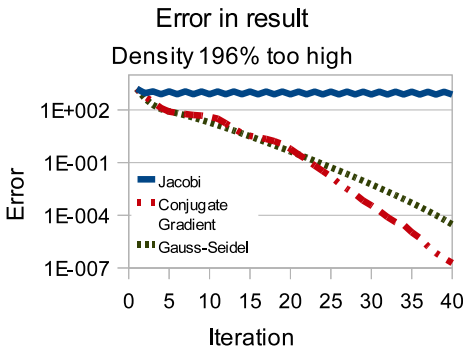


Figure 8.20: *Rate of convergence for a highly compressed fluid. While both Gauss-Seidel and CG converges, Jacobi is oscillating between two very inaccurate solutions.*

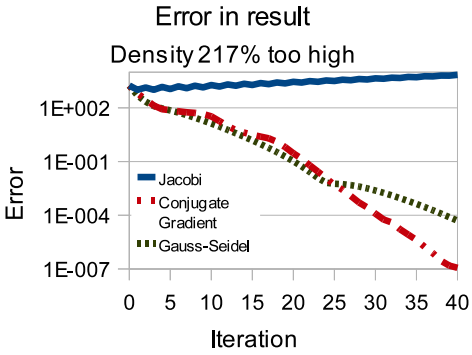


Figure 8.21: *When the fluid is compressed a little further, Jacobi starts do diverge.*

back is that the fluid becomes more compressible. Using the configuration given in Table 8.1, 100,000 particles can be simulated with reasonable results and a million particles simulated if high velocities and high piles are avoided. A reduction of the number of iterations to ten will still be able to simulate a million particles, but the density can locally increase significantly and the fluid will never come to rest because of the excessive spongyness. Five iterations will produce spongy and unstable simulations even for 100,000 particles and two iterations fails even for 10,000 particles. One iteration can barely run 800 particles in a stable simulation.

## 8.4 Demonstrator

The problem statement outlined in the beginning of this report, Section 2.1, states that the demonstrator should be able to visualize the fluid as it is being simulated and also be able to produce a dam break scenario.

The finished demonstrator can visualize the fluid in real time and also produce the dam break scenario described in the problem statement. In addition, the demonstrator enables the user to

give specifications on the fluid and the world in which it is simulated. The user can supply startup parameters to the demonstrator, through which, for example, the number of particles, the number of solver iterations, the size of the container, and the initial position of the dam wall can be specified. During runtime, the user can have a limited interaction with the fluid by changing the gravity vector and initiating the dam break. The dam wall can be moved in three ways. The first is the actual dam break, where the dam wall is moved from its current position to the far edge of the cell grid. Any particles that were stacked up behind the wall will begin to flow towards the new wall position. The other two wall moving operations initiate a constant movement of the dam wall in either direction, either squeezing the fluid together or letting it flow outwards. It is not recommended to squeeze the fluid too tight, since that will bring instabilities to the fluid, or to initiate a full dam break when the dam wall is extended outside the cell grid, since that may place particles far into the wall. Deep penetrations create large penetration forces, which in turn creates large velocities and possibly instabilities to the fluid.

Other features include continuous or single time stepping of the simulation, camera controls, display of the current frame rate and some state inspections of the fluid. Data that can be inspected are the current particle positions, density constraint violations and the distribution of the number of neighbors each particle has. Also, the demonstrator can print the memory usage of the fluid, detailing how much memory is used for each part of the library.

A screen capture of a dam break with 100,000 particles is shown in Figures 8.22 to 8.27.

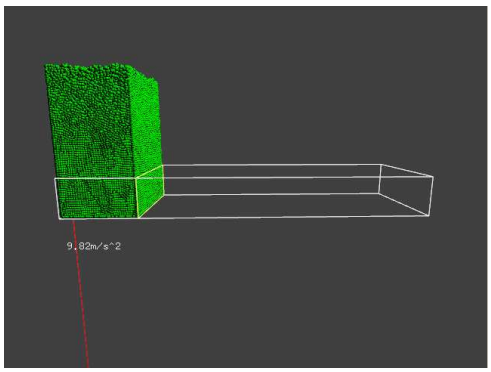


Figure 8.22: *A wall of fluid held back by a dam.*

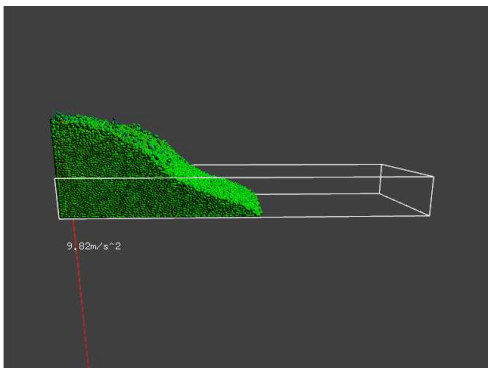


Figure 8.23: *The dam has been broken and fluid is flowing out into the container.*

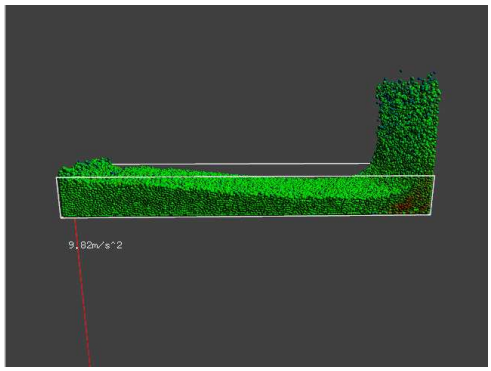


Figure 8.24: *The fluid hits the far end of the container and is being pressed upwards.*

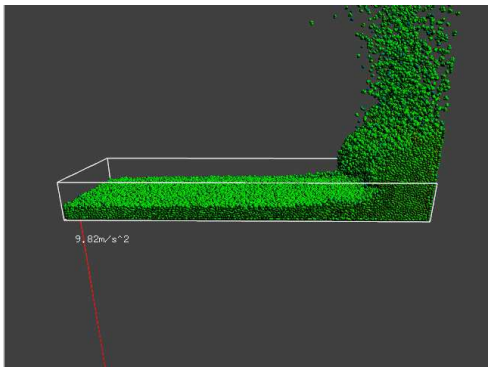


Figure 8.25: *Due to gravity, the fluid comes falling down again.*

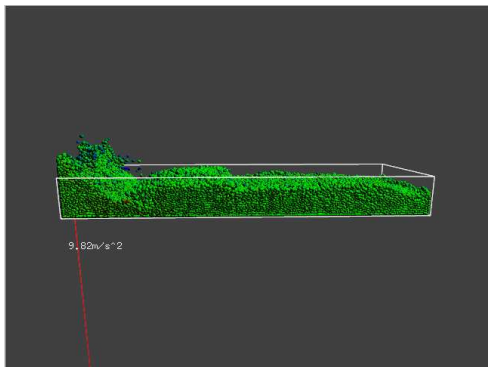


Figure 8.26: *The fluid is beginning to calm down.*

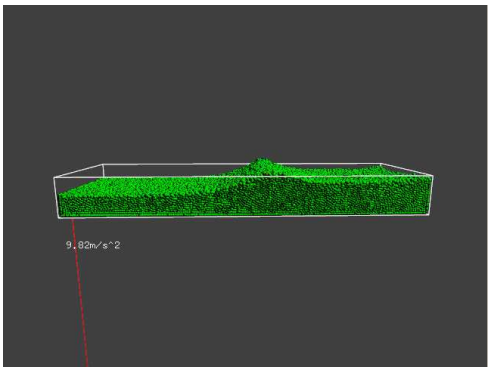


Figure 8.27: *The final wave.*

# Chapter 9

# Conclusions

The purpose of this maters's thesis was to evaluate the performance benefits possible by implementing the constraint fluid method for the highly parallel graphics hardware. The goal was to simulate hundreds of thousands of particles at interactive rates, which is currently not possible on conventional processors. This report has shown that all steps of the simulation can be performed in a parallel fashion and that the entire simulation fits well with the architecture of the target platform. The performance goal was reached since a hundred thousand particles can be simulated in twenty frames per second, and a speedup of fifty was achieved compared to the existing CPU implementation.

An important discovery that was made during the course of the project is that the Jacobi method is not well suited to solve the systems of equations that are generated during the simulations. Jacobi diverges during certain conditions and thus causes instable simulations, in particular in areas of high density. The CPU implementation, which uses Gauss-Seidel instead of Jacobi, does not have these problems.

## 9.1 Limitations

While the developed software can simulate the systems described in the problem description, the lack of a working preconditioning for the CG solver is a serious limitation that leaves Jacobi as the only practical solver. The significant part of the total simulation time that is taken by the solver makes it a candidate for optimization efforts and the major time consumer in all solvers is the matrix-vector operations. Any speedup achieved for these operations translate to an almost equal speedup for the entire application when many iterations are performed. Also, the memory requirements are quite large and imposes a hard limit on the maximum number of particles that can be simulated. On the GTX 280 card used during the development, the maximum number of particles is about 1,128,000, which requires 994MiB of memory on the device.

In summary, to improve the performance of the application, one area to investigate further is the implementation of the sparse matrix and its operations and the problems with the preconditioning of the CG solver since an efficient preconditioning can reduce the number of required solver iterations significantly.

## 9.2 Future work

Apart from the aforementioned optimization possibilities, there are many more changes that can be done to the library that will increase its usefulness.

The current implementation is the most basic simulation possible, the only simulated entity is the fluid and the only thing it can interact with is an axis aligned box. To make the simulation more useful, it is necessary to add interactions between the fluid and solids such as boxes, cylinders and triangle meshes. The non-penetration constraints could be used to make the fluid aware of the solids, but the current implementation would see the solid as a fixed wall and has no way to push on it. In other words, things can not float. An easier addition to the simulation that enables some interesting scenarios is to apply a height map to the container. Both the level and the normal of the floor can be found from the height map and creating the non-penetration violation from that will allow a designer to make non-flat surfaces that the fluid can flow on, modeling for example a river in a landscape.

All simulated particles are currently created and positioned at the beginning of the simulation and all of them are always updated each simulation step. This makes particle emitters impossible since there is no way to create new particles. Attempts have been made to run the simulation on a subset of the particles and create emitters by including more particles as the simulation progresses, but unfortunately, initial attempts failed and other aspects of the library were prioritized.

Regarding the lack of a mathematical foundation for the gradient function used in the simulator. While the choice of gradient function gives improved stability, it is not at all satisfactory that the choice of kernel function is so arbitrary. This is an important area for future research and, preferably, one would like to rely on a guiding physical principle for deriving or choosing optimal kernel functions.

# Chapter 10

# Acknowledgements

There are a number of people that have been essential to the progress of this project. First I would like to thank the supervisors of the project: Kenneth Bodin from Algoryx and Lars Karlsson at Umeå University. Kenneth has taught me about fluid simulations, while Lars has made sure I've understood everything and helped me to stay on course with the project. In the darkest moments, Claude Lacoursière stepped in with his vast knowledge and experience in theoretical matters and helped me understand and work around problems I've encountered, in particular with the Conjugate Gradient solver. Emil Ernerfeldth, who has previously implemented the constraint fluid method, has kindly answered all questions I've had and provided tips and suggestions for the actual implementation. I would also like to thank Nils Hjelte, who went through the trouble of creating the CPU timings used to compare the performance of my library with the implementation already available at Algoryx. In addition, all employees at Algoryx have shown their support in both technical, social, and instructive ways, helping me feel welcome every day.

# Chapter 11

# Terminology

**ALU - Arithmetic Logic Unit**
Hardware that performs numerical calculations on integer numbers.
**AMD - Advanced Micro Devices**
A semiconductor company that develops computer processors and related technologies.
**ATI - Array Technologies Incorporated**
A company developing graphics processing units. In 2006, ATI merged with AMD.
**Cache**
Very fast, but small, memory where frequently used data are stored temporarily to reduce the number of accesses to the slower memories lower in the memory hierarchy.
**CG - Conjugate Gradient**
An iterative method used to solve systems of linear equations.
**Constraint surface**
The set of object positions where a constraint is satisfied.
**CPU - Central Processing Unit**
Unit in a computer that fetches and executes instructions.
**CTM - CloseTo Metal**
Hardware interface released by AMD in 2006.
**CUDA - Compute Unified Device Architecture**
A development platform developed by NVIDIA that gives programmers direct access to the computational hardware of CUDA-enabled devices through a small extension to the C programming language.
**Device**
A CUDA-enabled hardware that executes kernel code.
**FPU - Floating Point Unit**
Hardware that performs numerical calculations on floating point numbers.
**DRAM - Device Random Access Memory**
Memory physically located on a device.
**GeForce**
A family of graphics cards by NVIDIA designed for the entertainment market, most notably video games.
**GPU - Graphics Processing Unit**
Hardware that specializes in graphics-related operations such as geometry rasterization, transformations, and texture filtering.
**GPGPU - General-Purpose Computations on Graphics Processing Units**

The use of graphics hardware for non-graphics computations.

**Host**

The CPU that launches kernels and performs memory copies is the host in a CUDA system.

**Indicator function**

A function that increases as a constraint violation is increased and is zero on the constraint surface.

**Kernel**

Code in a CUDA program that is executed on a CUDA device.

**Quadro**

Family of graphics hardware targeted for Computer Aided Design (CAD) and digital content creation developed by NVIDIA.

**SIMT - Single Instruction, Multiple Threads**

The architectural design of the Streaming Multiprocessors. Each SM can only issue one instruction at a time, but each SM harbors several threads that are free to follow independent execution paths.

**SM - Streaming Multiprocessor**

A hardware unit on a CUDA device that executes the threads of a thread block.

**SP - Scalar Processor**

The cores of a Streaming Multiprocessor that perform scalar arithmetic.

**Stream**

An ordered list of homogeneous data elemets.

**Tesla**

A family of hardware developed by NVIDIA based on the graphics cards but optimized for general-purpose computing.

**Warm starting**

Using the solution from the previous time step as the initial guess for the current time step when using an iterative solver.

**Warp**

A set of CUDA threads that physically executes in parallel.

# References


[1] David L. Alexander. Can you compress a liquid (water)? Webpage. Available at http://www.physlink.com/Education/AskExperts/ae15.cfm?CFID=18054225\&CFTOKEN=89305169, visited 2009-06-12.

[2] AMD. AMD "Close to Metal" Technology Unleashes the Power of Stream Computing . Webpage, 2006. Available at http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543_13744~114147,00.html, visited 2009-06-12.

[3] Owe Axelsson. *Iterative Solution Methods*. Cambridge University Press, 1994.

[4] Blaise Barney. Introduction to parallel computing. Webpage, Lawrence Livermore National Laboratory. Available at https://computing.llnl.gov/tutorials/parallel_comp/, visited 2008-06-15.

[5] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975. Available at www.cs.cmu.edu/~christos/courses/826-resources/PAPERS+BOOK/p509-bentley.pdf, visited 2009-06-18.

[6] Kenneth Bodin, Claude Lacoursière, and Martin Servin. Constraint fluids. *Internal report, submitted for publication*, 2008.

[7] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM transactions on graphics*, 2003. Available at www.multires.caltech.edu/pubs/GPUSim.pdf, visited 2009-06-17.

[8] Jeremiah U. Brackbill, Douglas B. Kothe, and Hans M. Ruppel. Flip: A low-dissipation, particle-in-cell method for fluid flow. *Computer Physics Communications*, 48:25–38, January 1988.

[9] Luc Buatois, Guillaume Caumon, and Bruno Lévy. Concurrent number cruncher - A GPU implementation of a general sparse linear solver. *International Journal of Parallel, Emergent and Distributed Systems*, 24, 2009. Available at http://alice.loria.fr/index.php/publications.html?Paper=CNC@2008, visited 2009-02-15.

[10] Matthias Christen. GPGPU: General purpose computing on graphics processing units. Available at http://fgb.informatik.unibas.ch/people/christen_matthias/lectures/resources/PPT_SS08_CS311_GPGPU.pdf, visited 2009-03-05, 2008.

[11] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav K. Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. In *Proceedings of ACM Interactive 3D Graphics Conference*, pages 189–196, 1995. Available at http://www.cs.jhu.edu/~cohen/Publications/icollide.pdf, visited 2009-06-18.

[12] Keenan Crane, Ignacio Llamas, and Sarah Tariq. *GPU Gems 3*, chapter 30 - Real-Time Simulation and Rendering of 3D Fluids. Addison-Wesley Professional, 2007. Available at http://http.developer.nvidia.com/GPUGems3/gpugems3_ch30.html, visited 2009-06-17.

[13] Scientific/Educational Matlab Database. Iterative solvers. Available at http://matlabdb.mathematik.uni-stuttgart.de/files.jsp?MC_ID=3&SC_ID=5, visited 2009-07-20.

[14] David Kirk and Wen-mei Hwu. CUDA Textbook. Draft. Available at http://sites.google.com/site/cudaiap2009/materials-1/cuda-textbook, visited 2009-04-16, 2008.

[15] James W. Demmel. *Applied numerical algebra*. SIAM, Philadelphia, 1997.

[16] Marios D. Dikaiakos and Joachim Stadel. The k-d tree structure. Webpage. University of Washington, Available at http://hpcc.astro.washington.edu/faculty/marios/papers/perform/node3.html\#SECTION00021000000000000000, visited 2009-06-18.

[17] J. Dongarra. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, chapter Sparse Matrix Storage Formats. SIAM, Philadelphia, 2000. Available at http://www.cs.utk.edu/~dongarra/etemplates/node372.html, visited 2009-04-30.

[18] Susan Eggers, Joe Emer, Henry Levy, Jack Lo, Rebecca Stamm, and Dean Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Computer special: How to use a billion transistors*, 1997. Available at http://www.cs.washington.edu/research/smt, visited 2009-06-15.

[19] Kenny Erleben, Jon Sporring, Knud Henriksen, and Henrik Dohlmann. *Physics-Based Animation*. Charles River Media, Inc., 2005.

[20] Jeff Freeman. Kaboom: Multi-threaded fluid simulation for games. Webpage, 2009. Available at http://software.intel.com/en-us/blogs/2009/02/05/kaboom-multi-threaded-fluid-simulation-for-games/, visited 2009-06-27.

[21] R. A. Gingold and J. J. Monaghan. Smoothed Particle Hydrodynamics - Theory and Application to non-spherical stars. *Royal Astronomical Society, Monthly Notices*, 181:375–389, November 1977. Available at http://articles.adsabs.harvard.edu//full/1977MNRAS.181..375G/0000375.000.html, visited 2009-06-10.

[22] Scott Le Grand. *GPU Gems 3*, chapter 32 - Broad-Phase Collision Detection with CUDA. Addison-Wesley Professional, 2007. Available at http://http.developer.nvidia.com/GPUGems3/gpugems3_ch32.html, visited 2009-02-24.

[23] Dominic Göddeke. Languages and programming environments. Presentation Slides. Available at http://www.mathematik.uni-dortmund.de/~goeddeke/arcs2008/outline.html, visited 2009-03-02.

[24] Takahiro Harada. Takahiro harada. Webpage. Available at http://www.iii.u-tokyo.ac.jp/~takahiroharada/, visited 2009-06-17.

[25] Takahiro Harada, Seiichi Koshizuka, and Yoichiro Kawaguchi. Smoothed Particle Hydrodynamics in Complex Shapes. *Spring Conference on Computer Graphics*, 2007. Available at http://www.iii.u-tokyo.ac.jp/~takahiroharada/PDF/2007_SCCG.pdf, visited 2009-06-16.

[26] Takahiro Harada, Seiichi Koshizuka, and Yoichiro Kawaguchi. Smoothed Particle Hydrodynamics on GPUs. *Computer Graphics International*, 2007. Available at http://www.inf.ufrgs.br/cgi2007/cd_cgi/papers/harada.pdf, visited 2009-06-17.

[27] Mark Harris. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 31 - Mapping Computational Concepts to GPUs. Addison-Wesley Professiona, 2005. Available at http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter31.html, visited 2009-02-24.

[28] Mark Harris. Optimizing CUDA. Available at http://www.gpgpu.org/sc2007/SC07_CUDA_5_Optimization_Harris.pdf, Accessed2009-02-20, visited 2009-03-15, 2007.

[29] Woosuck Hong, Donald H. House, and John Keyser. Adaptive particles for incompressible fluid simulation. Technical report, Texas A&M University, 2007. Available at www.cs.tamu.edu/academics/tr/tamu-cs-tr-2007-7-2, visited 2009-06-17.

[30] Philip M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, 1995.

[31] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *Computer*, 36(8), 2003. Available at www.cs.rice.edu/~rixner/kapasi_computer.pdf, visited 2009-08-12.

[32] Robert Klima. Numerics. Webpage, 2003. Available at http://www.iue.tuwien.ac.at/phd/klima/node18.html, visited 2009-07-06.

[33] James T. Klosowski, Martin Held, Joseph S.B. Mitchell, Henry Sowizral, and Karel Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4:21–36, 1996.

[34] Claude Lacoursière. *Ghosts and machines : regularized variational methods for interactive simulations of multibodies with dry frictional contacts*. PhD thesis, Umeå University, Computing Science, 2007.

[35] Hsien-Hsin Sean Lee. Multicore and GPU Programming for Video Games. Georgia Institute of Technology. Available at users.ece.gatech.edu/~lanterma/mpg08/mpglecture12f08.pdf, visited 2009-02-24.

[36] G.R. Liu and M. B. Liu. *Smoothed Particle Hydrodynamics: A Mesh Free Method*. World Scientific Publishing Co. Pte. Ltd., 2003.

[37] Gui-Rong Liu. *Mesh Free Methods: Moving Beyond the Finite Element Method*. CRC, first edition, July 2002.

[38] Aki Miettinen and Arne Pajunen. Concurrent and parallel computing - general purpose computing on graphics processors. Available at www.it.lut.fi/kurssit/08-09/CT30A7001/Seminars/group01document.pdf, visited 2009-03-10.

[39] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of ACM SIGGRAPH Symposium on Computer Animation (SCA)*, pages 154–159, 2003. Available at http://www.matthiasmueller.info/publications/sca03.pdf, visited 2009-06-12.

[40] NVIDIA. Geforce 8800. Webpage. Available at http://www.nvidia.com/page/geforce_8800.html, visited 2009-04-07.

[41] NVIDIA. Geforce gtx 285. Webpage. Available at http://www.nvidia.com/object/product_geforce_gtx_285_us.html, visited 2009-04-07.

[42] NVIDIA. Tesla. Webpage. Available at http://www.nvidia.com/object/tesla_computing_solutions.html, visited 2009-04-03.

[43] NVIDIA. What is CUDA. Webpage. Available at http://www.nvidia.com/object/cuda_what_is.html, visited 2009-01-28.

[44] John Owens. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 29 - Streaming Architectures and Technology Trends. Addison-Wesley Professiona, 2005. Available at http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter29.html, visited 2009-02-16.

[45] Per-Olof Persson and Sreenivasa R. Voleti. Applied Parallel Computing Solving Very Large Finite Element Problems in Parallel. Technical report, , 2002. Available at http://beowulf.lcs.mit.edu/18.337-2002/projects-2002/persson/18.337/fem/fem.html, visited 2009-08-13.

[46] Mahesh Prakash. Fluid solver: fluid simulation for motion picture special effects. Webpage, 2009. Available at http://www.csiro.gov.au/products/Fluid-solver.html, visited 2009-06-17.

[47] Chris Seitz. GPGPU: Is a Supercomputer Hiding in Your PC? Webpage, 2005. Available at http://www.informit.com/articles/article.aspx?p=398882, visited 2009-02-24.

[48] Jonathan R Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Pittsburgh, PA, USA, 1994. Available at www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf, visited 2009-06-29.

[49] Rizwan A Siddiqui, Isil Celasun, and Ulug Bayazit. Octree based compression of volumetric and surface 3d point cloud data. In *Virtual Systems and Multimedia*, 2007. Available at australia.vsmm.org/papers/siddiqui.pdf, visited 2009-06-18.

[50] Next Limit Technologies. Computational fluid dynamics. Webpage, 2009. Available at http://www.nextlimit.com/techno_cfd.php, visited 2009-06-17.

[51] W.A. Wiggers, V. Bakker, A.B.J. Kokkeler, and G.J.M. Smit. Implementing the conjugate gradient algorithm on multi-core systems. In *International Symposium on System-On-Chip*, 2007. Available at eprints.eemcs.utwente.nl/11441/01/Final_paper_soc2007.pdf, visited 2009-06-17.

[52] Adrew Witkin. Physically based modeing: Principles and practice - constrained dynamics. Robotics Institute, Carnegie Mellon University, 2007. Available at www.cs.cmu.edu/~baraff/sigcourse/notesf.pdf, visited 2009-03-24.