UMEÅ UNIVERSITY

REAL TIME SPHERICAL DISCRETIZATION

Surface rendering and upscaling

Philip Asplund

Master Thesis, 30 credits
Supervisor: Patrik Eklund, peklund@cs.umu.se
External supervisor: Niklas Melin, niklas.melin@algoryx.se
Master of Science Programme in Computing Science and Engineering

2020

Abstract

This thesis explores a method for upscaling and increasing the visual fidelity of coarse soil simulation. This is done through the use of a High Resolution (HR)-based method that guides fine-scale particles which are then rendered using either surface rendering or rendering with particle meshes. This thesis also explores the idea of omitting direct calculation of the internal and external forces, and instead only use the velocity voxel grid generated from the coarse simulation. This is done to determine if the method can still reproduce natural soil movements of the fine-scale particles when simulating and rendering under real-time constraints.

The result shows that this method increases the visual fidelity of the rendering without a significant impact on the overall simulation run-time performance, while the fine-scale particles still produce movements that are perceived as natural. It also shows that the use of surface rendering does not need as high fine-scale particle resolution for the same perceived visual soil fidelity as when rendering with particle mesh.

Acknowledgments

I would first like to thank the people at Algoryx for their help and feedback when working on this thesis. In name, I would like to personally thank Niklas Melin, my advisor at Algoryx, for his assistance on this thesis project.

I would also like to thank my advisor Patrik Eklund for his cooperation and guidance through the thesis project.

Abbreviation list

HR
High Resolution

FPS
Frames per second

DT
Time step

EMA
Exponential moving average

SSM
Screen space mesh

ms
milliseconds

PIC
Particle-in-Cell

FLIP
Fluid-implicit-particle

List of Equations

List of Figures

List of Tables

Contents

1 Introduction

Physics based real-time simulations are used for a multitude of reasons, where for example one of these are training simulators. Training simulators are an important part of the industry because they alleviate the strain on training new operators and training for new vehicles. An important factor for training is immersion, which can help operators focus on the training part. Good rendering is an important factor for helping to increase the immersion factor. Soil simulation follows in that it can for example be used for the training of excavator use.

Thus if there are visual artifacts in the rendering then the immersion can be broken. This is why this thesis explores how to increase the visual fidelity of soil simulation, specifically one which is simulated using coarse spherical bodies.

The introduction also provides the structure of the thesis, background of the cooperation, a detailed problem formulation, and a research question.

1.0.1 Outline

The outline of the thesis:

- **Chapter 2: Theory** explains theory for understanding this thesis and this chapter also explores previous work that is related to this thesis.
- **Chapter 3: Method** describes the implementation of the chosen method.
- **Chapter 4: Results** presents the results of the implemented method both in terms of visual and performance, where performance also includes a time distribution of the algorithm.
- **Chapter 5: Discussions** discusses the results from Chapter 4.
- **Chapter 6: Conclusions** where conclusions regarding the method and the results are discussed along with future work.

1.0.2 Algoryx

Physics based real-time simulation of soil mechanics has been made possible through efficient implementation of validated models and with the help of faster computers. Algoryx Simulation, Umeå, is a company specialized in developing tools for the training simulator industry, engineering companies, and heavy machine manufacturers. Recently Algoryx have developed a new earth moving module, capable of simulating realistic soil mechanics in real-time.

The project was done in cooperation with Algoryx Simulation AB. Algoryx Simulation AB develops and sells the product AGX Dynamics, which consists of libraries for simulating articulated and contacting multibody systems. The customers integrate these libraries into their products and develop simulators and other tools for an end-customer market. The simulators are used for operator training, virtual prototyping and for developing autonomous, intelligent machines, and more.

AGX Dynamics C++ SDK comes with a basic rendering pipeline and C#, and Python bindings.

Algoryx is also working together with companies such as EPIC Games (Unreal Engine) and Unity (Unity3D) on integrations into their products. Efficient and realistic rendering is essential for the overall experience of the product.

1.1 Problem formulation

Real-time rendering of soil comes with numerous challenges, where the main challenge is that the application has to be run in real-time which creates a limited time budget for rendering and simulation. This limited time budget creates a problem in creating a visually realistic and appealing soil rendering. If only simulation particles are used to render the soil a tremendous number of particles would have to be simulated to render soil in a visually realistic fashion, but this is not reasonable for real-time rendering and simulation because of the high computational expense in simulating these simulation particles. In this context, a simulation particle means a fully simulated particle in terms of internal and external forces. Instead coarse simulation particles (large particles) can be used for the simulation part but this then deviates from visual realism. Thus another approach is needed to create a more visually realistic dynamic soil rendering, other than just rendering these particles as is.

1.1.1 Research questions

This thesis implements a High Resolution (HR) Granular based approach with surface generation for rendering dynamic soil, given an underlying physical model. Thus the thesis explores if this approach can be used to create a visually compelling dynamic soil, which will be determined by visual inspection. Visual inspection in this case means that the visual fidelity is assessed by the author and by experts in physic based simulation at Algoryx. This is done in this way because measurable visual fidelity assessment is not readily available. In addition, the thesis also evaluates the performance impact of such a model on the overall simulation time, to evaluate if the HR Granular based approach with surface generation can be used in real-time simulation.

2 Theory

This chapter explores and covers important concepts and backgrounds used in this thesis. It also covers a deeper look into related works on the bigger concepts used for this thesis.

2.1 Computer graphics

This section covers and explains important concepts related to the field of computer graphics.

2.1.1 Real-time rendering

Rendering is the process of displaying and creating a 2D image on a screen of a 3D object or environment, by the use of shading, texturing, and more. Thus real-time rendering is the process of rapidly rendering theses images in concern with intractability. The measurement of real-time rendering is done by measuring either the number of frames/images that get rendered each second or by the delay between each frame/image. With intractability in mind, i.e. the fact that a user's inputs can change the scene by moving the camera or affecting the objects in the environment, one wants a delay as low as possible between each frame to get as smooth and interactive experience as possible. With this, there is no real defined point regarding to what counts as real-time rendering, but at a rate of around 6 frames per second (FPS) one starts to get a sense of interactivity with the image which grows until around 30-60 FPS where diminishing returns of intractability starts to come into play. Exactly when diminishing returns start depends on the use case [1].

2.1.2 Polygon mesh

A polygon mesh represents the shape of a 3D object by a collection of vertices, edges, and faces/primitives which is used in computer graphics to render objects. A face can be a multitude of constructions but the most common in computer graphics is that a face is a triangle, i.e. a triangle mesh. A vertex is a data structure that contains a point in space, generally a 3D space point, a normal, a texture coordinate, and more. The edge is thus the connection between vertices and the face, is a set of edges that in this case constructs the triangle.

2.1.3 Billboard

A billboard is an orientated rectangle texture mesh based on the camera view direction and position. This means that as the camera position and view direction change the billboard's orientation also changes to match. This matching is generally done either in all axes or only a subset of them. Billboards are a simple and common technique to render VFX effects such as fire, smoke, explosions, and more. This can be done by rendering the billboard which is attached to the position of a particle and textured with an alpha texture, see Figure 1, which shows a normal planar billboard but also a spherical billboard.

Figure 1: Planar and spherical billboard of fire and smoke VFX effect [2].

2.1.4 Spaces

In computer graphics and rendering there are generally four main different spaces. In order of transformation; *Object/Local space*, *World Space*, *Camera/View space*, and *Projection/Screen space*. Transformations between these spaces are done by transformation matrices called in order of transformation; *Model matrix*, *View matrix*, and *Projection matrix*. This transformation can also go the other way around by using the inverse of these transformation matrices. *Object/Local space* denotes the mesh/object internal coordinates meaning how the vertices that build up the mesh/object correlate to each other in space. *World Space* then correlates to the coordinates in the world, i.e. how objects correlate to each other in the world/environment. *Camera/View space* can be seen as world space transformed so that the objects are in front of the camera, i.e. how the object related to the camera. Thus *Projection/Screen space* is the view space but projected as a camera lens would, i.e. oblique or perspective projection, meaning we get viewed depth in the screen where the coordinates are defined as the pixels of the screen.

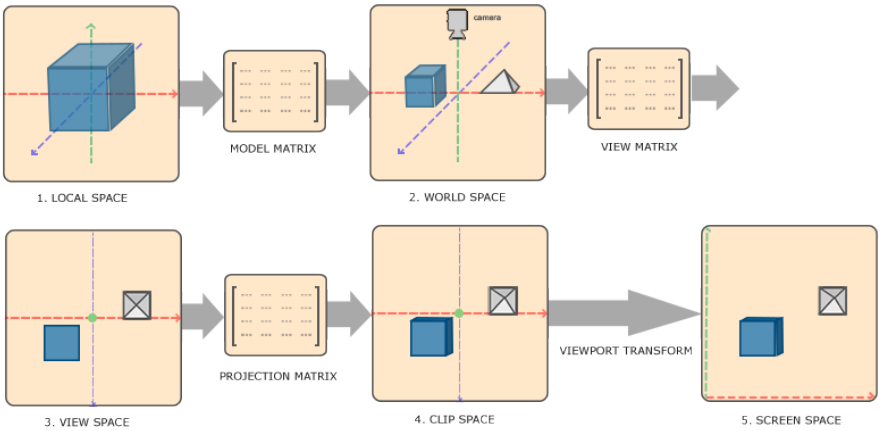


Figure 2: Different spaces and the corresponding transformation matrix [3].

2.1.5 Graphics pipeline

The general graphics pipeline can be seen in Figure 3 [1].

- **Vertex shader** is the process and program which works on vertices. The main purpose of the vertex shader is two-fold; calculating the position of the vertices and evaluating vertex data such as normals and texture coordinates.
- **Geometry shader** is the process and program for generating new vertices and primitives given another primitive.
- **Clipping** is the process of removing vertices that lie outside of the view volume. If the clipping process removes a subset of a triangle new vertices are created on the clipping edge.
- **Rasterization** is the process of transforming vertices into fragments. This is done by finding all the fragments that lie inside of the primitives. The vertex data, i.e. normals, and texture coordinates are interpolated to generated new values for each fragment.
- **Fragment shader** is the shader that works per each fragment/pixel, for shading each fragment; i.e. calculating shadows, ambient occlusions, specular light, diffuse light, and more.
- **Output buffer** is where the resulting data from the graphics pipeline process is held while waiting to be displayed or used in some other way.

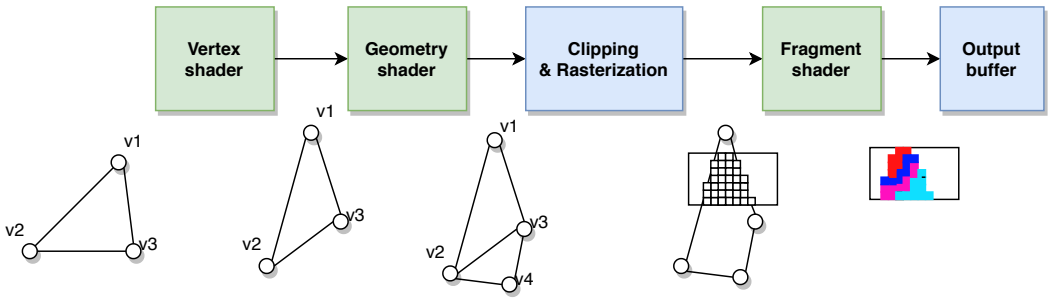


Figure 3: General graphics pipeline, where green stages can be fully modified by the implementation of shaders, but not the blue stages.

2.1.6 Compute shader

The compute shader is a more special shader than other shaders, and lives outside of the general graphics pipeline. The compute shader's specialty comes from that it is a general-purpose shader, meaning that they are used and meant to work for computing general computing tasks on the GPU instead of just working on triangles. Thus the introduction of compute shaders for the GPU means that one can do more complicated programs on the GPU highly efficiently if they are parallelizable. The compute shader works by setting up how many threads in each X, Y, Z dimension that shall be run for each group, where $X \cdot Y \cdot Z \leq 1024$ and $Z \leq 64$ in HLSL for compute shader version cs_5_0. The program is then run by a dispatch call where one defines how many of these threads groups in each dimension should be started.[4] In short if one wants to compute N process in the X dimension the number of dispatch in that direction follows Equation 2.1, where $\hat{X}$ is the resulting number of dispatch in X direction.

$$\hat{X} = \left\lceil \frac{N}{X} \right\rceil \tag{2.1}$$

2.1.7 Depth Maps

A depth map in computer graphics is an image/texture containing information about the distance from the camera to all objects in the camera view, with perspective in mind. Depth map is also interchangeable with z-buffer and depth buffer.

2.1.8 Normal calculation

When it comes to rendering one can generally dived normals into two types; vertex normals and face normals. Face normals are thus normals from faces. Face normals can be calculated from Equation 2.2 where p_0, p_1, p_2 are tree vertex positions of the triangle face and *normalize* is the function to divide the input value by its norm.

$$n_f = normalize((p_1 - p_0) \times (p_2 - p_0)) \tag{2.2}$$

Vertex normals are normalized vectors that come out from the vertex. Vertex normals can be calculated in several ways, with one way being to calculate the face normals of the faces that use the vertex and then averaging that value. This can be seen in Equation 2.3 where m is the number of faces connected and n_{fi} is the i face normal.

$$n_v = normalize\left(\frac{\sum_{i=1}^{m}(n_{fi})}{m}\right) \tag{2.3}$$

2.2 Physic simulations

This section covers and explains important concepts related to the field of physics and general simulations. This section also gives a brief explanation of the coarse soil simulation used.

2.2.1 Real-time simulation

Simulation is as the name states the process of simulating something, in this case, dynamic cut-able soil. Real-time simulation is thus a simulation that shall be done in real-time. In contrast to what was stated in Section 2.1.1, the definition of real-time rendering was fuzzy as to what counts as real-time. This is not the case for real-time simulation. Where it can be found from the time step (DT) of the simulation by following Equation 2.4, for example, a time step of $0.02s$ would imply an update frequency of 50 hertz. This hard definition can then be transferred to real-time rendering of real-time simulation by the fact that the rendering needs to be faster or as fast as the update frequency, meaning that in the example stated one would need at least 50 FPS.

$$frequency = \frac{1}{DT} \tag{2.4}$$

2.2.2 Coarse simulation

For the purpose of this thesis, the coarse simulation will mainly be treated as a black box. But there are two main points that need to be revealed and addressed about the coarse dynamic cut-able soil simulation. Firstly cut-able in this case means that dynamic soil particles spawn from a cutting force in a static soil heightmap which separates the dynamic soil from the static soil, meaning that dynamic soil particles are spawned. Secondly what dynamic means and refers to in this case, which is two-fold. Dynamic in that the soil particles can move freely in all axes, meaning that particles internal forces will be needed to be calculated such as collision and friction; external forces such as gravity; collision, and friction with external bodies. Dynamic in this case also means that when particles are spawned from a cutting force they spawn with a certain mass and radius that correlates to each other, but this mass and radius is not static and can decrease or increase in size until a max point which can be seen in Figure 4.

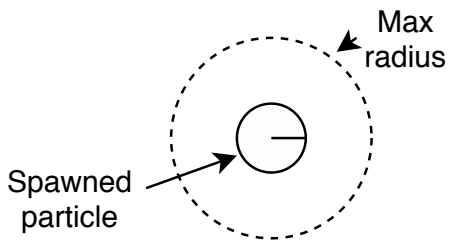


Figure 4: Coarse soil particle mass and radius expansion.

2.2.3 Voxel grid

A voxel grid is simply a regular grid in three dimensions that has a transformation matrix such that a world-coordinate can be translated to a voxel grid index. This means that one can divide the world into voxels, meaning that one can store data in these and handle the data and simulation in a voxel based view which is discrete instead of a continuous normal simulation. Figure 5 shows how a regular voxel grid works.

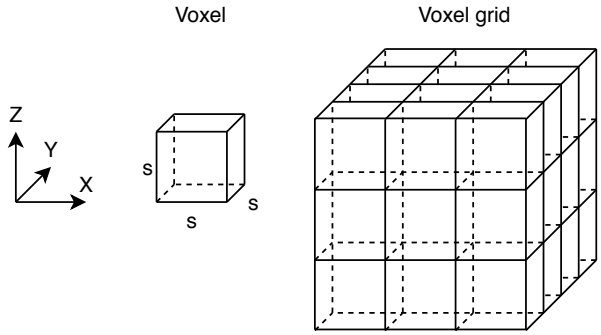


Figure 5: Voxel grid.

2.2.4 Velocity grid

The velocity grid is a voxel grid, whereas the name states the velocity grid stores information about the velocity in a specific voxel. This velocity represents the coarse soil particle velocity weighted and interpolated using Equation 2.5 where n is the number of influencing particles, v_i is the i particle's velocity, w_i is the weight of particle i which is based on the distance between the particle and the voxel center.

$$v_{voxel} = \frac{\sum_{i=1}^{n}(v_i \cdot w_i)}{\sum_{i=1}^{n}(w_i)} \tag{2.5}$$

2.2.5 Mass grid

The mass grid is also a voxel grid. The mass grid stores the information about the total mass in a specific voxel. The total mass comes from the particle mass following equation 2.6, where n is the number of influencing particles, m_i is the i particle's mass, w_i is the weight of particle i which is based on the distance between the particle and the voxel center. The mass of each active voxel is thus the range $(0.0, 1.0]$, where a mass of 1.0 indicates that the voxel is full.

$$m_{voxel} = \frac{\sum_{i=1}^{n}(m_i \cdot w_i)}{\sum_{i=1}^{n}(w_i)} \tag{2.6}$$

2.3 Filters

This section covers relevant filtering algorithms used within the context of this thesis project.

2.3.1 Exponential moving average

Exponential moving average (EMA) is a first-order infinite impulse response filter. A first-order infinite impulse response filter means that it is a recursive filter with resulting values influenced by the current value and previous input and output. EMA thus works by applying weighting factors to exponentially decrease the effect of older values to the filtered value. This results in that EMA smooths change of changing values over time. The EMA filter follows Equation 2.7; where Y is the observation, S is the EMA value and α is the smoothing factor $[0, 1]$, where 0 results in the EMA to be constant at the initial observation and 1 results in the EMA to be the current observation. Values of α in the range of $(0, 1)$ is thus the weight from the current observation into the current EMA.

$$S_1 = Y_1 \tag{2.7a}$$

$$S_t = \alpha \cdot Y_t + (1 - \alpha) \cdot S_{t-1} \tag{2.7b}$$

2.3.2 Gaussian filter

Gaussian filter is a filter for blurring values together by the use of a Gaussian function approximation. Gaussian filter is mainly used for blurring images. Equation 2.8 can be used to apply a Gaussian filter onto a 2D image, where $I_{filtered}$ is the filtered image, p is the 2D coordinates of the current filter point, Φ is the set of points around and including p, G is the Gaussian function defined in Equation 2.9. From Equation 2.9, σ is the standard deviation of the distribution.

$$I_{filtered}(p) = \frac{\sum_{p_i \in \Phi} I(p_i) G(p_i - p)}{\sum_{p_i \in \Phi} G(p_i - p)} \tag{2.8}$$

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.9}$$

2.3.3 Bilateral filter

Bilateral filter is also a filter for blurring values together, mainly for the use of blurring and smoothing images. But in contrast to a Gaussian filter, Bilateral filter is edge-preserving, meaning that the resulting filtered images will still keep edges after the blur. Equation 2.10 can be used to apply a Bilateral filter onto a 2D image, where B is the filtered image, I is the input image, p is the current filter point in the image, Φ is the set of points around and including p, f is a range kernel function generally Gaussian function, g is a spatial kernel function also general a Gaussian function.

$$B(p) = \frac{1}{W} \sum_{p_i \in \Phi} I(p_i) f(\|I(p_i) - I(p)\|) g(\|p_i - p\|) \tag{2.10a}$$

$$W = \sum_{p_i \in \Phi} f(\|I(p_i) - I(p)\|) g(\|p_i - p\|) \tag{2.10b}$$

2.4 Related work

This section covers and explores previously done work that is related to this thesis in the concepts of; *particle rendering*, *fine-scale particles*, and *surface generation*.

2.4.1 Particle rendering

To render particles in a 3D graphical program there generally exist two different approaches of rendering singular particles. The first and most simple in terms of computational cost and ascetics is to render these particles using billboards, this method was for example used by Haglund et al. in 2002 [5] to render particle based snowflakes. The other more computational expensive option is to render these by rendering a mesh of the particle, both spherical and non-spherical meshes depending on what the particle represents.

When rendering particles that represent a clump of matter a third main option also exists, which is to generate a surface of the particle cloud which is explored in section 2.4.3. This also then includes hybrids of using billboards and mesh rendering with surface generation.

2.4.2 Fine-scale particles

The process of decoupling the rendering data and process from the simulation data and process can be seen as started by the development of *PIC (Particle-in-Cell)*, used in water simulations by Harlow in 1957 [6] which was later improved into *FLIP (fluid-implicit-particle)* by Brackbill and Ruppel in 1985 [7]. A combination of *FLIP* and *PIC* was later used to animate and simulate sand as a fluid by Zhu and Bridson in 2005 [8]. Generally speaking, these methods maps the simulation onto a velocity grid which in turn is used to move the simulation particles, the simulation particle's velocity is then also interpolated into the velocity grid again. This means there is a start to decoupling of the simulation and render data but it is not entirely there yet, and these methods are generally not real-time.

Another approach is using a large scale or coarse particle simulation, and then doing a second fine-scale simulation that is guided by the coarse simulation, this was first done by Sony Pictures Imageworks in 2007 [9] as far as the author knows to render and animate sand. This was later developed into a more concrete simulation by Ivan et al. in 2009 [10], which uses as before a coarse particle simulation, and then interpolating these coarse particles to guide the internal forces of the fine-scale particles. This method was later modified by using the granular influence of outside forces and another kernel function for internal force interpolation by Ihmsen et al. in [11] instead of just using a cut off when there is only a small number of neighboring coarse particles. These methods, when they were implemented, were not used for real-time simulation and rendering, based on two points. Firstly the number of small particles is too large, and that external forces still influence the fine-scale particles.

2.4.3 Surface generation

In the field of surface generation, there are many algorithms for creating and generating a visual or geometric surface from a point/particle cloud. One of the most popular and well-known surface generation algorithm is *Marching Cubes* which was first presented by William E. Lorensen and Harvey E. Cline in 1987 [12]. *Marching Cubes* in short is an algorithm that works in world space and uses a particle cloud to generate a mesh by calculating triangles and edges based on these particles. An evolution of the *Marching Cubes* algorithm is an algorithm called *screen space mesh (SSM)* which was presented by Matthias Müller in 2007 [13], which as the name suggests, works in screen space to generate a surface mesh of the point cloud. Generally speaking, this is done by using a *Marching Cube* algorithm in screen space given a projection of the point cloud. The *SSM* method claims to get a performance increase over the traditional *Marching Cubes* algorithm. The SSM algorithm has for example been used in *Vortex* by *CMLabs* to generate a surface for their soil particle representation [14].

A more interesting approach to generate a visual surface representation of the particle cloud is *Particle Splatting*, by Bart Adams in 2006 [15]. Instead of generating an explicit surface mesh of the particle cloud, this approach blends the particles, resulting in a visual surface. According to the article, *Particle Splatting* does not suffer from the temporal discretization artifacts when particles move, which is a notable problem in the *Marching Cube* algorithm, when using a smaller grid resolution. Derived from this another interesting approach that also does not generate an explicit surface mesh of the surface, is *Screen Space Fluid Rendering with Curvature Flow*, by Laan and Green and Sainz in 2009 [16] when rendering a particles cloud surface. This was later presented as *Screen Space Fluid Rendering for Games*, by Green in 2010 [17]. This method was meant to be used as a basis for rendering fluids in games in a fast and computationally effective manner. The basis of this technique is to render the particles to a depth map, smooth the depth map, then use that depth map to render the particle surface based on the camera, i.e. in screen space.

3 Method

This chapter covers how the research question from Section 1.1.1 will be answered. This comes from showcasing the fine-scale simulation in the form of management of particles and the simulation/moving of fine-scale particles. It also includes the visual aspects in how the surface generation of the particle cloud is performed.

3.1 Fine-scale simulation

The fine-scale simulation can be split into two parts; managing particles, i.e. spawning and removing particles, and simulating/moving particles. The method of moving particles is based on the fact that the simulation ignores external and internal forces on the render particles and only calculates their movement from the coarse particles. This means that the fine-scale particle simulation can be highly parallelized as the particles do not affect each other, which suits simulation on the GPU. The management of the particles is thus not as suited for the GPU because it is not as easily parallelizable and the fact that data needs to be created and removed, thus this part is done on the CPU.

3.1.1 Manage fine-scale particles

1. Update mass/velocity grid, using EMA
2. Sort coarse particles into X, Y, Z columns
3. Get particle data from GPU
4. For each particle remove the corresponding mass on each mass voxel. If the mass value in the voxel is negative, despawn the particle.
5. For each voxel with mass left spawn an appropriate number of new fine particles in that voxel.
6. Send new particle buffer to the GPU.

Mass/velocity grid update

Each time step, an extra representation of the mass and velocity grid is updated using EMA as explained in Section 2.3.1. The reason to use EMA is to create a smoother transfer of the grid. This is needed for two main reasons. The first reason is to achieve smoother start and end spawning, in that particles gradually spawn instead of all appearing at the same time. The second reason is when the grid moves the use of EMA gives the fine particles smoother transfer between voxels.

Sorting coarse particles

All coarse particles get sorted each frame into columns using X, Y, and Z axes of the voxel grid, and in the columns themselves, the coarse particles are sorted by the position of the column axis. These columns are based on the voxel size.

Spawning particles

When particles are spawned they spawn in a randomly selected place inside their designated voxel, but the spawning zone inside of this voxel can be decreased in all axes. The decrease in an axis depends on the coarse sorting, by that a particles shall not spawn where there is no coarse particle influence, this can be seen in Figure 6. The reason for this restriction of the spawning zone in a voxel is to make a more natural like spawning of the particles and a less box-like spawning of particles. When particles are spawned they also get a random selected UV-coordinate to colorize the particle cloud by a terrain texture.

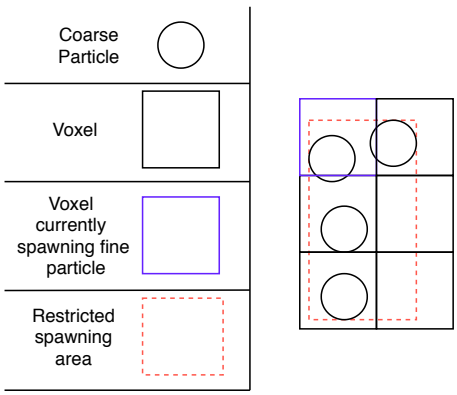


Figure 6: Particle restricted spawning 2D version.

Despawning particles

Particles get despawned when a particle is either inside of a voxel with zero mass or when Equation 3.1 holds. In Equation 3.1, Φ is the set of particles inside the voxel, M is a function to get mass of a particle, m_v is the mass of the voxel in question, and ϵ is an extra mass addition generally double the value of a single particle mass. Thus as long as this holds particles will be despawned from the voxel. The reason for ϵ is to create a buffer for despawning particles in order to reduce particle flickering.

$$m_v + \epsilon < \sum_{p \in \Phi} M(p) \tag{3.1}$$

3.1.2 Simulating fine-scale particles

1. Transform the velocity and mass grid to a buffer of active voxels.
2. Send buffers to the GPU.
3. Call dispatch following the rules from Section 2.1.6.

On the GPU, for-each particle, Equation 3.2 is used to calculate the velocity of the particle. From Equation 3.2, v_p is the resulting velocity of the current particle; Φ is the set of indices of voxels that surround the current particle, P gives world position given index, V gives the velocity of a voxel given index, M gives the mass of a voxel given index, s_{voxel} is the size of the voxel, and p_{pos} is the current particle's world position.

$$W(i) = max(0, 1 - \frac{distance(p_{pos}, P(i))^2}{s_{voxel}}) \tag{3.2a}$$

$$v_p = \frac{1}{\sum_{i \in \Phi}(W(i) \cdot M(i))} \cdot \sum_{i \in \Phi}(W(i)V(i)M(i)) \tag{3.2b}$$

To update a particle's position in time step i Equation 3.3 is used. From Equation 3.3, v_i is the calculated velocity of the particle on time step i, DT is the time step and p_i is the particle's position at time step i.

$$p_i = v_i \cdot DT + p_{i-1} \tag{3.3}$$

3.2 Surface rendering

The method chosen for generating a visual surface of the soil particle cloud, is a version based on screen space fluid [16, 17]. This method in contrast to SSM as an example is not particle dependent, excluding the depth map generation and it does not, as previously stated in Section 2.4.3, generate an actual surface mesh. This means that excluding the depth map generation, the number of particles in the simulation should not affect the performance of this algorithm. Instead, the performance factor comes from screen resolution and which blurring algorithm is chosen.

The surface rendering algorithm:

1. Generate depth map.
2. Blurring/Smoothing of the depth map.
3. Render full screen quad.
4. Calculate normals given the depth map.
5. Shading.

Depth map

To generate the depth map of the particle cloud, particles are rendered by the use of a billboard that is oriented toward the camera with depth replacement in the fragment shader, to give a spherical depth to the billboard. The depth replacement in the fragment shader is done by Equation 3.4. From Equation 3.4, t is the UV coordinated of this fragment, f_4 is a function to create a 4D vector, p_{view} is the fragment's position in view space, r is the radius of the particle, M_p is the projection matrix, and one also checks if $(n_{xy} \cdot n_{xy}) > 1.0$ then one discards that fragment because it lies outside of the sphere. In the process of generating the depth map, a UV map is also created with the randomized UV values from the particles set in the UV map.

$$n_{xy} = 2t - 1 \tag{3.4a}$$

$$n_z = \sqrt{1 - (n_{xy} \cdot n_{xy})} \tag{3.4b}$$

$$c_{xyzw} = f_4(p_{view} + n \cdot r, 1) \cdot M_p \tag{3.4c}$$

$$depth = \frac{c_z}{c_w} \tag{3.4d}$$

Smoothing

The purpose of the smoothing step is to generate a visual surface instead of a cloud of spheres. This is done by applying a smoothing/blurring algorithm to the depth map. There is a multitude of different blurring algorithms but the chosen one is the bilateral filter explained in Section 2.3.3, which has the important property of edge-preserving. Edge-preserving is an important property for rendering a visual surface because otherwise, depth values will blend outside of the actual surface. There is one problem with the bilateral filter and that is that it is distance independent. For water or other transparent materials, the property of the bilateral

filter being distance independent does not greatly affect the look, but for opaque materials such as soil, this becomes a problem. Because as the camera moves further away from the soil it will appear as if it is shrinking, which is not as visible for transparent materials. Thus a modification needs to be done to make the bilateral filter distance dependent. This modification can be done by changing the constant σ values of the Gaussian functions f and g which can be seen in Equation 3.5. From Equation 3.5; σ_f is the σ for f, σ_g is the σ for g, s_f is a constant range scaling value for f, s_g is the constant spatial scaling value for g and d is the depth value of the pixel that is currently being filtered.

$$\sigma_g = s_g \cdot d \tag{3.5a}$$
$$\sigma_f = s_f \cdot d^2 \tag{3.5b}$$

The UV map also needs to be included in the blurring stage, this is to blend colors between particles. The blurring step on the UV map can be done by a less computational expensive approach than the depth blurring. Thus a GPU implemented Gaussian filter which has been explained in Section 2.3.2 is used.

Full screen quad

To render the surface a full screen quad is used, wherein the fragment shader the depth of the fragment is replaced by the depth value in the depth texture. By using this, it is possible to render the surface directly without an explicit surface mesh.

Normals from depth map

To introduce lighting to the surface in the fragment shaders one needs to calculate normals of the surface. The normals can be calculated from the depth map. The technique used in this thesis to calculate the normal of a fragment given a depth map is based on vertex normal calculation, which was explained in Section 2.1.8. The four face normals are thus calculated by sampling the neighboring corners of the current depth map position, given by the fragment's UV coordinate. These neighboring corner position and the current position are then used to form four triangles, which the face normals are calculated from. These four face normals are then averaged and normalized to generate the vertex normal, i.e. final normal for this fragment.

Shading

The shading that is done in the fragment shader for this thesis is a simple diffuse shading from a directional light source.

4 Results

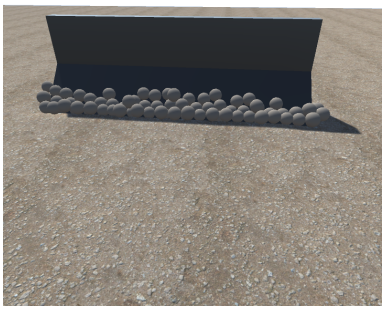
This chapter covers the result from Chapter 3, including visual inspection of the coarse and fine-scale simulation with either particle mesh or surface rendering. This chapter also includes performance measurements of the tested method with the same setup as the visual inspection, and finally a time distribution of how the GPU and CPU time is utilized.

4.1 Visual inspection

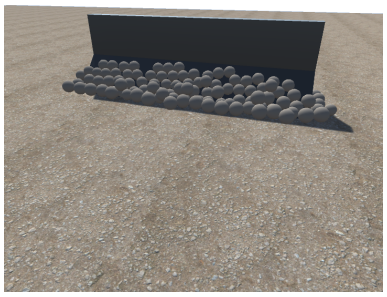
The scenario that will be used for testing the method is one where a shovel linearly moves in one direction cutting the static soil and thus creating dynamic soil. Thus dynamic soil will continuously increase in volume and mass, meaning that as the simulation runs, more and more rendering particles will be used. A scenario where the shovel drops soil is in addition also shown. The reason these shovel scenario are of interest is that they mimic simplified real-world examples, i.e. an excavator.

From Figure 7 one can see the coarse version of the simulation using particle meshes during different simulation steps. From Figure 8 one can see the equivalent coarse simulation using surface rendering.

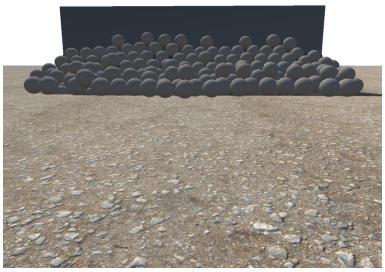
From Figures 9-16 one can see the equivalent ground simulation using the fine-scale rendering process with either particle mesh rendering or surface rendering, with decreasing fine particle mass, i.e. increasing resolution of the rendering.



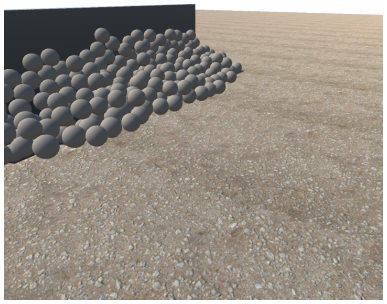
(a) Simulation step 150



(b) Simulation step 300

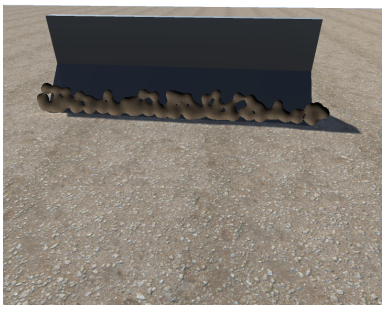


(c) Simulation step 550

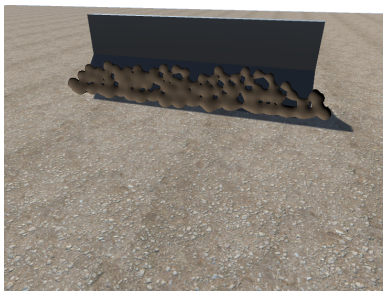


(d) Simulation step 800

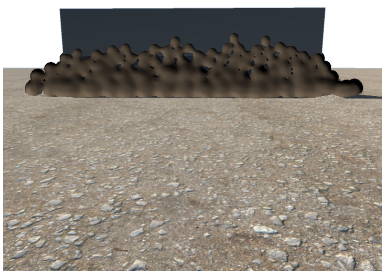
Figure 7: Particle mesh rendering of coarse particles.



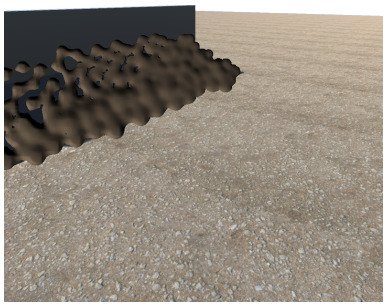
(a) Simulation step 150



(b) Simulation step 300

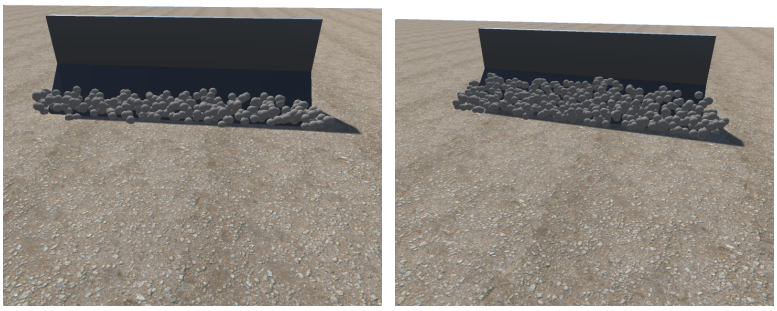


(c) Simulation step 550

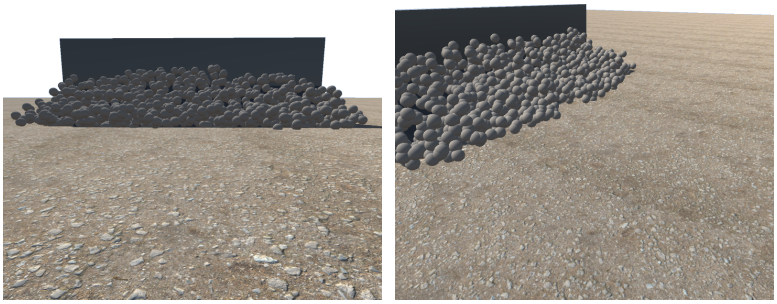


(d) Simulation step 800

Figure 8: Surface rendering of coarse particles.

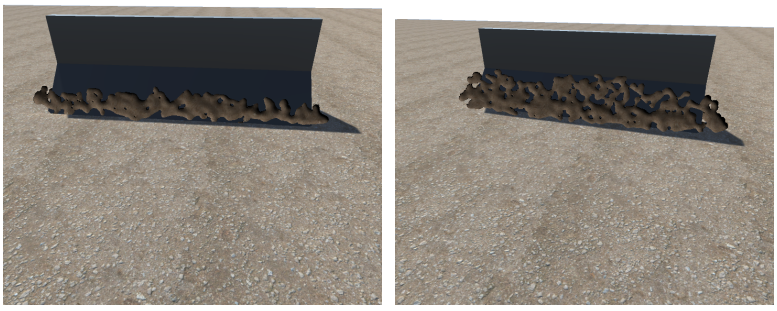


(a) Simulation step 150 **(b)** Simulation step 300

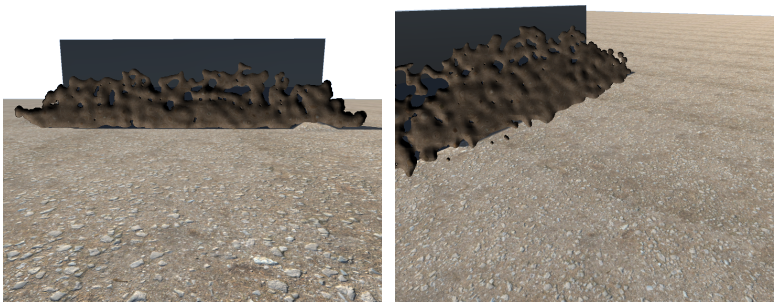


(c) Simulation step 550 **(d)** Simulation step 800

Figure 9: Particle mesh rendering of fine particles with particle mass 0.1.

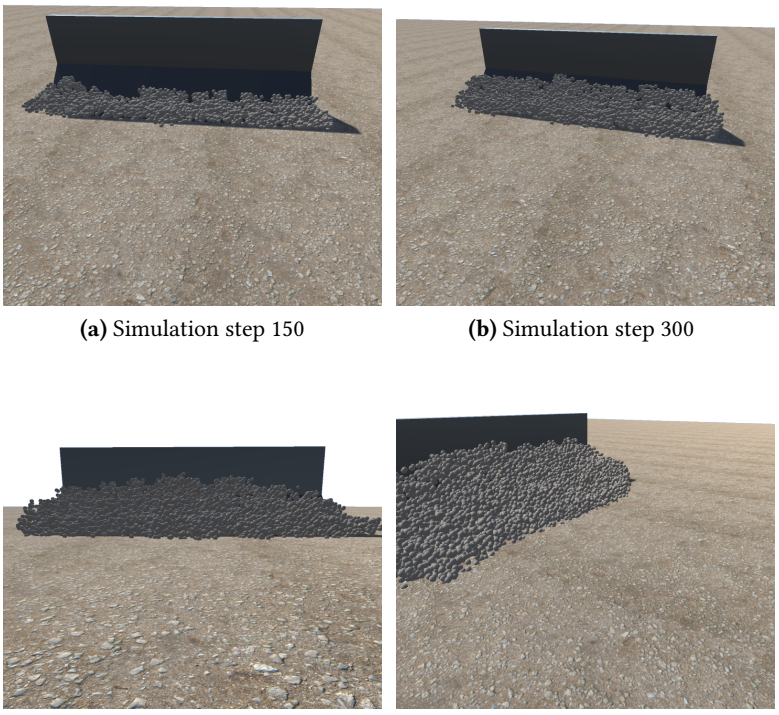


(a) Simulation step 150 **(b)** Simulation step 300



(c) Simulation step 550 **(d)** Simulation step 800

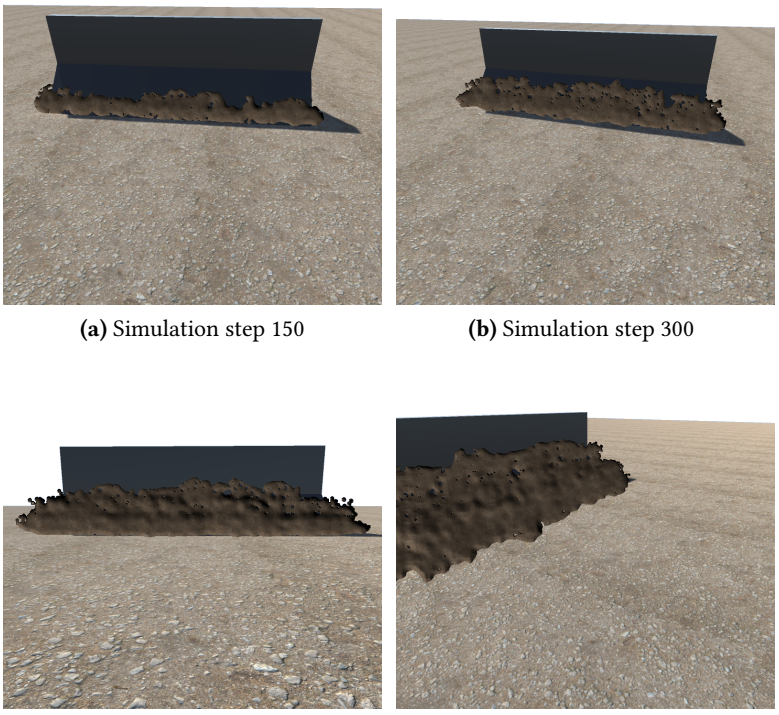
Figure 10: Surface rendering of fine particles with particle mass 0.1.



(a) Simulation step 150 **(b)** Simulation step 300

(c) Simulation step 550 **(d)** Simulation step 800

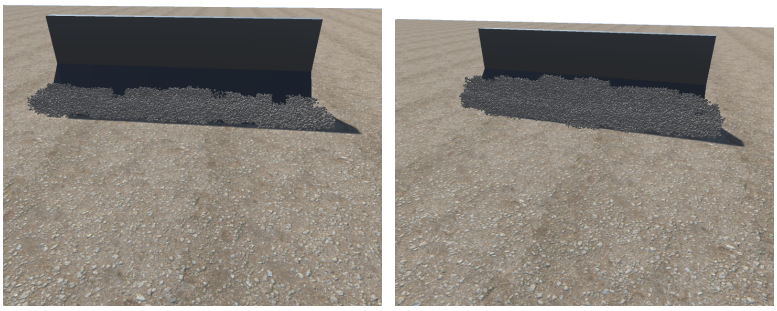
Figure 11: Particle mesh rendering of fine particles with particle mass 0.01.



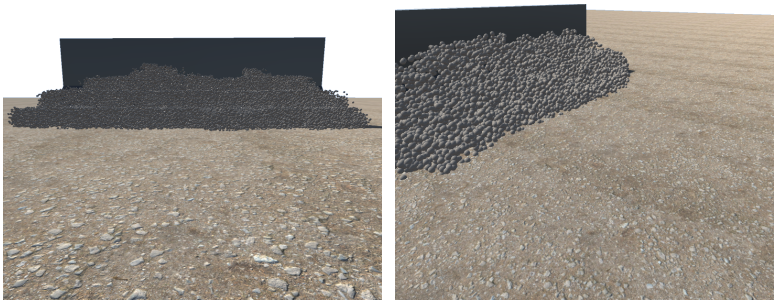
(a) Simulation step 150 **(b)** Simulation step 300

(c) Simulation step 550 **(d)** Simulation step 800

Figure 12: Surface rendering of fine particles with particle mass 0.01.

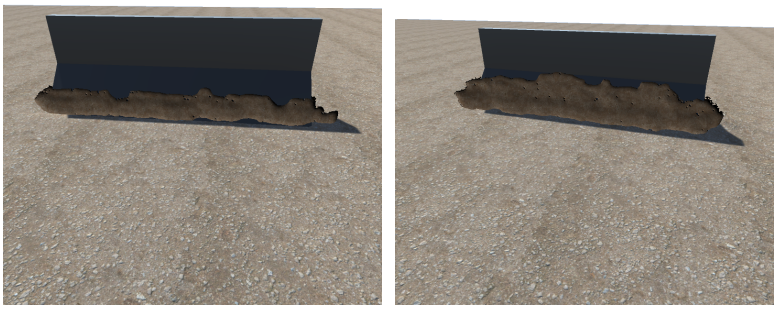


(a) Simulation step 150 **(b)** Simulation step 300

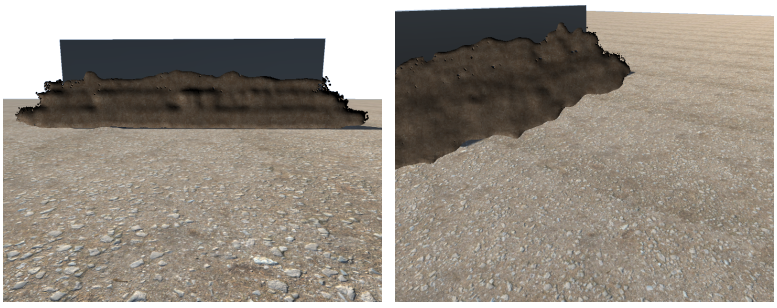


(c) Simulation step 550 **(d)** Simulation step 800

Figure 13: Particle mesh rendering of fine particles with particle mass 0.001.

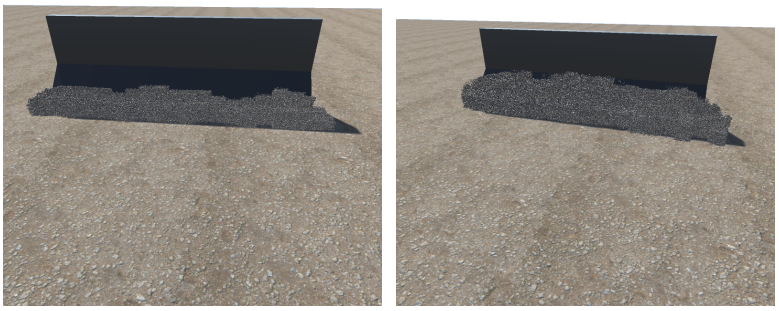


(a) Simulation step 150 **(b)** Simulation step 300

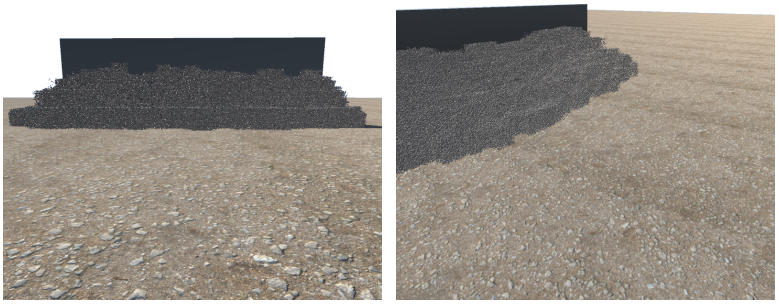


(c) Simulation step 550 **(d)** Simulation step 800

Figure 14: Surface rendering of fine particles with particle mass 0.001.

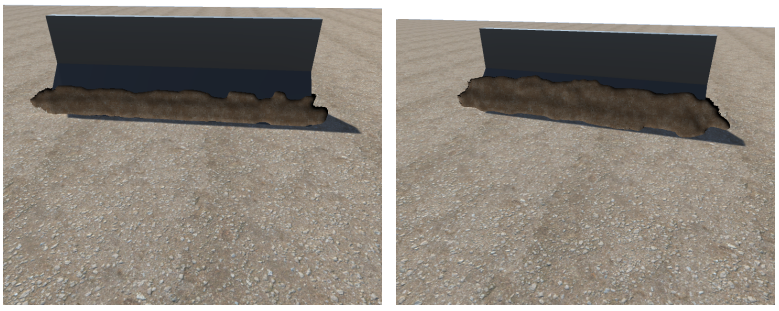


(a) Simulation step 150 **(b)** Simulation step 300

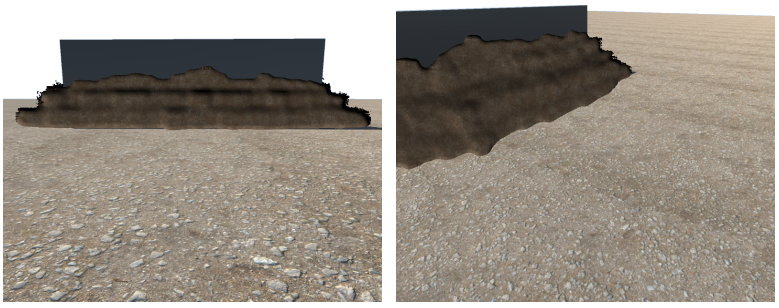


(c) Simulation step 550 **(d)** Simulation step 800

Figure 15: Particle mesh rendering of fine particles with particle mass 0.0001.



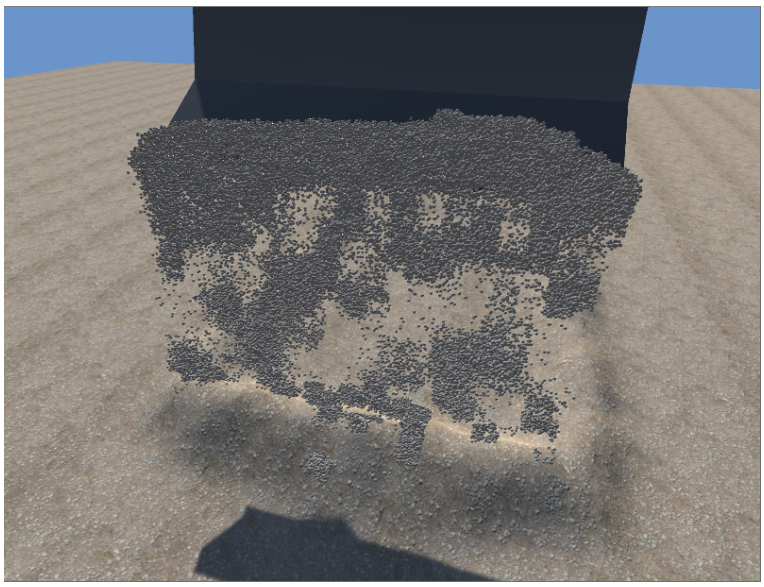
(a) Simulation step 150 **(b)** Simulation step 300



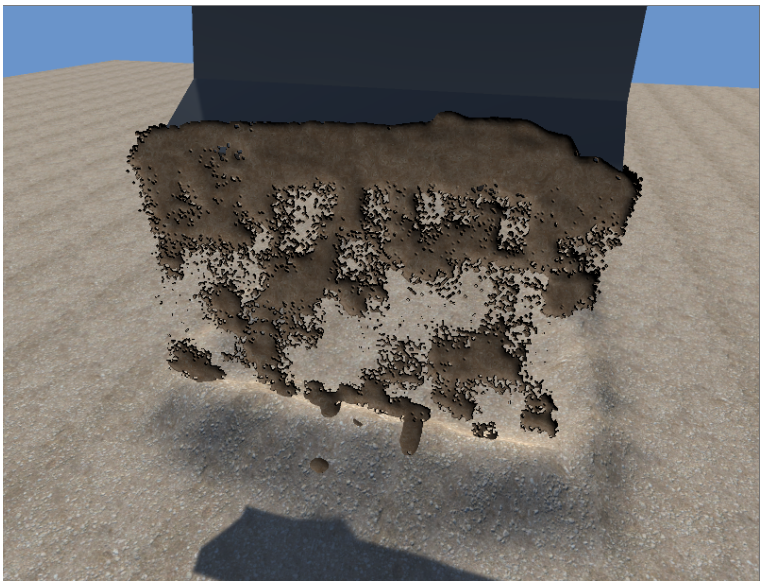
(c) Simulation step 550 **(d)** Simulation step 800

Figure 16: Surface rendering of fine particles with particle mass 0.0001.

Figure 17 showcase the visual aspects of the fine-scale rendering with a scenario where the soil falls of the shovel to present that this is not just a deformable terrain rendering and simulation.



(a) Particle mesh rendering



(b) Surface rendering

Figure 17: Comparison of falling fine particles with either mesh or surface rendering, with a particle mass of 0.001.

4.2 Performance results

The performance data was capture on a machine with the GPU: *NVIDIA GeForce RTX 2070 SUPER* and the CPU: *Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz*. The resolution of the display screen was set to 800x608 and the filter kernel size was set to 32x32x1 and for the fine-scale simulation, the kernel size was set to 1024x1x1.

Figure 18 shows the performance results from the surface rendering of the simulation shown in Section 4.1, where the computation time is the frame-to-frame time. From Table 1 one can see how many particles existed at the last frame, which was capped at frame 1200.

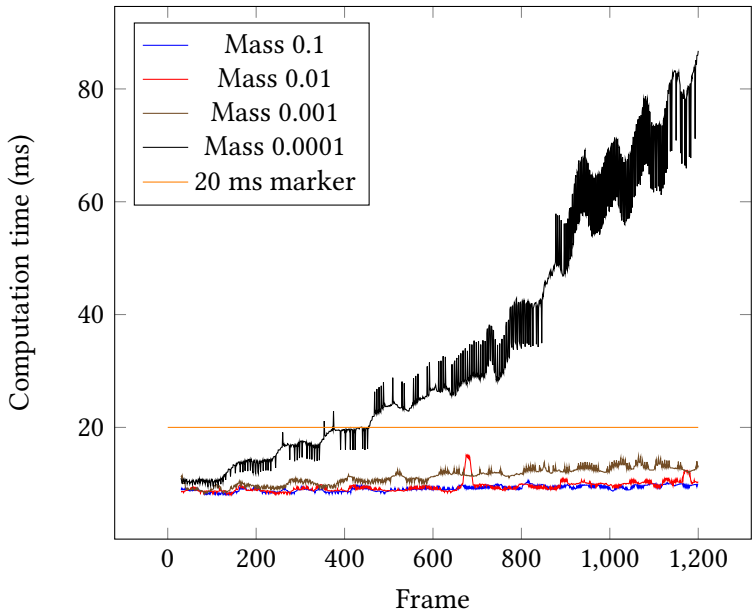


Figure 18: Averaged data from fine-scale with surface generation comparison.

Table 1 Number of particles at frame 1200 with surface rendering.

Mass	Number of particles
0.1	1045
0.01	10409
0.001	107710
0.0001	1278720

Figure 19 shows the performance results from particle rendering of the simulation shown in Section 4.1 and from Table 2 one can see how many particles existed at the last frame, which was capped at frame 1200.

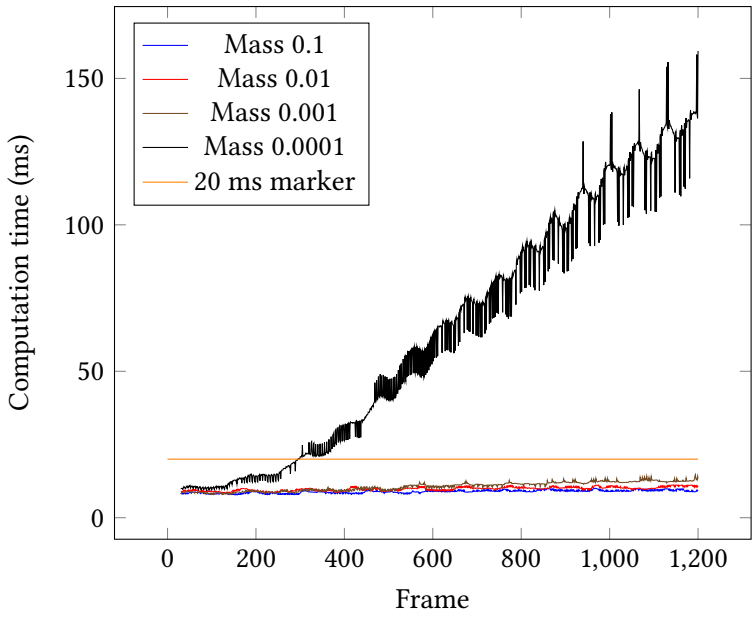


Figure 19: Averaged data from fine-scale with particle mesh comparison.

Table 2 Number of particles at frame 1200 when rendering with particle mesh.

Mass	Number of particles
0.1	1011
0.01	10252
0.001	109357
0.0001	1447406

Figure 20 shows the baseline performance data of only the coarse simulation using either surface rendering or particle rendering. This graph is used as a baseline for the performance impact of the fine-scale simulation.

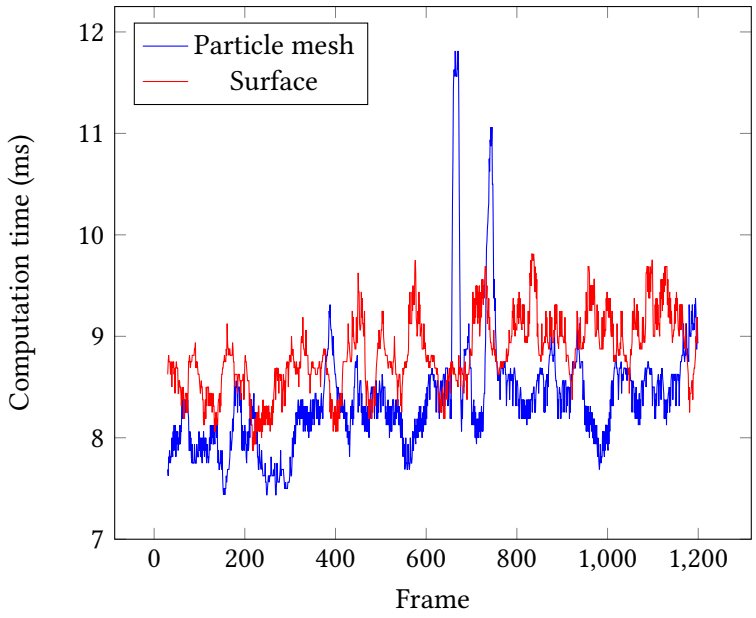


Figure 20: Averaged data from coarse particle mesh and surface generation comparison.

Figure 21 shows the combined data of all the fine-scale simulations, to show how computation time corresponds to the number of particles for both of the two rendering types. An important point in this is the 20ms marker of computation time, as it correlates to 50 FPS which in Table 3 one can see the border of how many particles are needed to cross the 20ms marker.

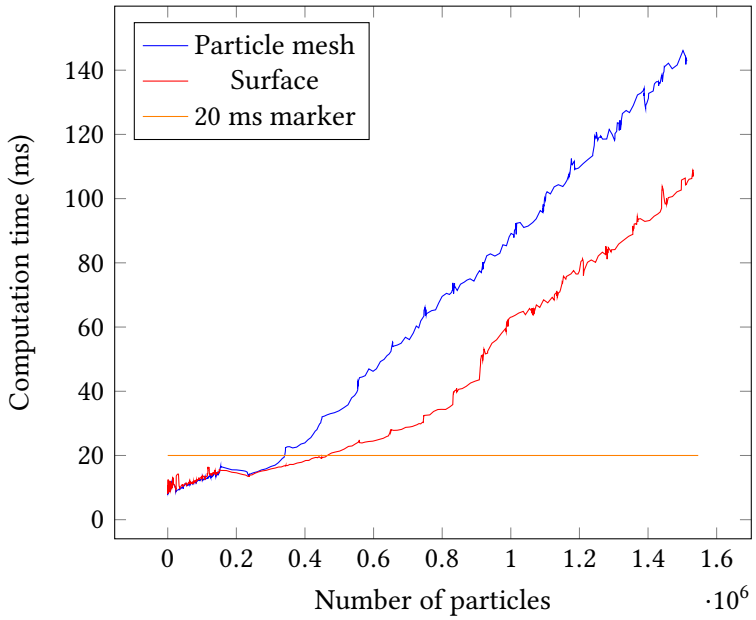


Figure 21: Averaged data from fine-scale with surface generation number of particles comparison.

Table 3 Number of particles existing when the averaged performance reached the 20 ms marker.

Type	Number of particles
Particle Mesh	341293
Surface rendering	467410

Figure 22 shows the variance of the averaged data from the computation time. This variance comes from two aspects, firstly the difference between frame and time step; secondly the fact that a skip value k is used to skip CPU particle management each k time steps.

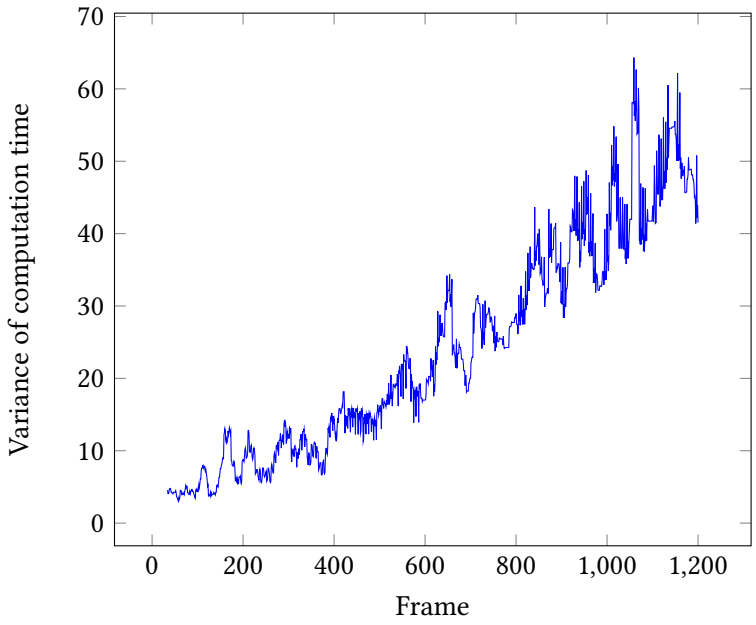


Figure 22: Variance of fine particle surface generation of mass 0.001.

4.3 Time distribution

Figure 23 shows the CPU time charts of the particle management algorithm and Figure 24 shows the whole GPU time charts. These values have been generated from a deep profiler as such the data might not 100% showcase the real percentages as some interaction might suffer more from the profiler, but this should still give a good overview of how the CPU and GPU time is spent and distributed.

- *Get data from GPU,* the sum of all the waiting for the GPU to complete its current task, and the transfer sum of all the GPU to CPU transfer time.
- *Check and remove mass,* the function that removes mass from the mass voxel grid for each particle and also checks if each particle should still be alive based on the mass voxel grid.
- *Buffer data,* send data to the GPU.
- *Data conversions,* data conversion between two different coordinate systems used.
- *Clone grid and sort coarse particles,* creating the EMA voxel grid and the sorting of the coarse particles in all axes for the restricted spawning system.
- *Spawn particles,* the act of going through all the active voxels and if there is mass left, spawn fine-scale particles.
- *Dispatch,* send dispatch call to the compute shader.
- *Filter,* the smoothing/blurring step algorithm that is used on the depth map and on the UV map.
- *Fine-scale simulation,* i.e. the act of calculating the velocity of each fine-scale particle and updating the particle's position.
- *Draw mesh and draw depth map,* the drawing of random meshes and the creation of the fine-scale particle depth map.
- *Idle,* i.e. the idle time of the GPU.

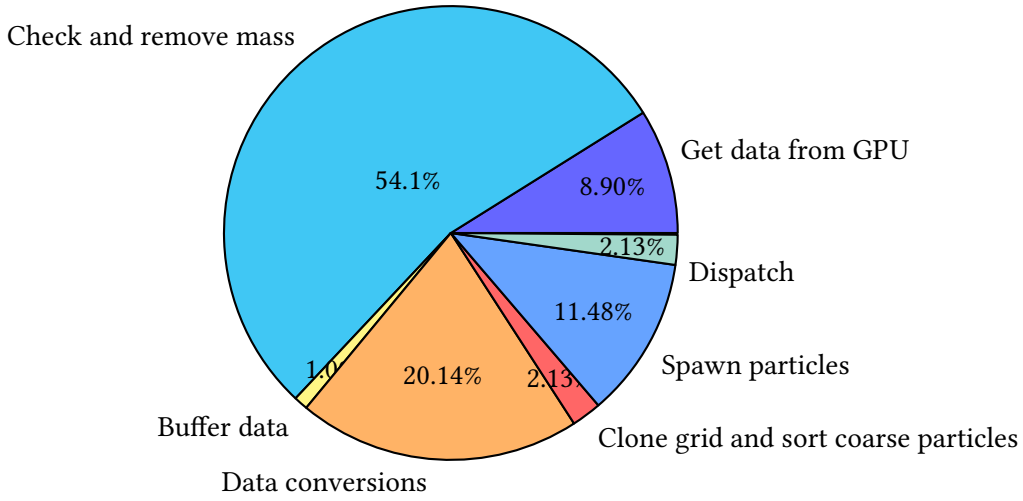


Figure 23: CPU time distribution of particles surface generation with 0.001 mass.

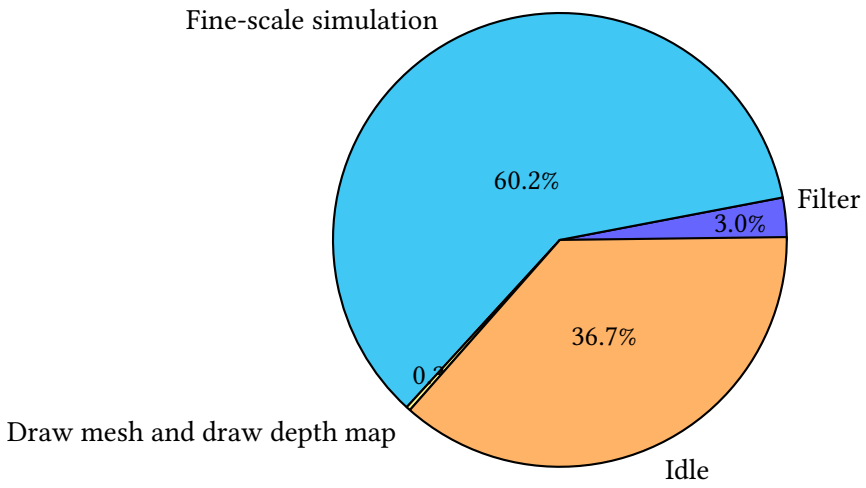


Figure 24: Whole GPU time distribution of particle surface generation with 0.001 mass.

5 Discussions

This Chapter covers discussion around the results generated in Chapter 4 using the method described in Chapter 3, where the results both concern visual results and performance results.

Side-note: A CPU only version was first developed to test if/and compare if the data transfer between the GPU and CPU was going to be a bottleneck for the simulation that needs to happen each time step. It was concluded that the parallelism of the GPU outweighed the data transfers time so the CPU only version was scrapped and will not be included.

5.1 Visual

The first aspect that needs to be discussed concerning the visual side of the method and the result is how particles move. If one were to just use the velocity data from the velocity grid one would get a very blocky/rigid movement from the fine-scale particles based on the voxels, i.e. the fine-scale particles does not move in a natural way if one only directly uses the velocity data. Thus one can introduce interpolation of the surrounding voxels based on the distance from the center of the voxel to the particle. This results in that a more natural and smoother movement of the fine-scale particles emerges. Lastly, a new weight modifier is introduced to the interpolation where the weight is based on the mass of the voxel. This results in two properties. Firstly the particles will move in a mass-oriented way, meaning that the particle will move in the way of the most mass. Secondly, it removes the drag created by voxels with zero mass and velocity, meaning that particles that are at the edge of the voxels will not lag behind and thus do not need to be removed for occupying space with no mass in them.

The second visual aspect concerns spawning and despawning of fine-scale particles. As stated in Chapter 3 particles are spawned based on the mass of a voxel, if there is mass left after tallying all the particle's mass in the voxel, then, new particles will be spawned randomly inside the voxel. If there is less mass in the voxel then particles will be despawned. This would create a rather blocky pattern of particles but there are two aspects that combat the blocky voxel pattern. The first is that the particle movement interpolation reduces the blocky pattern over time. The second aspect comes from the restricted spawning system, which restricts the spawning of particles in a voxel to where coarse soil particles have an influence. These two aspects reduce the block pattern and create a more natural soil particle look. But the artifact can still be seen in certain scenarios.

An EMA filter as said in Chapter 3 is used on the mass grid, this results in two visual properties. The first property and most visually obvious property is that the particles will fade in and out when spawning and despawning, reducing the discrete popping effect that occurs when particle spawn and despawn from the static soil. The second property the EMA gives is a more stable spawning and despawning system, in that as the coarse soil moves EMA gives more time for the fine-scale system to move and follow which results in fewer particles spawning and shortly after despawning, i.e. particle flickering. The particle flickering effect

is also damped by the mass buffer ϵ from Equation 3.1.

There is another visual artifact that shows using this system other than the block pattern. This visual artifact follows from that the fine-scale particles do not care about external forces such as gravity. This results in that the fine-scale particle will float. This artifact is not as prevalent when rendering with surface rendering as particles will blend together.

The increasing resolution of the fine-scale particle has a direct effect on the visual fidelity, but the effect on the visual fidelity varies depending on if particle mesh rendering or surface rendering is used. As the resolution of the fine-scale particles increases, one can see a direct continuous improvement in the visual fidelity of the particle mesh rendering. However, as the particle mass starts to get lower than 0.001 then the rendering fidelity starts to get diminishing returns as the resolution of the particles from a reasonable distance gets small enough for it to start to look like noise. For rendering with surface generation one does not get the noise problem when using small scale particles and the point at which increasing the resolution only gives diminishing returns in visual fidelity starts earlier at around a particle mass of 0.01 to 0.001. This results in that to get a high visual fidelity one does not need as high of a fine-scale particle resolution when rendering with surface rendering instead of particle meshes.

Thus combining all that has been discussed and the figures from Section 4.1 one can start to form a comparison between the coarse rendering and the fine-scale rendering. Comparing the coarse particle mesh rendering with the fine-scale rendering, it can be noted that even with the simpler velocity calculation of the fine-scale particles just the fact the number of particles increases results in an increase in visual fidelity. Using surface rendering on top of this gives more depth to the soil rendering in terms of believability that soil is what is being rendered. Using surface rendering directly on the coarse particles does not give a perceived visually pleasing result. This comes from two facts, firstly that there is too little visual information to render a convincing depth map of the soil surface and secondly the size of the particle means that the surface generation looks distorted.

5.2 Performance

From Figure 18 and 19 one can note that the performance measurements using the hardware shown in Section 4.2 is adequate. The main requirement stands that the fine-scale rendering shall have an as low impact as possible on the overall performance of the simulation and the aligning goal that run-time cost should in this case not exceed the 20 ms marker. To this it can be seen from Figures 18 and 19 that the simulation does not exceed the 20ms marker and stays under the 20 ms marker as long as the mass of the fine-scale particles does not approach 0.0001, meaning more than 450k particles.

From Figure 21 which shows how many particles are being rendered and simulated in response to the performance of the two rendering methods. From that graph, it can be seen that the computation time grows linearly with the number of fine particles used. From Figure 21 one can construct a comparison between the performance impact of the two rendering methods. From this graph, two main points can be concluded when comparing the rendering methods, firstly that the surface rendering methods gives better performance than the particle mesh rendering methods. This comes primarily from the number of triangles that needs to

be rendered as with a high number of particles is too high even with a rather simple particle geometry of around 80 triangles per particle. Secondly as can be seen from the graph the particle mesh rendering starts to increase earlier than the surface rendering method and after which they increase at around the same pace. The early increase of the particle mesh rendering shows that the particle mesh rendering version is more expensive but as the number of particles increases the main bottleneck of the fine-scale rendering and simulation lies in the simulation and not in the rendering of the fine-scale particles.

As stated before from just Figure 21 one could conclude with a high degree of certainty that the bottleneck of the method described in Chapter 3 lies in the simulation part. Continuing from this, if one focuses on Figure 22 which shows the variance of the computation time. The variance as explained in Section 4.2 mainly comes from the fact that a skip value is used to skip CPU work each k time step, i.e. not doing any particle management and only simulating the particles in the GPU. One can see that as the time goes on and within the number of particles increases the variance also increases, this points towards that the main bottleneck lies in the CPU work, i.e. the management of particles. Because if the main bottleneck lied in the GPU work then the variance would not be as high and would not increase as much overtime as it does.

5.3 Time distribution

The time distribution data presented in Section 4.3 shows where there is still time budget to be used, where a bottleneck might exist, and thus where to look for in terms of performance improvement opportunities. From Figure 24 which shows the whole GPU time distribution, one point is reinforced which was concluded from Section 5.2. This point being that the surface rendering does not have a great impact on the performance of the whole simulation as the drawing of the depth map and filtering the depth map only just exceeds 3% of the GPU time used and most time used is for the fine-scale simulation and rest is just idle time, which gives the method room to still grow concerning GPU usage.

The more interesting time distribution figure is Figure 23 which shows just the fine-scale particle management CPU time usage. Most of the time is spent on the particle mass checking and mass removing from voxels. This specific part is thus an algorithm that is based on the number of particles each using and modifying the temporary voxel grid mass data, to check if they should live or not. This algorithm is thus not a trivial one to parallelize for performance improvement. This comes from that all particles share the same data point, i.e. the voxel grid. Thus if one wants to parallelize the algorithm, the point of parallelization would need to be from the active voxels and not the particles. This is not trivial because of how the particles need to be stored for both the CPU and the GPU calculations.

6 Conclusions

The purpose of this thesis is to investigate a way to upscale the rendering and with that increase the visual fidelity of the coarse soil simulation, with spherical discretization. The methods chosen and created for the investigation were an HR-based method that uses a coarse simulation to guide a more simple fine-scale simulation, which was rendered using either surface rendering or particle meshes. Thus this chapter will cover conclusions of the method based on Chapter 4 and Chapter 5, i.e. Results and Discussions. At the end of this chapter possible future work related to this thesis will also be covered.

The upscaling of the coarse simulation is done by using the coarse simulation to guide a fine-scale simulation where this fine-scale simulation does not calculate the direct external and internal forces and instead only uses the velocity from the velocity voxel grid with the use of interpolation. Where the use of the interpolation and of a restricted spawning system reduced the visual block pattern, but this visual artifact can still be seen in certain scenarios.

An increase in visual fidelity of the coarse rendering is achieved without a huge impact on the overall simulation performance, were huge in this case means not dropping under the 50 FPS mark. Where the perceived sweet spot when min-maxing the visual fidelity to the performance is at a particle mass of around $0.01 - 0.001$ when the rendering mode is surface rendering.

When comparing the use of surface rendering to the particle mesh rendering mode it can be seen that surface rendering does not need as high particle resolution for the same perceived visual fidelity. It was also shown that the visual artifact of floating particles was also reduced using surface rendering.

Finally and summarizing, this thesis presents a method for upscaling and thus increasing the visual fidelity of a coarse soil simulation without a huge impact on the overall simulation.

6.1 Future work

There are two aspects concerning the work in this thesis that would need future work and investigation; performance improvements, and improvement of the visual quality of the soil rendering.

When it comes to performance improvements, i.e. optimization of the method, it is shown that the current bottleneck lied in the CPU management of the particles. Thus an interesting point to look at is how one can manage the particle data in such a way that one can parallelize the particle management on the CPU in an efficient way. The other way to go is to look into if a significant part of the current CPU particle management can be offloaded to the GPU, which is non-trivial because data needs to be created and destroyed based on the whole set.

The first part of the visual side to look into, which this thesis did not, is shading and lighting, i.e. look into how one can introduce more complex shading techniques in the shaders to further increase the visual fidelity without huge performance impacts.

The second part of the visual side is to look into how different blurring/smoothing filters might affect the look of the soil surface rendering. In other words how different filters might be better for different types of soil or if there is one that works for most types of soil with the right texturing.

The final visual aspect to look into is to further reduce the perceived discrete popping effect that occurs when particle spawn and despawn from the static soil, i.e. to hide the transition between the static height map and the dynamic soil rendering. This effect was reduced by the use of EMA, but the effect still persists.

References

[1] Tomas Akenine-Mller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Fourth Edition*. 4th. USA: A. K. Peters, Ltd., 2018. ISBN: 0134997832.

[2] Tamás Umenhoffer, László Szirmay-Kalos, and Gábor Szijártó. "Spherical billboards and their Application to Rendering Explosions." In: *Graphics interface*. 2006, pp. 57–63.

[3] Joey de Vries (@JoeyDeVriez). *LearnOpenGL*. URL: https://learnopengl.com/Getting-started/Coordinate-Systems.

[4] Michael Satran, Steven White, and Jacobs Mike. *Compute shader numthreads*. Ed. by Microsoft. URL: https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/sm5-attributes-numthreads.

[5] Håkan Haglund, Mattias Andersson, and Anders Hast. "Snow accumulation in real-time". In: 007 (2002), pp. 11–15.

[6] Martha W Evans, Francis H Harlow, and Eleazer Bromberg. *The particle-in-cell method for hydrodynamic calculations*. Tech. rep. Los Alamos National LAB NM, 1957.

[7] J. Brackbill and H.M. Ruppel. "FLIP: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions". In: *Journal of Computational Physics* 65 (Aug. 1986), pp. 314–343.

[8] Yongning Zhu and Robert Bridson. "Animating sand as a fluid". In: *ACM Transactions on Graphics (TOG)* 24.3 (2005), pp. 965–972.

[9] Christoph Ammann et al. "The Birth of Sandman." In: *SIGGRAPH Sketches*. 2007, p. 26.

[10] Ivan Alduan, Ángel Tena, and Miguel Otaduy. "Simulation of High-Resolution Granular Media". In: (Sept. 2009).

[11] Markus Ihmsen, Arthur Wahl, and Matthias Teschner. "A Lagrangian framework for simulating granular material with high detail". In: *Computers & graphics* 37.7 (2013), pp. 800–808.

[12] William E. Lorensen and Harvey E. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm". In: *SIGGRAPH Comput. Graph.* 21.4 (Aug. 1987), pp. 163–169. ISSN: 0097-8930.

[13] Matthias Müller, Simon Schirm, and Stephan Duthaler. "Screen space meshes". In: (2007), pp. 9–15.

[14] Myles Carter. *Rendering Even More Realistic Soil in Vortex Studio*. Ed. by Vortex Studio. URL: https://www.cm-labs.com/vortex-studio/resources/blog-screen-space-mesh-rendering-realistic-soil/.

[15] Bart Adams, Toon Lenaerts, and Philip Dutré. "Particle splatting: Interactive rendering of particle-based simulation data". In: (2006), pp. 16–16.

[16] Wladimir J van der Laan, Simon Green, and Miguel Sainz. "Screen space fluid rendering with curvature flow". In: (2009), pp. 91–98.

[17] Simon Green. “Screen space fluid rendering for games”. In: *Proceedings for the Game Developers Conference*. 2010.

UMEÅ UNIVERSITY