Dissertation Type: research

University of BRISTOL

DEPARTMENT OF COMPUTER SCIENCE

Real-Time Simulation of Sands and Soils using Voxel Grids

Thomas Jennings

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Friday 4th May, 2018

Declaration

Thomas Jennings, Friday 4th May, 2018

Contents

List of Tables

Executive Summary

When simulating a terrain composed of granular materials, such as sand and soil, balance between performance and accuracy is a very important consideration when choosing an implementation. In real-time applications, performance is even more important as a slow simulation can be frustrating to deal with and also detracts from the realism of the simulation. With applications such as virtual reality, which requires fairly high frame rates, many traditional terrain rendering implementations are simply not up to par. This thesis explores one implementation which utilises voxel grids to store, calculate and render the terrain data at runtime. The terrain mesh is extracted from the voxel grid every time step using the Marching Cubes algorithm. I have evaluated and attempted to optimise various attributes of the implementation in the interests of both performance and quality. My hypothesis is that in a voxel based terrain simulation, there many areas that can be optimised and expanded upon to create an application which can be tuned for various different use cases.

- I wrote over 3000 lines of source code in C++ using OpenGL and GLSL in order to create a real-time simulation of granular materials

Supporting Technologies

- I used C++ with OpenGL and the OpenGL Mathematics library.
- I used GLFW, an open source OpenGL library, and GLAD, an extension loading library for OpenGL.

Chapter 1

Contextual Background

In computer graphics, terrain rendering refers to the rendering of three-dimensional landscapes in a virtual system. In many applications, this is done by assuming that the terrain will remain static which allows the terrain data to be constructed beforehand and then imported into the application. This works fine for terrains consisting of solid materials, such as stone, but for granular materials, such as sand, it creates an unrealistic representation of the surface. For terrain composed of such materials to behave properly, it must be stored and rendered in a different manner and calculations will have to be done constantly at runtime. In order to implement this, it is necessary to find methods which have a minimal impact on the applications performance, as for real-time applications, such as video games and simulations, users can often value performance over quality. While Graphics Processing Units (GPUs) are constantly improving, high end hardware can be quite expensive, so it is also important that the solution runs fast on average devices.

The goal of this research is to implement part of an existing method and evaluate and optimise it's performance and quality. In particular, I implemented the Soil Advection algorithm described by Geiger[3] and the Marching Cubes[5] algorithm. I then explored the various attributes of the simulation to see where performance was being sacrificed for minimal gains in quality and vice versa, in order to optimise the implementation.

There are many use cases for realistic and efficient terrain representations, particularly with the rise of virtual reality software. Virtual reality demands both high frame rates and high quality from its applications in order to provide the most realistic virtual experience possible. This type of implementation would benefit the end-user on that they will be able to achieve a more realistic virtual experience at a lesser cost.

By using a voxel grid, it is easy to add and remove different attributes of the soil, allowing for the creation of a modular application which would allow users to choose performance or quality at will. This modular approach also aids the development of such applications as a developer can easily created a new module that can simply be added onto the existing application.

The main goal of this project is to explore and optimise an implementation for the rendering of terrain composed of granular materials using voxel grids. More specifically, the aims are:

1. Implement the Marching Cubes and soil advection algorithms as described.
2. Attempt to optimise the implementation for performance and quality.
3. Identify whether there exists a ”best” configuration of the implementation for all use cases.

Chapter 2

Technical Background

2.1 Voxel Grids

A voxel is a quantity in a regular, three-dimensional grid. They can be compared to pixels such that they are explicitly encoded with their position in the grid, but rather a value relevant to what the grid represents and their position is inferred relative to their surrounding voxels. This means that voxel grids are suited towards representing non-homogeneously filled three-dimensional structures, which is ideal for simulating terrain composed of granular material. Voxel based terrains are represented in a voxel grid using a density field[4]. The density field holds the density of the terrain at the corners of each voxel. The midpoint between the maximum and minimum possible density values is defined as the threshold value whereby density values on one side of the threshold are defined to be outside of the surface of the terrain and values on the other side are defined to be within the surface of the terrain.

2.2 Rendering

When rendering from a voxel grid, the various methods are split into two main categories, *direct*[2] and *indirect* volume rendering. This project uses an indirect volume rendering method to extract a geometric mesh from the density field, thus allowing traditional rendering techniques to be applied to the extracted mesh. The algorithm in question is called the Marching Cubes[5] algorithm. This algorithm uses the density values and the threshold value to decide which edges of the voxel will be intersected by the surface. As each voxel has eight vertices, there are a possible $2^8 = 256$ possible configurations of intersections for each voxel. This allows for the creation of a lookup table so that the voxel configuration can be calculated very quickly. The positions of the intersections can then be calculated by interpolation of the density values at both ends of the intersected edges. This will produce a seamless mesh of triangles from the voxel grid.

2.3 Shaders

The majority of computation for the simulation will be done using OpenGL Compute Shaders. A shader is a program designed to run on a GPU and is typically part of the rendering pipeline. A compute shader is separate from this and is generally used to compute arbitrary information in preparation for rendering. In this implementation, we can use these shaders to easily parallelise the various operations we perform on the voxel grid as compute shaders can be dispatched in three-dimensional sets, thus allowing us to dispatch a shader for each voxel. They also allow us to use three-dimensional textures, which are well suited to represent a voxel grid.

2.4 Advection

As granular material behaves in a manner similar to a fluid, we can apply fluid based advection algorithms[1] to simulate the attributes of the material and transport them through the voxel grid. The given advection algorithm[3] works by displacing each voxel in the grid according to its velocity over the duration of the time step. The attributes will then be distributed into the voxels that are overlapped. For the voxel given by the co-ordinates i, j and k, the fraction of its attributes to be distributed to a voxel given by the co-ordinates a, b and c is given by Equation 2.1.

$$\lambda_{a,b,c}(i,j,k) = \Omega(i + (\vec{u}_{i,j,k})_x \Delta t, a) \cdot \Omega(j + (\vec{u}_{i,j,k})_y \Delta t, b) \cdot \Omega(k + (\vec{u}_{i,j,k})_z \Delta t, c) \tag{2.1}$$

Ω calculates the overlap between two voxels in a single dimension and is given by Equation 2.2 where x1 and x2 are the components of the voxel positions in the same dimension.

$$\Omega(x1, x2) = 1 - min(1, |x1 - x2|) \tag{2.2}$$

Chapter 3

Project Execution

3.1 Soil Representation

The state of the simulation will be stored in a voxel grid, where each voxel will hold the attributes of the soil contained inside it, in this case a density value and a three-dimensional velocity vector. Each voxel with co-ordinates x, y and z, the density of the soil contained within the voxel will be denoted by $\rho_{x,y,z}$ and the velocity of the soil in the voxel will be denoted by $\vec{u}_{x,y,z}$. The sets of density values and velocity vectors will be referred to as the density field and velocity field, respectively.

The density values represent the fraction of the voxel's total capacity which is filled with soil, thus a voxel with a density value of 0.0 is completely empty and a voxel with a value of 1.0 is completely full. It is possible for the density value to momentarily exceed 1.0, at which point a voxel is considered to be $overflowed$ and measures are taken to distribute the overflowing density before the next time step. The soil contained within a voxel is assumed to be piled up uniformly from the bottom of the voxel. The velocity vectors represent the direction and speed of the contents of the voxel. The *state* of the voxel grid at the n^{th} time step will be denoted by β^n, where the density and velocity values at a given voxel with co-ordinates x, y and z are given by $\rho^n_{x,y,z}$ and $\vec{u}^n_{x,y,z}$ respectively.

3.1.1 Soil Simulation

The simulation will attempt to accurately simulate the projectile motion of a mass consisting of a granular material. The main influence in this case will be the downward acceleration of gravity. The state at the start of the n^{th} time step, β^n, will have a transformation applied to it to calculate β^{n+1}, the state at the end of the time step, taking into account the time between frames, Δt. The state of an individual voxel at the start of the n^{th} time step is denoted by $\beta^n_{x,y,z}$, where x, y and z are the co-ordinates of the voxel. The transformation used is based on the GPU-based soil advection presented by Geiger[3]. Included in the soil advection transformation is a heuristic to handle overflowed voxels.

Soil Advection

The format of the advection algorithm is given by Equation 3.1

$$\beta^{n+1} = advection(\vec{u}, \Delta t, \beta^n) \tag{3.1}$$

The algorithm is designed such that each $\beta^{n+1}_{x,y,z}$ can be calculated in parallel, which lends itself to a GPU implementation. In the original implementation, each voxel would only consider inflows of density from its adjacent voxels. This approach requires you to set a maximum velocity allowed so that a voxels contents will not move far enough to cross over the neighbour voxels and thus the portion of soil would effectively vanish from the simulation. I investigated the effect on performance if the neighbourhood of voxels was increased and whether there was a noticeable increase in simulation accuracy.

Each voxel with co-ordinates x, y and z has exclusive writing privileges for its own data, and shared reading privileges for itself and its neighbourhood of voxels given by co-ordinates a, b and c satisfying the constraints given in Equation 3.2 where K is the neighbourhood size. $K = 1$ was the neighbourhood size used in the original implementation.

$$\begin{aligned} x - K \le a \le x + K \\ y - K \le b \le y + K \\ z - K \le c \le z + K \end{aligned} \tag{3.2}$$

The density contained within a voxel is distributed according to the velocity into surrounding voxels. If a voxel is at it's max soil capacity, then it is considered to be *locked* and no more density can flow into it. If soil attempts to flow into a locked voxel, then the density will flow back into its original voxel. The velocity of the soil in a voxel will also be transmitted with the density, and the final velocity of the target voxel will be calculated as the weighted average of all inflowing densities. In the case that density flows back into a voxel due to attempting to enter a locked voxel, it will return with zero velocity. A locked voxel will become unlocked once it gains velocity and some of its density flows out. It is assumed that the edges of the voxel grid are always locked to prevent density from flowing out of the simulation and vanishing.

The density that flows into a voxel with co-ordinates x, y and z from a voxel with co-ordinates a, b and c is given by Equation 3.3. $\rho^n_{a,b,c}$ is the density within the originating voxel at the start of the time step and $\lambda_{x,y,z}(a, b, c)$ is the fraction of the originating voxel which overlaps with the receiving voxel.

$$inflow_{x,y,z}(a, b, c) = \rho^n_{a,b,c}\lambda_{x,y,z}(a, b, c) \tag{3.3}$$

The density backflow due to colliding with a locked voxel is equal to the density that attempted to flow into the locked voxel. While a voxel can receive inflow from itself, it cannot receive backflow. The final density, $\rho^{n+1}_{x,y,z}$, of an unlocked voxel is given by the sum of inflows and backflows from all its neighbours. For a locked voxel, this is instead the sum of all backflows plus any inflow from itself.

After all densities and velocities have been distributed, voxels with non-zero density values will be accelerated based on the downward gravitational acceleration vector and the duration of the previous time step. After this, voxels with a non-zero density be accelerated by a *wind* value if they are exposed. A voxel is exposed if there is and empty voxel adjacent to it in the direction that the wind is coming from, which will be controlled by the user.

Overflow Management

After the advection algorithm is complete, it is possible that some voxels exceed their maximum capacity. In order to resolve this issue, a heuristic is applied to redistribute the overflowed values. There are several different heuristics that can be applied. One such example heuristic is to simply push all overflowed values into the voxel above. This is the heuristic used in the original implementation. Listed below are some heuristics I have investigated the strengths and weaknesses of to see if there is a "best" heuristic for this scenario.

1. Push overflowed values to the voxel above
2. Evenly distribute to adjacent voxels
3. Distribute to adjacent voxels in the row above, giving more weighting to the centre voxel
4. Distribute evenly to adjacent voxels on the same level

It cannot be guaranteed that all overflows will be solved particularly in a system that contains large amounts of material, so it becomes necessary to do several passes of overflow management each time step. In the original implementation, four passes were used per time step. I have investigated the effect of more or less passes each time step and it's impact on quality and performance.

3.1.2 Visualisation

In order to render the voxel density values, a triangle mesh must be extracted from the voxel grid. This has been acheived using the Marching Cubes[5] algorithm. For this algorithm, a threshold density value must be selected where values above the threshold are considered to be within the surface and values at or below the value are outside the surface. This value is typically defined as half the max density, so for our purposes this threshold shall be 0.5. The Marching Cubes algorithm is performed on the GPU, with a single thread for each voxel calculating its containing triangles. These triangles are then added to a global buffer and passed directly to the vertex shader to be rendered. The rendered output triangles are assigned a colour based on their altitude to make it easier to see the changes in the terrain.

3.2 Implementation

3.2.1 State Storage

To record the state of the simulation, 3D textures were used as they can be easily used to represent a voxel grid and allow for the storage of up to 4 values per voxel. Each texture had a size equal to the desired size for the voxel grid, with 2 textures per value to be recorded, and 2 extra to handle the overflow heuristic. Textures were created in twos in order to have an input and output texture for each value to avoid reading and writing conflicts and this gave a total of 6 textures.

3.2.2 State Computation

The algorithms were implemented in a series of compute shaders, which perform parallel computation on the GPU.

Soil Generation Shader

The soil generation shader would compute the initial state of the simulation before the render loop. It would initialise the velocity and overflow textures to zero and initialise the density textures with values based on a density distribution function.

Soil Advection Shader

The soil advection shader applies the algorithm described in Section 3.1.1 It takes both the input density and velocity textures and writes to the output density and velocity textures. It also takes the value K for the neighbourhood size and a vector for wind velocity, which is used to apply acceleration to voxels that contain a non-zero density and are exposed to the given wind direction.

Overflow Resolution Shader

The overflow resolution shader applies a heuristic to resolve any voxels whose density exceeds their maximum capacity. This shader will be applied multiple times each time step to ensure the minimal number of voxels are overflowed at the start of the next time step. Originally, the same heuristic is applied each time, but it is possible that alternating heuristics or applying different heuristics in a certain order would provide better results. This shader takes both the input density and overflow textures and writes to the output density and overflow textures.

Marching Cubes Shader

The marching cubes shader applies the Marching Cubes[5] algorithm. It takes as input the density texture and two lookup tables which are required for the algorithm. The output is a large buffer containing all the generated triangles which form the soil mesh which is input directly into the vertex shader to be rendered.

Chapter 4

Critical Evaluation

4.1 Results

4.1.1 Neighbourhood Size and Soil Quantity

My first experiment was to see the effect of increasing the size of the neighbourhood to be evaluated during the soil advection algorithm. The results can be seen in Table 4.1 These results show that increasing the

Neighbourhood Size K	Radius of initial sand sphere	Frames per Second
1	8	60.06
	16	60.03
	32	60.02
2	8	44.28
	16	44.24
	32	42.29
3	8	24.95
	16	24.82
	32	22.72

Table 4.1: The effect of neighbourhood size and total soil density on frame rate

neighbourhood size has a drastic effect on the frame rate, quickly reducing it to undesirable levels. This means that in a situation where the material is required to reach higher speeds, it would be far better to increase the voxel sizes relative to the world, as this would not have a severe effect on frame rate. It can also be seen that the amount of material present in the simulation has very little effect on frame rate. This is likely due to the fact that each voxel in the neighbourhood is evaluated regardless of whether it contains any material, which could be a potential area to optimise for increased performance.

4.1.2 Overflow Heuristics

In this sub section I will evaluate various heuristics for handling voxel overflow.

1: Push overflowed values to the voxel above

This heuristic caused ”towers” of material to form when the pile was acted upon by sideways wind.

Number of passes	Frames per Second
1	60.04
5	60.03
9	60.03
13	60.02
17	59.50

Table 4.2: Heuristic 1 Results

2: Evenly distribute to adjacent voxels

This heuristic created piles which were more rounded, but which flattened when acted upon by sideways wind.

Number of passes	Frames per Second
1	60.03
5	60.01
9	59.80
13	56.52
17	53.31

Table 4.3: Heuristic 2 Results

3: Distribute to adjacent voxels in the row above, giving more weighting to the centre voxel

Similar effect to heuristic 1, but towers were more rounded like tall hills.

Number of passes	Frames per Second
1	60.03
5	60.02
9	58.41
13	54.84
17	51.71

Table 4.4: Heuristic 3 Results

4: Distribute evenly to adjacent voxels on the same level

Similar to heuristic 2, but flattened almost instantly when acted upon by wind.

Number of passes	Frames per Second
1	60.04
5	60.02
9	60.00
13	59.23
17	58.42

Table 4.5: Heuristic 4 Results

Comparisons

The data above shows that heuristic 1, which was used in the original implementation, is certainly the fastest, though the number of passes seemed to have little effect on improving quality and only served to exacerbate a heuristic's weaknesses. Given that all heuristics gave similar speeds at a low number of passes, the "best" heuristic would depend on the desired terrain. If one wished to have terrain which was only accelerated by gravity, heuristics 2 and 4 might be better, whereas if other sources of acceleration were to be present, heuristics 1 and 3 could be the better choice.

4.1.3 Grid Size

The effect of voxel grid size on frame rate can be seen in Table 4.6.

Grid Size NxNxN	Frames per Second
16	60.04
32	60.04
64	60.03
80	60.02
84	53.91
88	47.37
100	32.30
128	15.50

Table 4.6: Effect of voxel grid size on frame rate

Interestingly, the size of the grid seemed to have no effect on performance until around 80 to 84, at which point performance starts to drop fairly drastically. I cannot be certain why exactly this is the case, though I speculate that this pattern may be hardware specific. This implementation would need to be tested on different GPUs to know for sure.

Chapter 5

Conclusion

In summary, I have successfully implemented a real-time simulation of granular materials which incorporates an algorithm for soil advection using voxel grids which can run at a desirable 60 frames per second. The system uses 3D textures to represent the voxel grids which allows for the algorithms to be easily parallelised using OpenGL compute shaders. I have analysed various attributes of the system and attempted to explored optimisation for both quality and performance. I have also attempted to add a system to simulate wind blowing against the material.

From my experimental results, I can conclude that while there are areas where the quality of the simulation can be increased, the costs with regards to performance are far too great to consider for general use cases. It can be seen from my results that the simulation is already fairly well optimised for performance, however my analysis of overflow heuristics shows that quality must be optimised on a case by case basis. I can also conclude that there is unlikely to be a ”best” implementation for all use cases, due to the fact that quality is a subjective value and often the quality of one feature must be sacrificed for another.

Future Work

I believe that more research could be done into the overflow heuristics, particularly into using a combination of heuristics rather than just one, as I think that finding the right balance would could deliver a substantial gain in quality.

I also think that there are optimisations that can be performed on the soil advection algorithm as currently the quantity of soil in the simulation has little effect on the performance, implying that there is some inefficiency in the algorithm.

More development into adding to the simulation in general would also be of great benefit. There are many features that are still missing from the original implementation that exist in real terrains which could be added.

Bibliography

[1] Robert Bridson and Matthias Müller-Fischer. Fluid simulation: Siggraph 2007 course notesvideo files associated with this course are available from the citation page. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, pages 1–81, New York, NY, USA, 2007. ACM.

[2] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *SIGGRAPH Comput. Graph.*, 22(4):65–74, June 1988.

[3] Andrew David Geiger. *A Voxel-Based Approach to the Real-Time Simulation of Sands and Soils.* PhD thesis, Faculty of Graduate Studies and Research, University of Regina, 2015.

[4] Ryan Geiss. Generating complex procedural terrains using the gpu. *GPU gems*, 3:7–37, 2007.

[5] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, August 1987.