A Grid-Based Fluid Simulation of Hydraulic Erosion upon a Terrain

Alex Barnett (200960975)
May 2024

BSc. Computer Science with Game Engineering

Supervisor – Dr Rich Davison

Word Count: 14,220 words

ABSTRACT

When developing a game, one of the key factors is the game environment, many of which consist of a terrain similar to that which would exist in real life. The key problem here is that these kinds of environment are difficult to create and may need to be of a massive scale. This dissertation attempts to convey how I was able to create a simulation of hydraulic erosion running on a Central Processing Unit (CPU), in order to form these realistic landscapes procedurally. I aim to explain how this process works, as well as evaluate the efficacy of using such a program in a real-world scenario. Secondly, I aim to show how I significantly improved the performance of my simulation by running it instead on the Graphics Processing Unit (GPU). I will compare my two approaches using measurements I gathered and explain the key differences between them. I will cover the advantages and disadvantages of my method, concluding with an evaluation of the project and a brief outline of future steps.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Rich Davison, for supporting me throughout my project.

DECLARATION

I declare that this dissertation represents my own work except where otherwise stated.

TABLE OF CONTENTS

Abstract 1
Acknowledgements 1
Declaration 1
1 Introduction 3
1.1 Motivation 3
1.2 Aims 3
1.3 Objectives 3
1.4 Changes 4
1.5 Structure 4
2 Background 5
3 Methodology 6
3.1 Software 6
3.2 Overview 6
3.3 Ethical Considerations 19
4 Results and Evaluation 19
4.1 Results 19
4.2 Evaluation 24
5 Conclusions 27
5.1 Project Success 27
5.2 Personal Improvement 29
5.3 Future Work 30
6 References 32
7 Appendices 33
7.1 Movement Visualisations 33
7.2 Simulation Examples 34
7.3 Recording of Simulation Running 37
7.4 Project Repository 37

1 INTRODUCTION

1.1 MOTIVATION

When creating a virtual environment for a game engineering project such as a video game, simulation or visual showcase, a game developer needs to find a way to generate large-scale terrain which looks suitably realistic. You can see many examples of these realistic and highly detailed landscapes in modern games. As stated in the Eurogamer article "We asked a landscape designer to analyse The Witcher 3, Mass Effect and Dishonored" *[1]*, a suitable and well-made terrain is incredibly important for immersion and enjoyment within a game, - *"It gives the environments much more depth intrinsically linked to games' narratives, themes, stories and settings, ultimately making for better, more accurate and game-complementing virtual spaces for us to explore, experience and gallivant around in"*.

The process of creating these kinds of terrain could be done entirely by the developer using tools such as terrain painters to get hand-crafted landscapes. But this method is both time-consuming and scales poorly with larger projects. As games grow exponentially larger in scale throughout history, these environments become even more difficult to create using these techniques. Instead, many game developers are now using procedural generation for a game's landscape. This is often done by generating and distorting noise, then using this noise as a heightmap for the terrain. However, the realism can suffer from this approach. It can also be difficult to find exactly the right combination of values and attributes to use when creating these landscapes so that it is suitable for the specific game.

1.2 AIMS

For this project I am attempting to create a simulation of hydraulic erosion upon a terrain. I will build the simulation in Unity; the result of running my program should be a mesh detailing a terrain which looks realistic and aesthetically pleasing. My plan is to first create this program using C# scripts running on the CPU. I aim to then improve its efficiency by moving the workload to the GPU so that it can be ran in parallel. I will detail my progress with both the visual efficacy of the terrain as well as the overall efficiency of the program.

1.3 OBJECTIVES

- Research and evaluate fluid dynamics simulation algorithms specifically including grid-based 2D approaches such as 2D Navier-Stokes equations from key background sources to find the best approach for my specific use case.
- Initialise a custom terrain mesh using a large uniform grid with a noise-based heightmap. This will act as a starting point for my planned hydraulic simulation. It is also important that this terrain is truly random and many different terrain heights can be generated to prove my erosion simulation works on a range of landscapes.
- Implement a CPU based hydraulic erosion simulation wherein each individual droplet travels down the terrain in a path based on realistic fluid dynamics, and the erosion/deposition changes the height of each point the drop travels over.
- Translate this code into a compute shader program in Unity which runs in the same way as the CPU code but in parallel on the GPU for greater efficiency.

- Measure and evaluate both the overall realism and effectiveness of the project, as well as a comparison between the CPU and GPU approaches in terms of efficiency and performance.

1.4 CHANGES

During the course of my project, there were very few deviations from my original plan as outlined in my project proposal. I completed my main objectives in the same order as planned and within the time requirements, so there was relatively little needing to be cut down or changed.

The one change that was made, was a decision to move away from the more complex fluid dynamics equations such as Navier-Stokes, and to instead emphasise a grid-based approach. When planning my simulation, I had intended to integrate these fluid equations into how the drops would move. They would create a relatively accurate simulation of gravity, velocity, and inertia. However, upon further inspection, I learned that these equations are mainly used for particle-based fluid simulations and generally do not translate well to a grid approach. For example, many of the equations relate to pressure and density of the fluid. As I am only simulating one drop at a time, this means that interactions such as those between drops do not need to be calculated. This is also why many of the traditional particle based equations are not needed for my project. The inclusion of these equations would have been mostly redundant, as well as slowing down my simulation due to excess calculations, and add unnecessary complexity.

I therefore decided to make my own simplified fluid dynamics equations which were entirely grid-based and only as complex as necessary. This involved calculating velocity and direction magnitude for each drop using my own equations, as well as a simplified version of inertia and friction. I will cover this in more detail during the methodology section relating to droplet movement. While these equations are far less accurate to real life, the actual simulation of the drops does not need to be entirely accurate for my project. This is because the accurate movement of the drops is not my goal, the only product I am assessing in my project is my final terrain. I found that these equations were accurate enough to translate to a reasonably realistic and detailed terrain, without adding excess processing time.

1.5 STRUCTURE

Having covered why this project is important and what my aims are for it, I will move on to the rest of my dissertation. This will involve first, an overview of my background resources, with a brief introduction to each source and how I have used them during the course of my project. I will then move onto the largest section which is my methodology. Here I will describe each step of my project in detail, broken down into key sections according to my objectives. I will then display and evaluate my results in terms of both efficiency and aesthetics, as well as cover the key advantages and disadvantages of my method. Finally, I will conclude with an overview of the project's success, my own personal improvements and possible future steps.

2 BACKGROUND

During the course of the project, I referred to a range of papers on relevant topics for ideas or guidelines. Some of these papers were directly relevant to the creation of a hydraulic erosion simulation, while others focused on specific topics related to my project, such as noise generation and GPU based simulations. I referred mostly to papers included in my proposal, as these provided a comprehensive and diverse range of sources. Below I summarise each source used, as well as a brief explanation of where I used them in my project. I will explore these use-cases in more detail throughout the methodology section.

"An Erosion Model Based on Velocity Field for the Visual Simulation of Mountain Scenery" [2]
The paper introduced an early implementation of erosion simulation in computer graphics, shaping subsequent terrain modelling research. By introducing a model based on velocity fields to simulate erosion processes, the paper contributed to the creation of more realistic virtual landscapes, influencing the development of various future applications. As this paper is the oldest source I am using, some of these methods proposed here have been superseded by newer research. The system used for this erosion simulation uses velocity fields to simulate the water flow. As I am using a grid-based method, many of the mathematical equations were not relevant for me, however I did take inspiration from the motion calculations stated in this paper. To calculate the approximate motion of the velocity field, the paper describes an equation which uses the position, velocity, acceleration, and delta timestep of the water. For my movement I must decide which grid point would be most desired to move to, and the equation for this was derived from modifying the papers equation into one more suitable for my uses.

"Visual Simulation of Hydraulic Erosion" [3]
The paper presents a simulation method specifically focused on hydraulic erosion, emphasizing the visual representation of erosion processes caused by flowing water. The method used for erosion in this paper involves simulating layers of water on top of the existing terrain, which can form from either water sources or affect larger areas to simulate rain. The rain simulation specifically was useful for my project as I am simulating water in a similar way. This involves randomly distributing my rain drops over the entire terrain, similar to the method used in this paper. The paper also describes methods to compare the height neighbouring vertices. This was useful as a reference for my own drop movement which implements a similar technique.

"Fast Hydraulic Erosion Simulation and Visualization on GPU" [4]
The paper presents an approach to simulating hydraulic erosion using the computational power of GPUs. It introduces efficient algorithms tailored for GPU implementation, enabling a real-time visualization of erosion with high fidelity and performance. The paper uses a similar velocity field based method to some of my other references, simulating flow with a shallow-water modal. This is not the same approach as I took so again the equations are not all used, however with this simulation running on the GPU, it was useful to refer to when creating my own GPU based program. It was especially useful for comparing with my results for GPU compute time, as the performance of the paper's program is well detailed.

"Procedural Terrain Generation Using Perlin Noise" [5]
This paper explores the application of Perlin noise, a versatile and widely used procedural texture generation technique, in this case specifically for creating realistic terrain in computer graphics. My first objective for my project was to create the initial terrain to simulate the erosion

upon. For this I used Perlin noise to create the random heightmap of my terrain. During this process I referred to the paper for understanding how this would be achieved in a higher level context. As my project was not focused on the creation of this noise, I used an external Library called FastNoiseLite for the Perlin noise. This meant that the more technical aspects of creating the noise as covered in the paper were not necessary for me to fully understand, however there were still some parts I found useful. In particular, I followed similar steps in setting up my terrain to use the noise, storing the noise in a 2D array and using this as the heightmap set to the y coordinates of my terrain mesh.

"Stable Fluids" [6]
The paper introduces a method to solve the Navier-Stokes equations in a stable numerical way. This allows the simulation of fluid-like behaviour as it outlines a way to process this behaviour in code. In my project this was probably the paper I referred to the least, although I had planned in my proposal to use it. This is because the paper focuses on the Navier-Stokes fluid-dynamic equations, which I had planned to implement into my project. However, as discussed in my changes section, I decided to deviate from this plan and instead focus on my grid-based, more efficient approach. I made this decision after going through the paper to see how the methods and equations would translate into my approach, and finding that there were too many key differences which would make any meaningful translation of these equations impossible. One of these reasons is the way my simulation handles time-steps in comparison to most particle-based systems, such as the one outlined in this paper. The method used for these particle-based systems includes using small discrete timesteps and translating the movement of the water over this time. However, for my approach I did not need to consider the timeframe, nor the distance travelled, as I simply moved the drop one grid space each iteration. Between this key difference, and those outlined in my changes section, it makes many of the equations used in this paper not viable for my own use.

3 METHODOLOGY

3.1 SOFTWARE

- Unity 2022.3.17f1
- JetBrains Rider 2023.2.3
- FastNoiseLite

3.2 OVERVIEW

Initialising a Random Mesh
Beginning my project, I first created a grid of Vector3 points, where x and z values are the grid points evenly spaced according to the given mesh size, the y values equate to the height of the given point. All points are created along the x file from left to right in order, then down one z file, repeating until all points are created. This gives an array of ordered points of which I will be able to work with throughout my project. Triangles are created from the indexes of these points in the Vector3 array, looping over these points to create triangles using counterclockwise motion as to display the mesh facing upwards. This gives a flat mesh of a given number of points all evenly spaced on a grid.

I then made a 2D texture, created as a heightmap for the initial random terrain using a method called fractal noise. This method, as described in *"Procedural Terrain Generation Using Perlin Noise" [5]*, uses multiple scales of Perlin noise. I used the in-built Unity function to create these layers, each accounting for decreasing amplitudes as the scale of the noise increases, to create a complex texture with some detail. The texture is then sampled from to raise the height of each point on my mesh according to the noise value at each point. This however, created some issues with the terrain looking quite blocky when the size of the mesh became too large for the resolution of my noise. I therefore created a smoothing pass which checks the height of all neighbouring points and sets each point to the average of its neighbours. This needed to be run multiple times to properly smooth the terrain. The problem with this approach was that it made the initial generation of my terrain rather slow, especially at larger scales, as more smoothing passes had to be ran. It also meant that my terrain was not as detailed as it could be due to the aforementioned insufficient resolution of my noise.

To fix the resolution problem I decided to use an external library for sampling the noise, called FastNoiseLite by Jordan Peck. I felt it was necessary to use this library as it provided noise of a significantly higher resolution. As my project focuses mainly on the simulation I run on the initial terrain and not the creation of it, I was not concerned with creating this noise myself, as that is not within the scope of my dissertation.

For use with this library, I switched from the 2D texture approach to instead use a 2D float array for simplicity. I then used the same fractal noise method to create the heightmap and set the y value of each of my grid points to be at the height of the sampled noise. Because this noise was a higher resolution, the terrain was not at all blocky even when using a very large scale mesh. This meant that it was now unnecessary to run any smoothing passes at all, significantly decreasing the time it takes to create my initial mesh.

Simulating Droplet Movement

The first step in creating my simulation was to generate a random index for a point on the mesh, the drop starts from the corresponding point in the mesh array. This depicts the falling of rain randomly spread across the terrain, similar to how it is simulated in *"Visual Simulation of Hydraulic Erosion" [3]*. The drop is a Class of which contains all the needed information as it is simulated, such as the current index and position of the drop. To simulate the drop moving, it needs to flow downhill following the easiest path, similar to the movement of real water. To calculate this, it checks the height of all neighbouring points and moves to the lowest neighbour, providing it is lower than its current point. If none of the neighbours meet this criteria, then the droplet has nowhere left to move and so its simulation is finished. This is run recursively to move the droplet travel down the terrain. Its speed at each point can be calculated using the vertical distance travelled at each step of movement.

In this simulation there are no timesteps, and distance is always by exactly one grid point. I do not use the time of the drop or simulation for any of my calculations. This way I am able to avoid the complexity and compute time of many particle-based systems which need to run on very small discrete timesteps, such as Unity's in-built physics updates. I can instead run my entire simulation at once much more easily and decrease processing speed significantly. This movement however was very basic, only following the slope of the hills directly downwards and did not account for the velocity of the drop when making its movements. However, it is important that the inertia of the drop has an effect on its movement in order for it to look realistic.

I therefore added this inertia system to the simulation, inspired by those used in other papers such as *"An erosion model based on velocity fields for the visual simulation of mountain scenery" [2]*, which was made to encourage the drop to continue in the direction it was already moving, even going uphill if it has accumulated enough total speed in that direction. First, I added a velocity variable to the drop class which is stored as a Vector3. This converts the y change from the drop's movement into an x and y magnitude according to the direction travelled. I then use this when calculating the next path. Instead of checking the height of each neighbouring point, I calculate an adjusted height which has been offset by the velocity in the relevant direction. This encourages the drop to keep moving in similar directions if its speed is high, as the adjusted height of points in that direction will be lower. It also allows for some movements which are to a point of increased height if the adjusted height is still lower than any other point. I also added a variable called friction to act as a dampening multiplier, this ensures the droplet cannot continue to build speed indefinitely.

$$Current\ Velocity = Previous\ Velocity \cdot Friction$$

The value of this variable plays a large role in the end result of the simulation. If the value is close to 1, the drop will have very little friction. This means that the path the drop travels and erodes upon will be very straight and can easily climb up elevated terrain if it has built up some speed. If the value is very low however, such as 0.5, the drop will experience a very high friction. It will struggle to build up any speed at all and will curve down the mountain following the lowest path very closely. As well as this, it will not be able to continue its path for very long once it reaches a slope upwards and is very likely to slow down and get stuck in any divot in the terrain. A visual representation of this movement system can be seen in Appendices 7.1: Movement Visualisations

Simulating Droplet Erosion

Using the movement of droplets, I now needed to make a system to erode some of the landscape. The first version of this simply decreases the height of the terrain at the point the drop moved on, according to the speed at which the droplet was moving and the change in height it encountered. It also made any upwards traversals add to the height, and at the end of the simulation a portion of the total sediment eroded is added to the terrain to simulate sediment deposition. This system was more of a proof of concept however, as it did not look particularly realistic. The cause of this lack of realism, was that sediment could be created or destroyed, and drops could erode an unlimited amount of sediment and carry all of it. It also caused many holes to appear in the terrain where the drops were eroding a small divot, which continuously expanded as more erosion was happening than deposition.

I therefore implemented a new version of this erosion, using a volume for each drop. This volume starts at a set rate, each step decreasing by a proportion of its volume to simulate evaporation. As the drop moves downhill, the maximum sediment it can hold is calculated using the volume of the drop multiplied by the absolute value of its current velocity. It can only erode points up until it is fully saturated with sediment, and once this happens it will deposit a portion of the overall sediment it has accumulated. It will also deposit a proportion of its sediment when moving uphill, and once it has finished its travel it will deposit the remaining sediment it has picked up. This system means that no sediment is ever created or destroyed, only ever moved from one spot on the terrain to another. This fixes the issue of holes appearing in the terrain.

$$M = Directional\ Magnitude\ (x, y, z),$$
$$CP = Current\ Position(x, y, z), LP = Last\ Position(x, y, z)$$
$$A = Change\ Amount, VL = Volume, S = Current\ Sediment$$

$$V = Velocity = |M|.x + |M|.z$$
$$D = Distance\ Change = CP.y - LP.y$$
$$X = Maximum\ Sediment = VL \cdot V$$

$$Sediment\ Deposit:\ \ (S < X) \rightarrow \max(0, \min(D \cdot A \cdot V, X - S)), \qquad (S \geq X) \rightarrow -\frac{S}{\max(1, V)}$$

I used the above calculations to derive how much sediment would be deposited or eroded from the terrain at a given step. The first side of the sediment deposit equation is for when the current sediment being carried by the drop is less than the maximum amount of sediment which the drop can carry. In this case the result will be positive, which will lead to the terrain being eroded. The amount to be eroded is calculated using the change in y level of the step, the change amount variable which controls how much of an impact each drop has, and the current velocity of the drop. This amount cannot be less than 0, nor can it be greater than the maximum sediment the drop can carry minus the current sediment being carried, as that way the drop would be carrying more sediment than the current maximum allowed.

As for the second half of this equation, where the current sediment is greater or equal to its maximum allowed sediment, this is always negative, so the drops will deposit sediment. The amount to be deposited is a fraction of its current carried sediment, meaning it can never deposit more than it is carrying. This sediment is divided by the current velocity, so that the faster the drop is moving, the less sediment it will lose. This means the drop will deposit more once it slows down. It also never divides by less than one as this would cause more sediment to be deposited than the drop currently holds. Overall, this equation follows the established rules of my simulation, where the sediment cannot be created or destroyed, only moved. It also ensures that the erosion and deposition look realistic due to the velocity of the drop always affecting the amount of change in the sediment.

The next step was implementing a radius for the drop. As the terrain can be very large, when the erosion is only impacting one point at a time it creates many thin slices in the terrain rather than a smooth erosion. To fix this I created a radius for the drop wherein the drop will erode/deposit onto surrounding points. With a radius of 1 the drop effects the main central point and its 8 immediate neighbours, with a radius of 2 the drop will affect 25 points as it also impacts these neighbour's neighbours.

$$Number\ of\ Impacted\ Points = ((Radius \cdot 2) + 1)^2$$

```
private List<int> FindNeighbourPoints(int pointIndex)
    {
        List<int> neighbours = new List<int>(Mathf.RoundToInt(Mathf.Pow((2 * radius) + 1, 2)));
        for (int x = 0; x < (radius * 2) + 1; x++)
        {
            for (int y = 0; y < (radius * 2) + 1; y++)
            {
                neighbours.Add(pointIndex-(radius-y)+(rowLegnth*(radius-x)));
            }
        }
        return neighbours;
    }
```

Finding these points around the drop takes some specific calls due to the order of the points in my array. The points are ordered from left to right along the x column, then down a row on the z column repeating for all the points. This means that to find the direct neighbours you can use the position of the current point in the array, along with the length of the row to find the points surrounding. You can see how I have done this in the code extract above, which returns the list of all points within the specified radius around the main index.

The movement and erosion calculations for these drops are still based on the one point in the centre of the radius, however the sediment deposition/erosion effects all the points within the radius. Once this was implemented, I created a system whereby when simulating the erosion, it has multiple passes. It starts with a larger radius such as 4 for the macro-level landscape changes, then the radius moves down by 1 and the simulation is run again on the same terrain until the radius is 0. This allows for both the larger scale changes while also adding the necessary detail that is created at these lower radius levels. Each of these layers have their own variables which need to be changed according to the radius so that each layer has the right effect on the terrain.

To complete my CPU based approach, the last process was to change the variables to work correctly together to create a realistic looking terrain. As there are many variables to experiment with, such as volume, friction and evaporation rates, this process can take some time. Especially when considering that each of these variables can be independent for each layer and may need to differ for each radius. As my testing simulation runs consecutively for 5 layers, with each layer impacting the erosion of the next, this means that every variable for every layer can have a large effect on the final terrain. Getting all of these variables to work together took a lot of testing and experimenting until I was satisfied with the final result.

I will cover this aspect in more detail during my analysis section, as this specificity can cause the program to be difficult to run without getting an undesirable result. This is especially prevalent when changing factors such as terrain size, noise scale, or number of drops simulated, as the processes of fine-tuning variables must be completed again.

GPU Implementation

Once I was finished with my creation of the CPU program, I moved onto my GPU based implementation. This was written in HLSL to make use of Unity's compute shaders which can run on many threads at once for parallelisation. I do not have as much experience in HLSL as I do with C# for the CPU side, so this provided additional challenges. The overall goal here was to translate the code I had already written into the new project, to save the work of completely restarting the project as well as giving me the ability to do a direct comparison between each approach. This is because if each program worked in an altogether different manner, it would be futile to compare the two approaches in terms of efficiency.

To set up my GPU system I first created a compute shader and worked on sending and retrieving data from the shader. I sent the heightmap of the terrain to a float3 structured buffer and set the variables of the simulation to floats within the shader such as the radius and volume of the drop to be simulated. At first, I had planned to send the heightmap data back to the CPU using another structured buffer. However, when running this simulation on the 32 X 32 threads I would have retrieved 1024 different heightmaps, which I would have needed to combine together and been very inefficient. My solution for this was to instead send back instead only two variables of data for each change, the index of the point changed as well as the value of change. I would have two structured buffers, one to hold the indexes and another for the

change amounts, each thread could then append onto these buffers whenever a point was changed. The CPU would then have a much smaller task of reading through this buffer and changing each point on the original heightmap accordingly.

However, while this was significantly more efficient, it still relied too heavily on the CPU, causing a bottleneck in my simulation. It also would have been inefficient to constantly send and retrieve data between the CPU and GPU every time a group of threads ran. I found that the best method for retrieving the heightmap was to have each thread save its changes back into the structured buffer it received the data on. As the data can both be viewed and edited, this made it possible to avoid having to send the data to the CPU at all before the mesh had to be updated at the end of the simulation. Because the buffer which the threads read from was also where the changes were saved to, this meant that the next group of threads could simply read this data back and it would be the partially eroded heightmap of which they could continue to simulate on. This removed all communication between the CPU and GPU unless the variables had to be changes, such as when changing the layer being simulated, dramatically increasing the performance, and removing both the communications and CPU processing bottleneck.

I had some difficulties directly translating the erosion code due to a few key differences in the languages, as well as the overall manner in which GPU code is be run. I could no longer simulate my droplet using recursion, as this is not possible in HLSL due to how GPU's compute. To solve this problem, I instead moved the main code loop to a while-true loop, with a catch to stop the simulation once a drop had completed its run. However, this method also did not run, and I found that this problem is caused due to the GPU threads needing to know how many computations will be run beforehand, as to allocate the correct amount of memory space needed. I therefore created a way to bypass this issue by running my main code as a for-loop of a fixed large length (I used 500). This loop would never end this way however, as I exited the loop once the droplet had finished its path and before the loop had run the 500 iterations. This caused slightly more memory space to be allocated than may have been required, but overall fixed the issues with code refusing to run and was the only way I found to pass this problem.

I found this particular issue to be very prevalent in the time I spent creating the GPU program. This is because it not only denies the use of recursion and while-true loops, but also prevents variable length for-loops from running. Many of these were used throughout the project, for example looping through an array or list, which is dependent on the size of this variable. I solved these in a similar way to the recursion by using large fixed-length for-loops which I would break out of once the required iterations had been reached. In a similar way, arrays created in HLSL could not be of a variable length, these had to be hard-coded values. For some arrays such as the one holding the radius points, these had to be set as the maximum size needed for my largest radius at all times, even at lower radii when they were larger than necessary. This is again a small but unavoidable memory increase due to the nature of my GPU code.

My next GPU based issue was due to how randomness is handled. When a drop is simulated, it must start at a completely random point on the mesh, represented by a single integer between 0 and the size of my heightmap, to act as the starting index. For my CPU code in C#, I used the in-built Random.Range function to choose this random index. However, on the GPU there exists no random function which I could utilise. To solve this, I could simply send a random integer from the CPU for every simulated drop, but this would be extremely inefficient and bottleneck the program within the constraints of this data transfer. Instead, I needed to devise a way in which I could create randomness on the GPU with the least amount of communication to the CPU required.

To do this I utilised the current thread count to create a pseudo-random hash function which would create a random integer within the required range which would be independent for each thread. I still needed to send some random seed for this hash to differ for each group of threads, however this one seed could now be used for all 32 X 32 threads before needing to be updated. This massively reduces the amount of communication required between the CPU and GPU, as well as decreasing the CPU workload as many times less random seeds needed to be generated. The class I created for this random hash utilises the randomness of the sin function with large primes, offset by the thread count and the initial seed. This leads to a sufficiently random distribution of drops both within the same group, and between each group.

Finally for the GPU program, I had to get all the variables correct for each layer. These differ from the CPU variables as there are many more drops being simulated so each drop will contribute much less in comparison, but together will create a similar, even more detailed terrain.

Parallelisation Issues

Now that the GPU program was complete, I had to consider one of the problems that running the drops in parallel could cause. Because many drops are running simultaneously, reading and writing to data at the same time, this could cause some overlaps. My compute shader runs on 32 X 32 threads, and for my calculations I assume the worst case scenario wherein all of these drops are simulated at exactly the same time. In reality they are more staggered when running, where most threads do not save data at exactly the same time as others. But for my calculations I assumed that all drops have the possibility to override data to find the maximum amount of data that could be lost.

For my calculations I also need to know how many points are changed by each drop. I ran multiple tests attempting to calculate the average number of points that each drop effects, but this is massively influenced by the value of my variables. If the evaporation or friction rate is a higher value, then the drop will be simulated for far longer. I will therefore show this calculation with a few cases of differing numbers of points being changed by each drop to evaluate this problem. My simulation currently runs on a 1024 X 1024 grid, the total points being 1,048,576 for these calculations.

$$Points \equiv 32 \cdot 32 \cdot 5 = 5120 \rightarrow e^{-5120^2/2 \cdot 1048576} = 3.726653172 \cdot 10^{-6}$$

$$Points \equiv 32 \cdot 32 \cdot 20 = 20480 \rightarrow e^{-20480^2/2 \cdot 1048576} = 1.38389652 \cdot 10^{-87}$$

$$Points \equiv 32 \cdot 32 \cdot 50 = 51200 \rightarrow e^{-51200^2/2 \cdot 1048576} = 1.35486998 \cdot 10^{-543}$$

The first calculation follows the assumption that each drop will alter 5 points on the terrain. It shows the probability that no two points will ever overlap, accounting for the birthday problem as described in the paper *"Methods for Studying Coincidences" [7]*. It shows that the chance of there being no overlaps at all in this scenario is improbable, and so the chance that there is at least one overlap is probable. The second equation is calculating using the same equation but for 20 points changed per drop. The result of this shows that the chance of there being no overlaps at all here is almost impossible. The final equation accounts for each drop altering 50 points. Here the chance of no overlaps is so minute that it should be considered statistically impossible.

These equation illustrate the importance of considering how my program deals with these overlaps, and how many there are. In my simulation, if two drops try to edit a data point to different values at the same or very similar times, it can lead to loss of data. This in itself is not a major problem, as the data is simply overwritten and does not break the simulation. If there were many overwrites however, it would cause a lot of data to be lost, causing inefficiency in the simulation. I therefore attempted to calculate how many drops would overlap on average.

To calculate the average number of drops overlapping in relation to the number of points being changed, I derived a formula myself. For this process I first assumed 5 points being changed per drop for simplicity:

$$Average\ Overlaps = \frac{5 \cdot 5}{1048576} + \frac{2 \cdot 5 \cdot 5}{1048576} + \frac{3 \cdot 5 \cdot 5}{1048576} + \cdots + \frac{n \cdot 5 \cdot 5}{1048576}$$

The above equation shows the concept of overlapping point probabilities. The first set of 5 points have no chance to overlap as there are no other sets to overlap with. In the second set each point has a 5/1048576 chance to overlap due to the 5 points in the previous set, and there are 5 points with this chance. The third set therefore has a two sets of 5 to account for so the probability is 10/1048576 for each point to overlap, for 5 points in the set. This can be continued until all sets have been calculated.

$$Number\ of\ Points\ in\ Set = P$$
$$Number\ of\ Sets = K$$
$$Range\ for\ Each\ Point = M$$
$$Average\ Overlaps = \frac{1 \cdot P^2}{M} + \frac{2 \cdot P^2}{M} + \frac{3 \cdot P^2}{M} + \cdots + \frac{K \cdot P^2}{M}$$

The equation above shows this same concept but altered to use for any values. I then simplified this equation into the summation below:

$$Average\ Overlaps = \sum_{n=0}^{K} \frac{n \cdot P^2}{M}$$

This summation, however, is not necessary to use, as the equation itself can be further simplified. As the first and last additions are equivalent to the second and second to last, it can be further simplified as follows:

$$\frac{1 \cdot P^2}{M} + \frac{K \cdot P^2}{M} \equiv \frac{2 \cdot P^2}{M} + \frac{(K-1) \cdot P^2}{M} \therefore Average\ Overlaps = \frac{(K+1) \cdot P^2}{M} \cdot \frac{K}{2}$$
$$Average\ Overlaps = \frac{(K^2 + K) \cdot P^2}{2M}$$

This equation allowed me to easily calculate the average number of drops overlapping accounting for multiple values of points changed per drop. I also calculated the percentage loss for each value of points changed.

$$Number\ of\ Sets = 1024$$
$$Range\ for\ Each\ Point = 1048576$$

$$Number\ of\ Points\ in\ Set = 5$$
$$Average\ Overlaps = \frac{(1024^2 + 1024) \cdot 5^2}{2 \cdot 1048576} = 12.51$$
$$Percentage\ Loss = \frac{12.51}{1024 \cdot 5} \cdot 100 = 0.24\%$$

$$Number\ of\ Points\ in\ Set = 20$$
$$Average\ Overlaps = \frac{(1024^2 + 1024) \cdot 20^2}{2 \cdot 1048576} = 200.19$$
$$Percentage\ Loss = \frac{200.19}{1024 \cdot 20} \cdot 100 = 0.98\%$$

$$Number\ of\ Points\ in\ Set = 50$$
$$Average\ Overlaps = \frac{(1024^2 + 1024) \cdot 50^2}{2 \cdot 1048576} = 1251.22$$
$$Percentage\ Loss = \frac{1251.22}{1024 \cdot 50} \cdot 100 = 2.44\%$$

The above equations show that as the number of points being changed by the drop increases, the percentage of points lost to overrides also increases. However, even at a high value such as 50, the percentage lost is still relatively low at 2.44%. Also, all of my calculations assume the worst case for overrides, which in reality is extremely unlikely. My calculations assume that all drops are simulated at exactly the same time on each thread, and so are attempting to save data to the buffer simultaneously to all other threads. Because each drop spends most of it's time running many calculations, it is much more likely that these threads would not all be saving the data simultaneously.

My calculations are also somewhat simplified, assuming all the points being changed are at random indexes. Due to how my simulation works, this is not the case as the indexes follow a pattern as the drop moves down the terrain. This, however, is practically impossible to calculate and account for, so I believe my calculations are as accurate as they can be.

Overall, having calculated a reasonably accurate worst case average for point changes lost to data overrides, I am not concerned about this issue. This is because the impact is not large enough to drastically decrease the performance of my simulation.

Variables

As mentioned previously, my simulation has many variables all of which have an effect on the final terrain. While this can be problematic as each of these need to be set correctly to work with each other, it is also a benefit as it allows for a lot of customisation in the terrain. Here I will cover some of the most impactful variables, with examples of how changing these effects the terrain.

For these demonstrations, all other variables will remain unchanged, in order to properly illustrate the effect of each variable. It is important to note therefore, that there is much more customisation available for the actual simulation, as changing multiple variables at once can have different effects. However, due to the massive range of options this allows, it would be

impossible to fully demonstrate all of these combinations of variable changes. I will also not be using my layering technique for this as each of the layers have differing variables.

My demonstrations will all be running on a 1024 X 1024 terrain with the same seed. They will run for 500 repeats, each simulating 32 X 32 X 32 drops for a total of 16,384,000 drops, which takes around 1,400Ms to simulate. This is a reasonable test for only one layer, with 5 of these layers each with different variables, it would create a very detailed terrain. For my tests these are the values of all variables, unless it is the variable being highlighted; Friction Rate: 0.95, Radius: 4, Volume: 0.05, Evaporation Rate: 0.8.

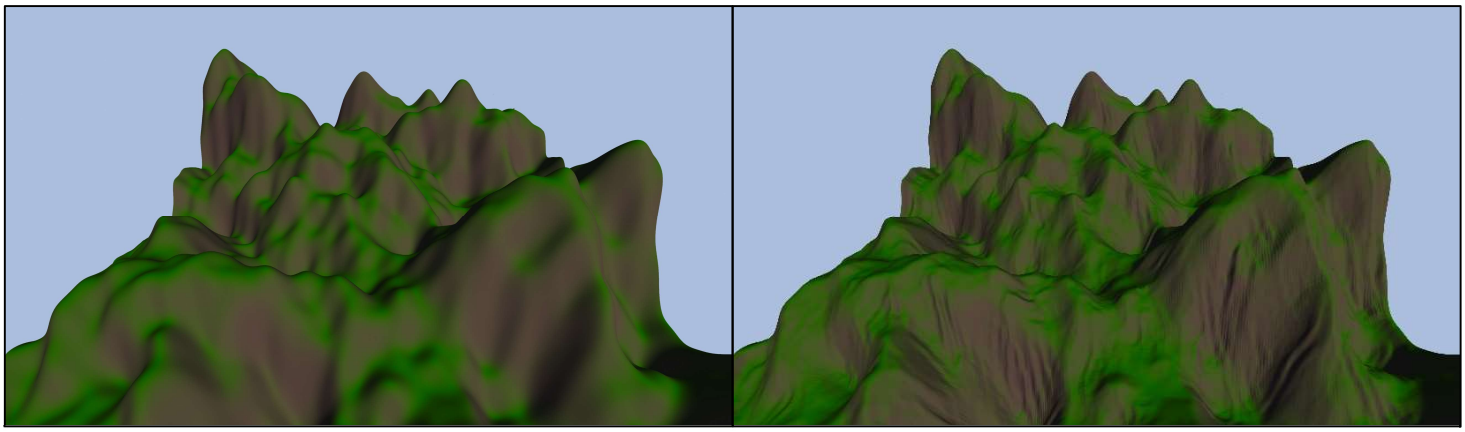


Figure 1: Simulation running with default variables.

Figure 1 shows the result of my demonstration using all the default variables for my test. As the test is only one layer, it lacks some detail in comparison to terrains which use many layers of differing radii. As it is simulating with a relatively large radius, the changes are impacting a larger area to a greater extent, but without as much detail. I chose to demonstrate these variables with a large radius as to show the greatest extent of change that the variables can make.

Friction Rate

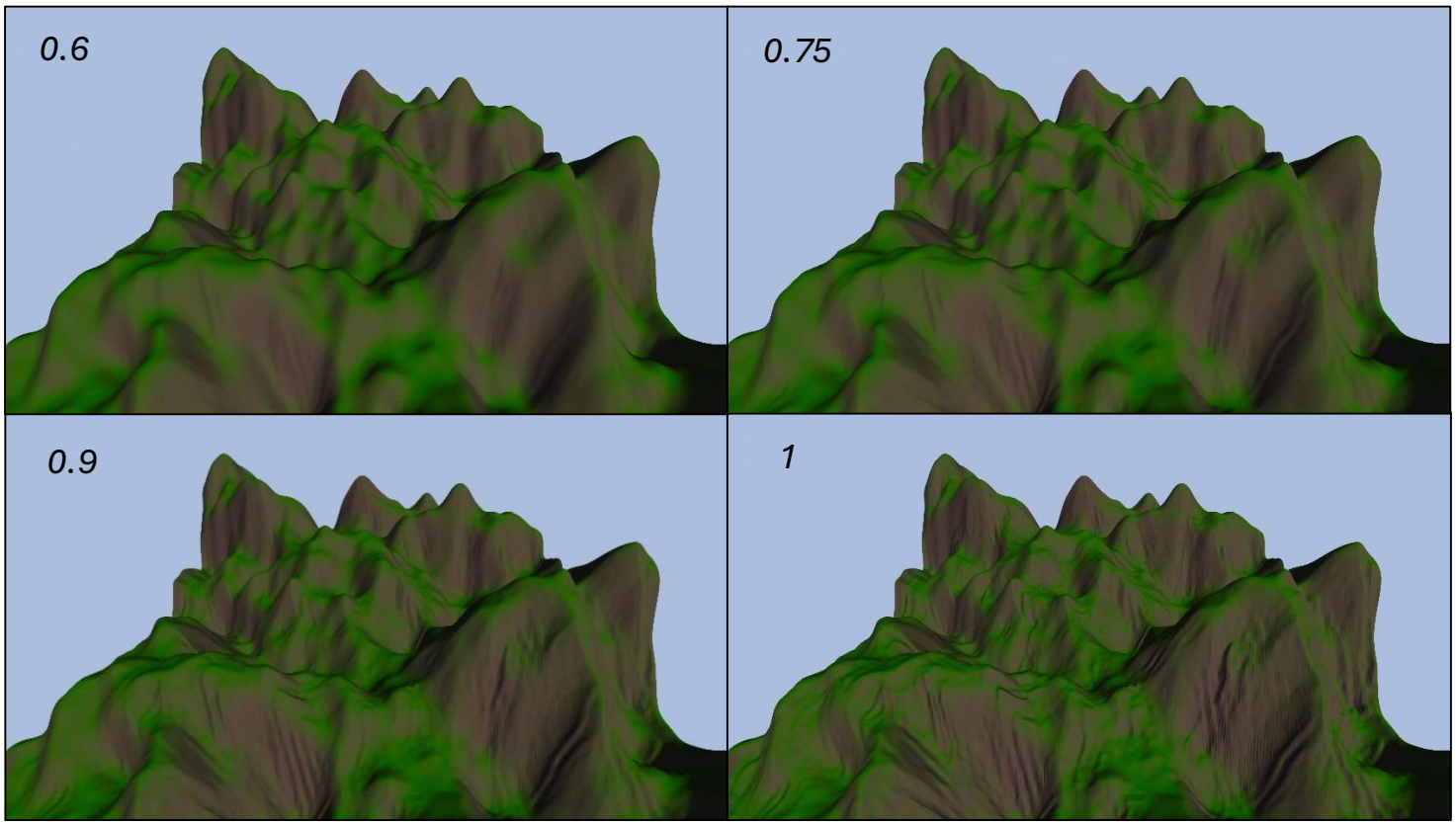


Figure 2: Simulation running with various friction rates.

Figure 2 shows the effect that friction rate has on the simulation. As this is a dampening multiplier, lower values slow down the drops more. This means that they cannot build up as much speed, and are more likely to be stuck in divots and valleys, ending their simulation. Because of this, the most prevalent change is that lower friction values decrease the amount of erosion and deposition. It also effects the path of which the drops will travel. Higher rates means that drops will gain speed and thereby be able to climb uphill when they have enough inertia.

Radius

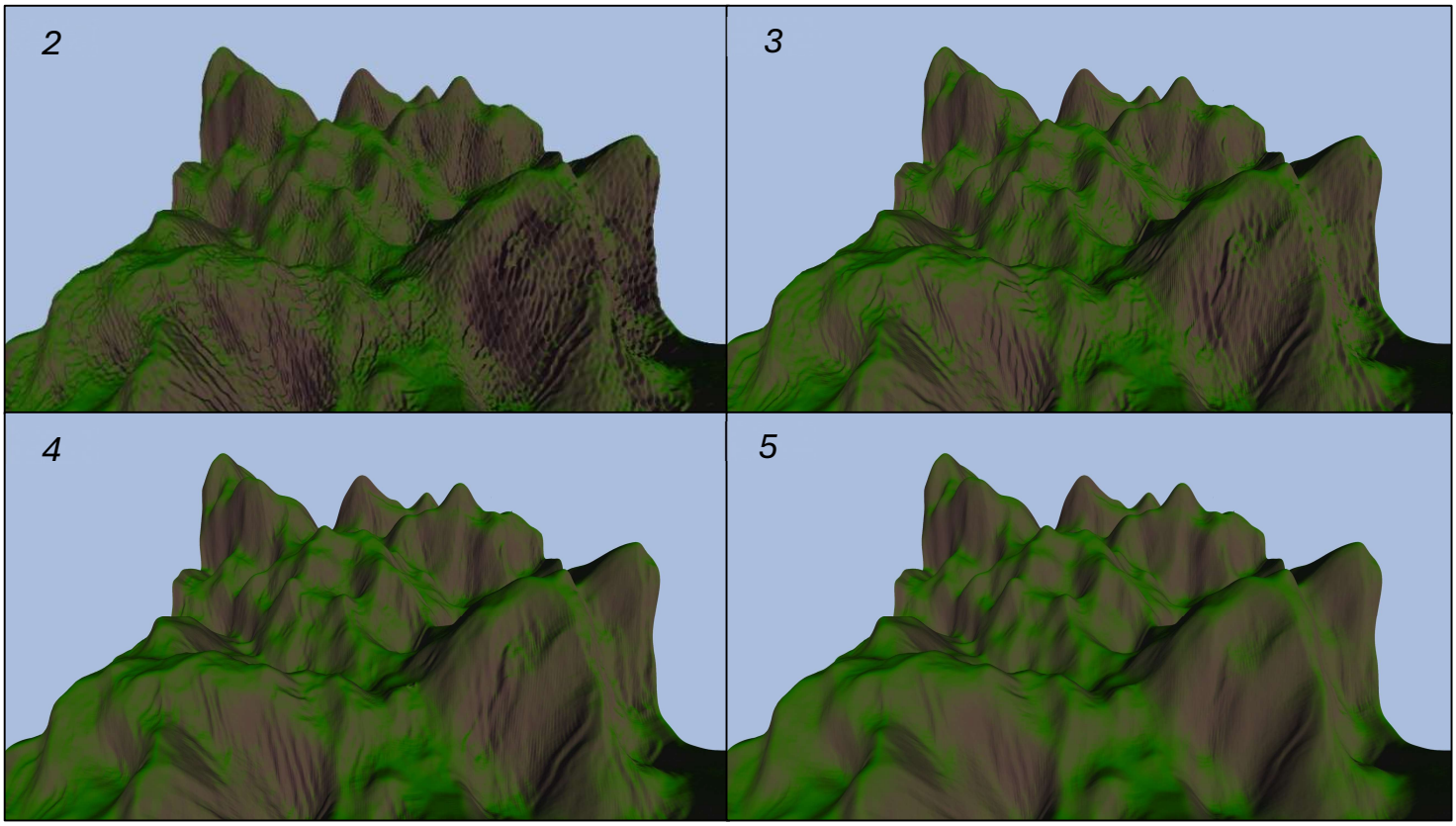


Figure 3: Simulation running with various radii.

Figure 3 shows the effect that the radii of the drops have on the simulation. In the simulations with lower radii such as 2, there are many smaller holes forming, wherein the larger radii have a smoother effect on the terrain, over a larger area. These simulations also demonstrate how changing some variables can make the simulation much less realistic, such as with these lower radii. As the default variables were tuned for a radius of 4, lowering only the radius leads to unsatisfactory effects. This is because at a lower radius the erosion and deposition are much more concentrated. The radius of the drop also has the biggest impact on performance out of all the variables. This is because a larger radius means that more points are changed during erosion and deposition, which takes significantly more compute time.

Volume

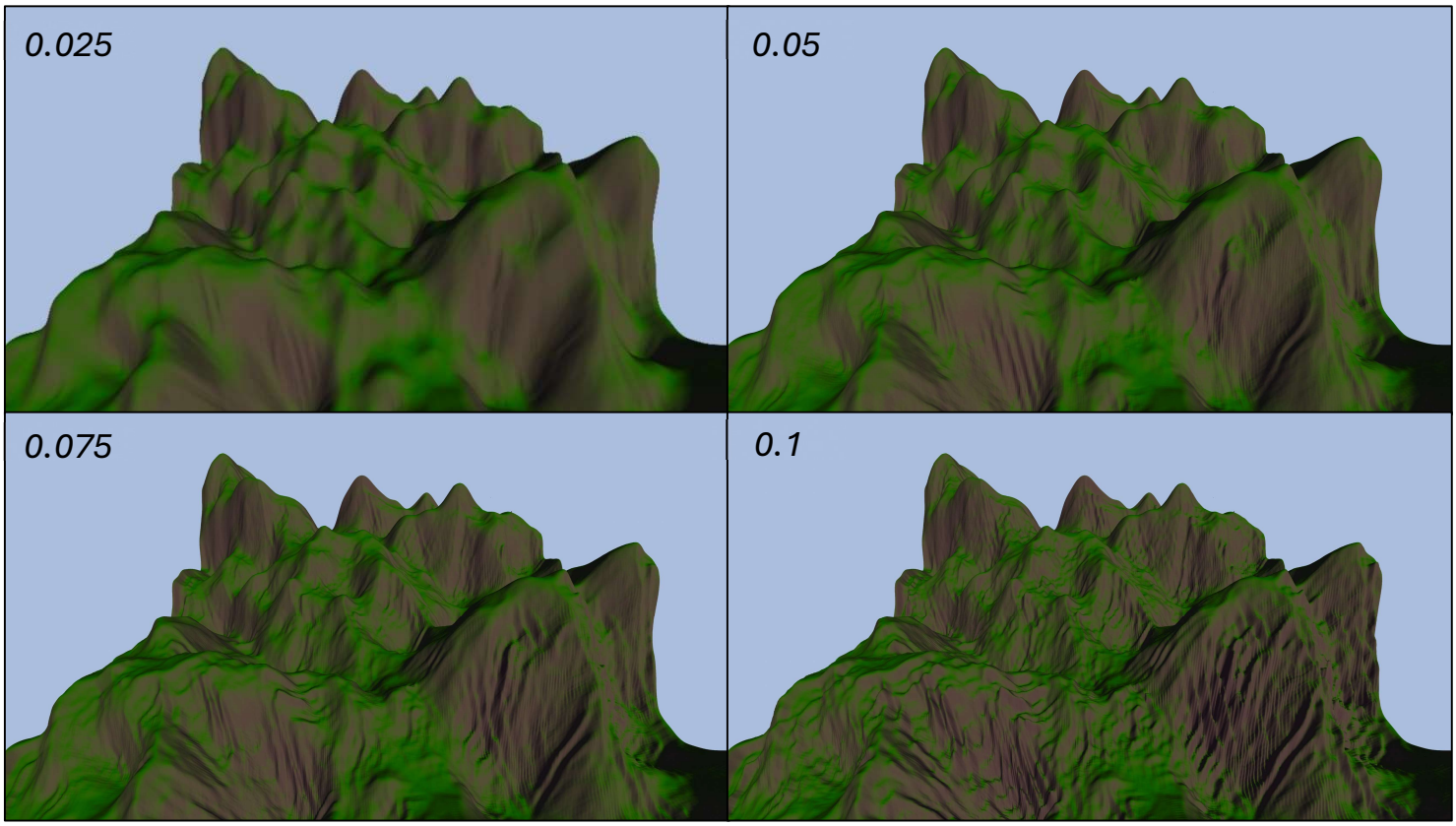


Figure 4: Simulation running with various volumes.

Figure 4 shows the how different volumes for the drops impacts upon the terrain. The volume of a drop directly affects how much it can erode and deposit, so it is clear that as the volume increases, as does the rate of erosion. It also causes the drops to be simulated for longer, as they will take longer to evaporate. This leads to slightly higher processing time, and generally longer features such as ridges throughout the terrain.

Evaporation Rate

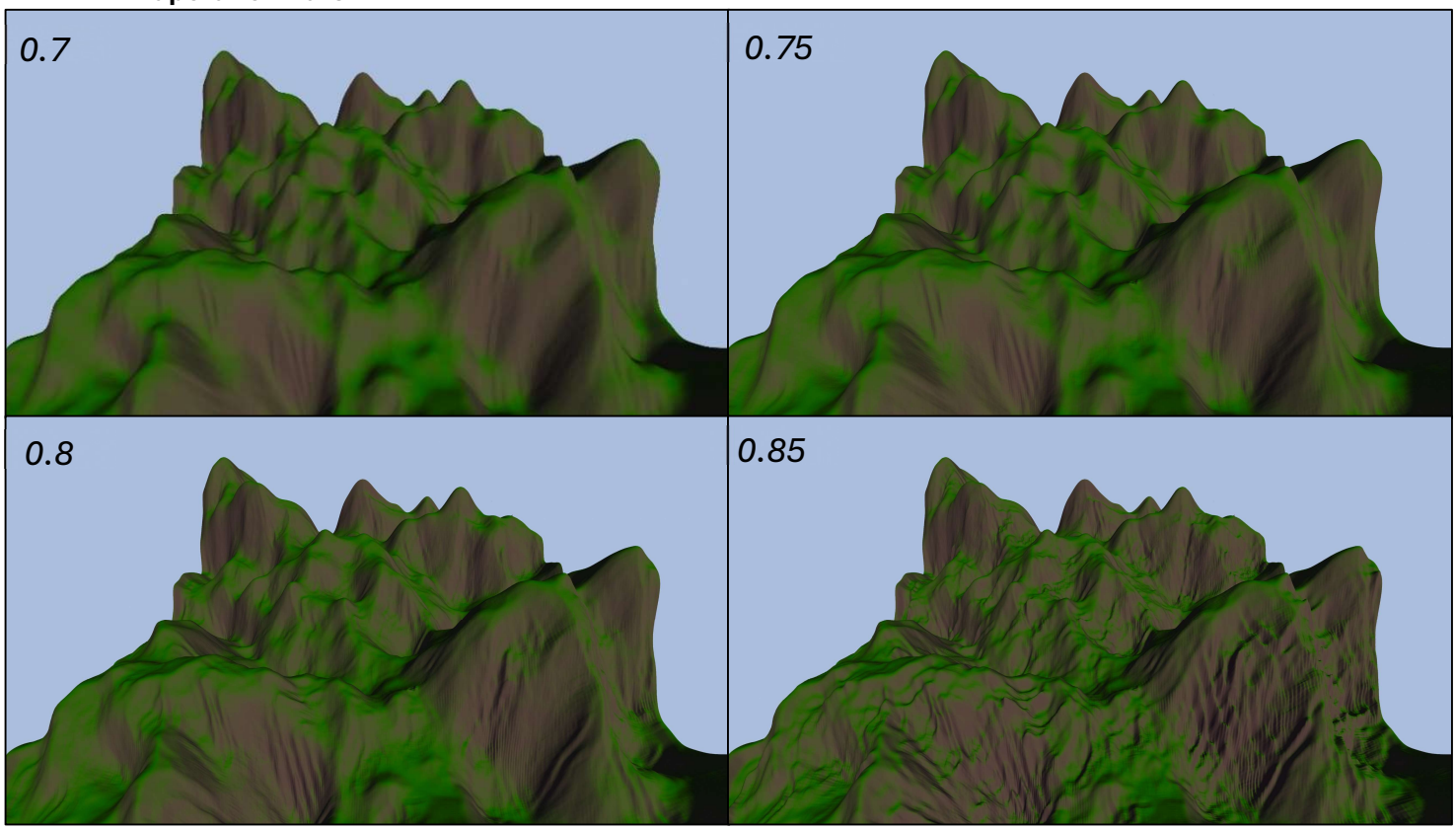


Figure 5: Simulation running with various evaporation rates.

Figure 5 shows how different evaporation rates have an impact on the terrain. This is relatively similar to volume in terms of the effect it has on the terrain. This is because the evaporation rate impacts the volume of the drop. This, like the friction rate, is a dampening multiplier, so lower rates mean drops take longer to evaporate and maintain their volume for longer. At lower rates the drops quickly evaporate away, having less of an impact on the terrain as they are simulated for less time, and have a lower volume so cannot erode as much.

Reviewing all of these variables gives an insight into why tuning the variables in the simulation can be so difficult. Firstly, it's clear that some changes can have a massively negative impact if they are not properly set. This is shown clearly in the figure 5 where all the evaporation rate values are relatively similar, but the value is too low at 0.7 and too high at 0.85. Changing variables can unintentionally have this effect especially if the user is not familiar with the change each variable causes, and the level of sensitivity for each.

Another issue that becomes apparent when viewing these variable changes, is that many of them have a similar visual effect on the terrain. For example, raising the volume too high has a very similar effect to raising the evaporation rate too high. This can make it extremely difficult to determine which variable is causing a certain problem when tweaking the variables of a terrain. Because of this, properly setting all of these variables requires a large amount of testing and experimenting. Adding to this, if there is an issue with the simulation, layering makes this problem exponentially harder to find. This is because not only do you need to find the correct variable that is causing the issue, but it must also be the at the correct layer that the problem is on. It is clear that the precision needed for these variables is one of the more problematic issues for my method, and I will cover this in further detail in the evaluation section.

Performance Testing

Once my program was complete, I needed to evaluate the performance differences between my GPU and CPU approach. To do this I conducted many tests at numerous numbers of drops, measuring the time in Ms in which the programs took to complete. For each number of drops, I tested both programs five times, resulting in a single average time taken. Each of the five tests were on different random seeds, but both of the programs were always tested on the same seed as each other. All tests were also run on 1024 X 1024 sized terrains.

As the number of drops began to increase, I noted that the limits of the CPU and GPU programs would not be able to be tested at the same time. I will discuss this further in my evaluation, however the main problem here was the disparity between the performance of each approach. The CPU program simply could not be run for the same number of drops as the GPU program needed to be tested at, the timescale for the CPU running these same tests would be impractical. I thus tested both the CPU and GPU programs with increasing numbers of drops until they both took at least 50,000 Ms or more to complete. This should hit the upper limit of drops which each program could reasonably simulate and be tested for without taking substantial amounts of time.

3.3 Ethical Considerations

My project did not raise any ethical concerns during my planning, and has not changed since then in any meaningful way in terms of ethics. Thereby the research satisfies all of Newcastle University's ethical expectations and follows the University's code of good practice in research.

4 Results and Evaluation

4.1 Results

There are two main measures to evaluate the results of my dissertation: performance and aesthetics. I will cover each of these separately.

Performance

The most quantifiable evaluation of my results is the overall performance and efficiency of my programs. To properly evaluate performance, I have conducted a large number of tests on my simulation. Each of these involve running my simulation for a set number of drops on a random terrain. To keep the measurements as comparable as possible, all of the variables of the test are the same, except for the drop count. This includes terrain size, radius of the drops, and evaporation rate, all of which could have a large impact on the speed of my program. For example, a drop with a higher radius takes more time to compute as more indexes get changed when eroding or depositing, and a lower evaporation rate means that the drop will take longer to simulate as it will not evaporate away as quickly.

I also used the same random seed for simulating on both the GPU and CPU programs when the drop count is covered for both. This allows for a more direct comparison between the two. All of these tests run with five layers of different radii, as that is the way my program creates the best terrain and so how it would likely be used in a real-world scenario.

I ran the tests five times each with different seeds for every number of drops, and the graphs use the mean average of these tests to mitigate against outliers.

It is important to cover some of the drawbacks for my tests, most notably that the results can vary at lower drop counts. This is because if the entire simulation can run very quickly, say around 50Ms, the variance in these results is mostly caused by random external factors within my computer such as background tasks. This is why I have focused on testing at much higher levels of drops to properly stress my programs and get more accurate results, where a few added milliseconds from these factors make very little difference to the end result. This test is only timing the simulation step for calculating my erosion, not including the terrain initialisation nor changing the mesh to display the erosion changes, as to minimise the external factors in the testing.

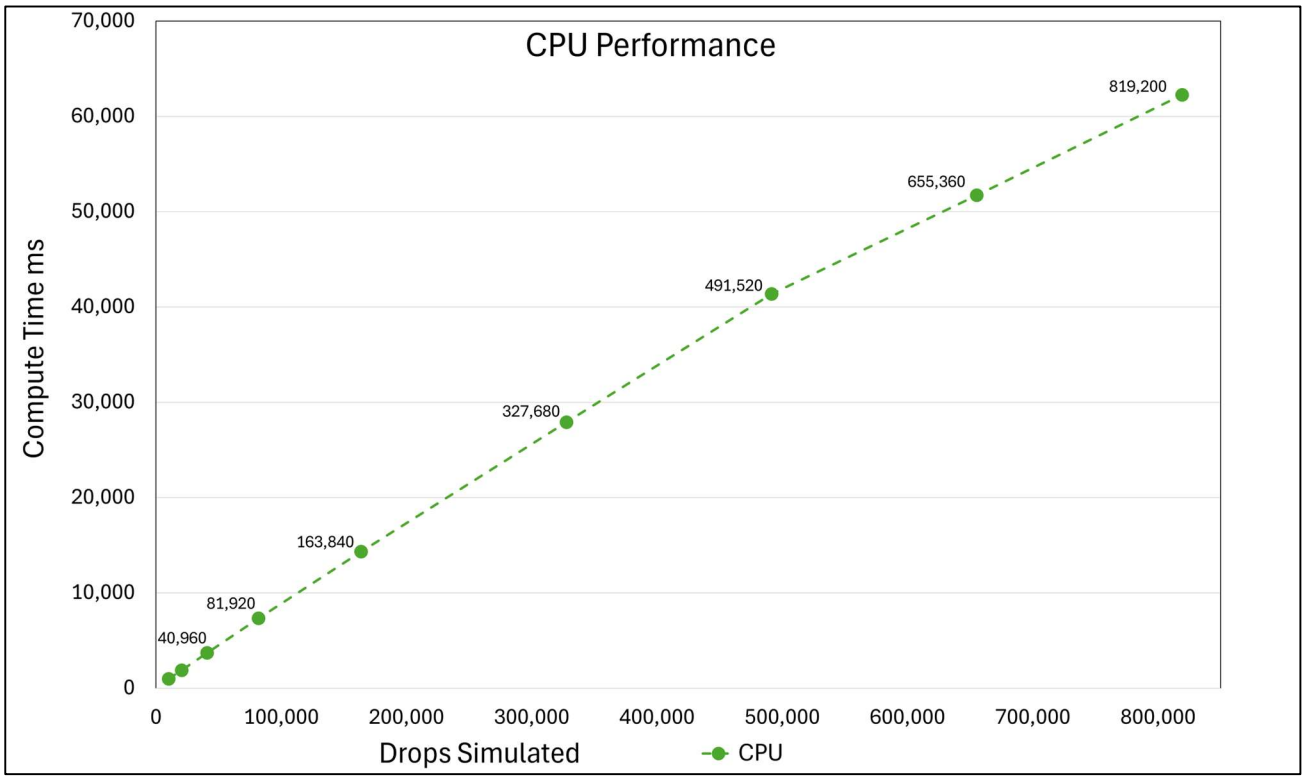


Figure 6: Graph of CPU performance.

The graph in figure 6 shows the performance of my CPU erosion simulation. The compute time increases at a fairly linear rate as the number of drops simulated increases. For my CPU implementation, it is often necessary to run the simulation for these higher compute time drop amounts, in order to make a suitably detailed terrain. This can clearly take quite a lot of compute time however, which is the main disadvantage of the program, especially as the number of drops simulated here is still far from what the GPU is capable of.

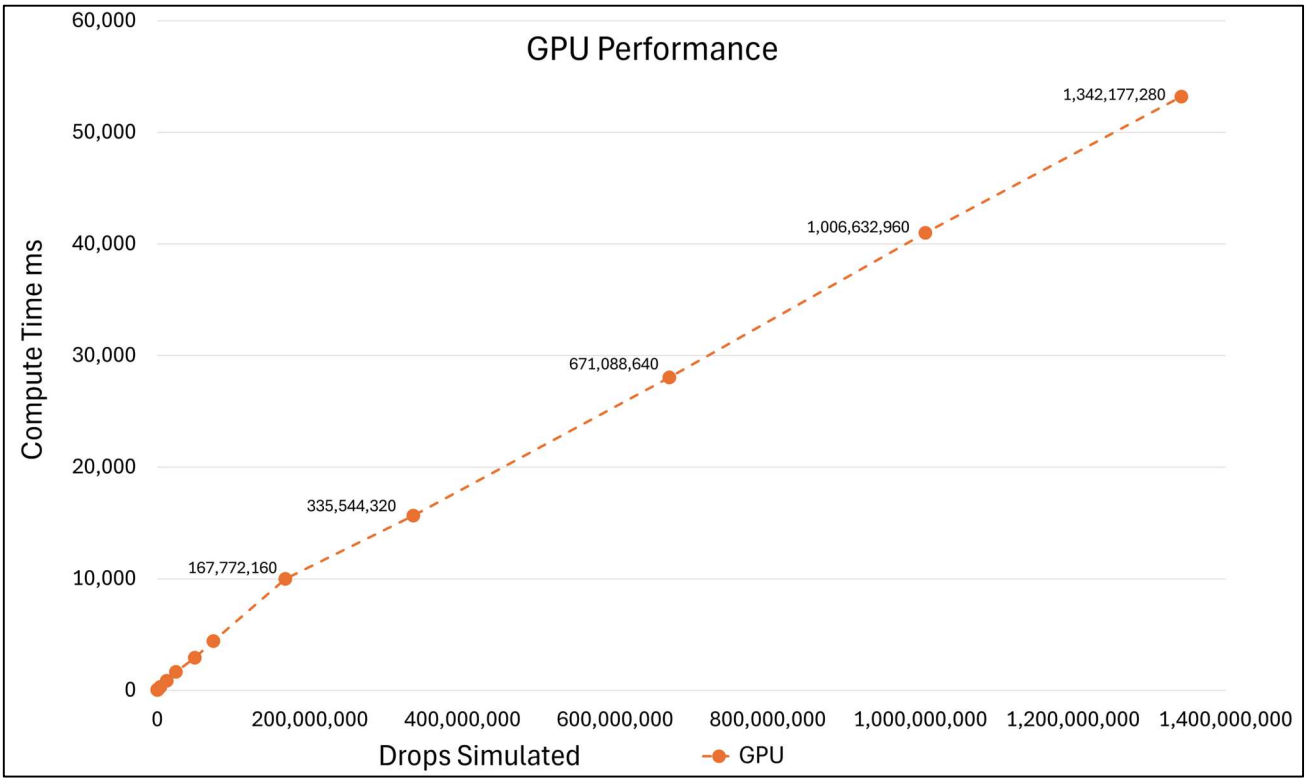


Figure 7: Graph of GPU performance.

In figure 7, the graph shows the performance of my GPU erosion simulation. Similar to the CPU performance, this graph also shows a fairly linear increase in compute time. There are some deviations from this linearity at the beginning of the graph where the compute time is generally very low. This is where my results are less reliable due to randomness in my computer, so are less impactful than the later points which properly stress the simulation. It is important to note the difference in scale between this graph and the previous, in terms of drops simulated. This comparison is clearer in the graph below in figure 8.

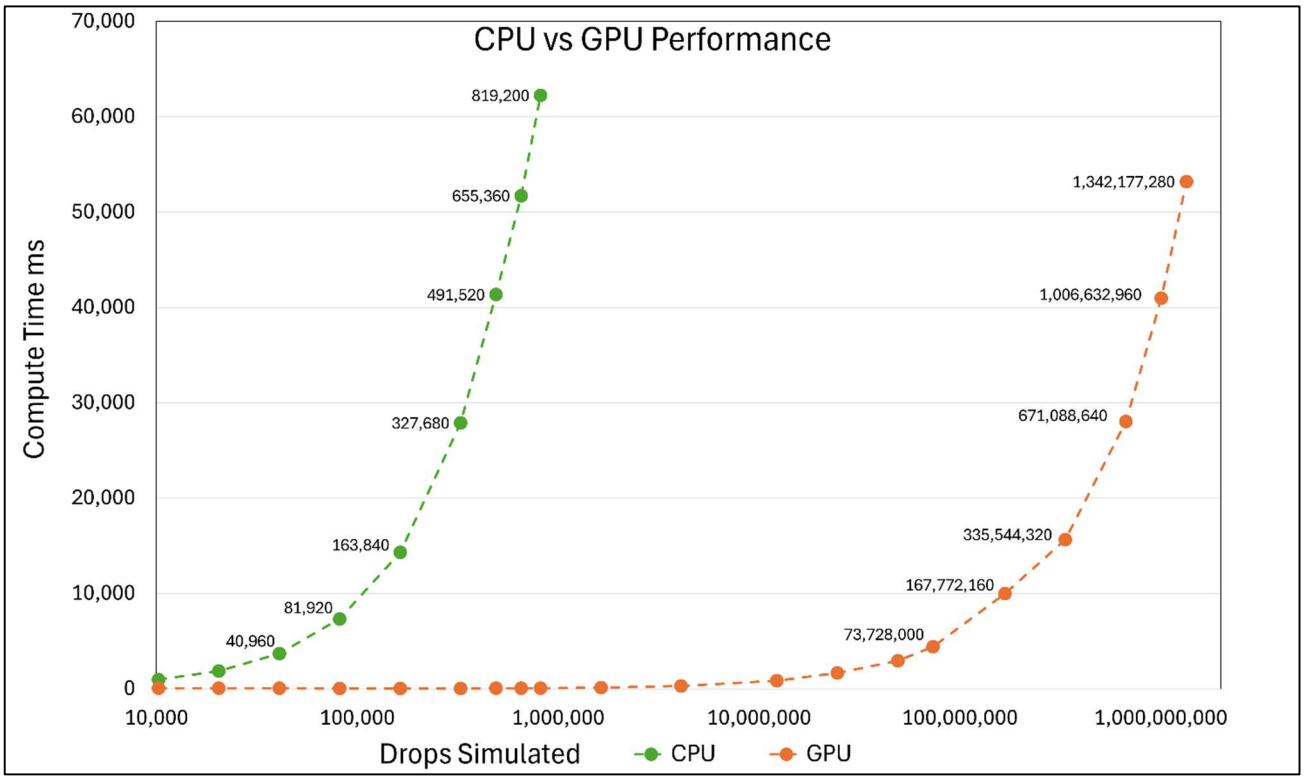


Figure 8: Graph comparing CPU and GPU performance on a logarithmic scale.

This graph in figure 8 shows a comparison in simulation time between my GPU and CPU approaches. This is a fairly direct comparison between the two, as all of my CPU tests were also measured for my GPU approach. All of these, however, take such little time that it does not show the true capabilities of the GPU approach, and so I have continued this testing further specifically for the GPU program.

It is important to note that the x axis showing the number of drops simulated is a logarithmic scale. This is so that it is possible to clearly see both results on the same graph, but this must be taken into account when comparing the two approaches. This is because it makes the difference in performance appear smaller than it actually is. I have therefore also calculated an average drop per millisecond count to explain this more clearly. I calculated these from the five largest tests I did for each approach in terms of drops simulated. The CPU simulation runs at an average of 12.18 drops per millisecond. The GPU simulation runs at an average of 22,382.70 drops per millisecond. This puts the GPU's processing speed at around 1,838 times faster than the CPUs, in terms of drops per millisecond.

Aesthetics

The second criteria I am using to measure the success of my project is based on the visual aspect of my created terrain. I have focused more heavily on efficiency throughout my project as it is more quantifiable, but as the goal for my project is to create a terrain of which could be used in a game environment, the aesthetic appeal of my terrain is clearly very important.

It should be noted that, along with a difference in performance between my CPU and GPU approach, there is also a difference in the look of the final terrain. This is partially due to small differences between the programs caused by changes which needed to be made in order to run on the GPU. But it is mostly caused by this efficiency difference. As my GPU program can easily simulate far more drops than my CPU program in a reasonable amount of time, it can create far more detailed terrains.

I will demonstrate my simulation running on terrains of varying sizes to show the results of my project. These simulations are all run using the same variables, and the number of drops simulated is scaled directly with the increase in terrain size. They are also all created using my GPU program, as this gives better results in much less time due to the massively increased number of drops that can be computed, and so would, in almost all cases, be the best program to use. This performance increase also means that I can simulate at very high scales.

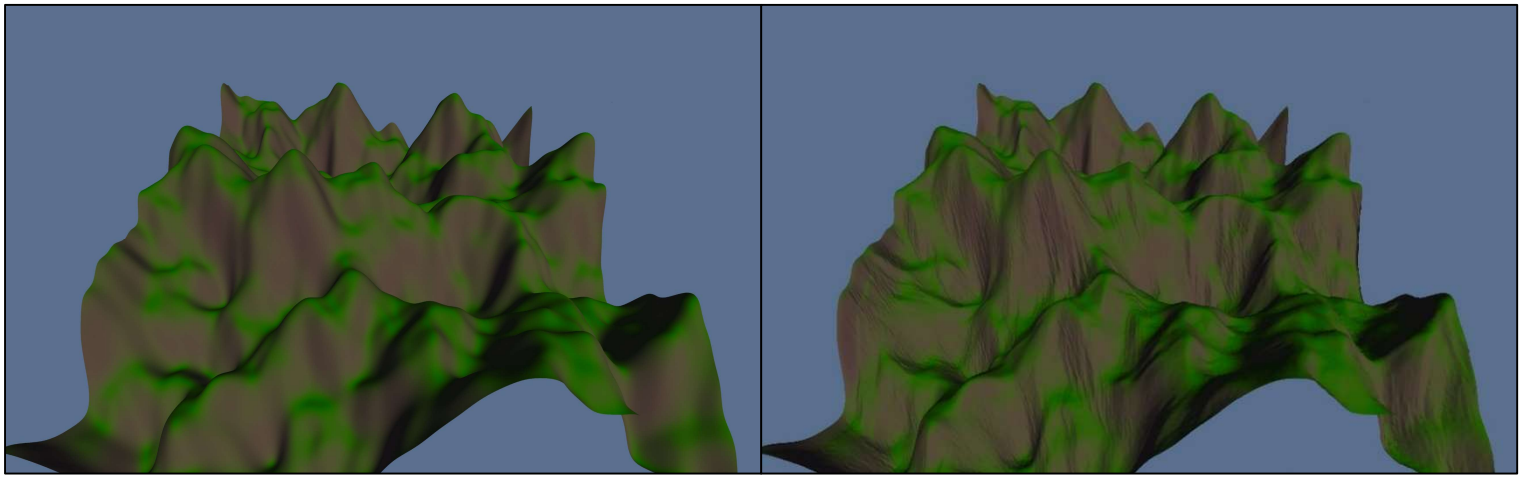


Figure 9: Simulation running on a 512 X 512 grid.

Figure 9 shows my erosion simulation running on a 512 X 512 sized terrain. This is a relatively small scale and so runs for a lower number of drops, it thereby simulates very quickly. I believe the erosion effect works fairly well here, adding detail to the landscape especially at the slopes and cliffs. At this small of a scale however, some texture on the surface is lost.

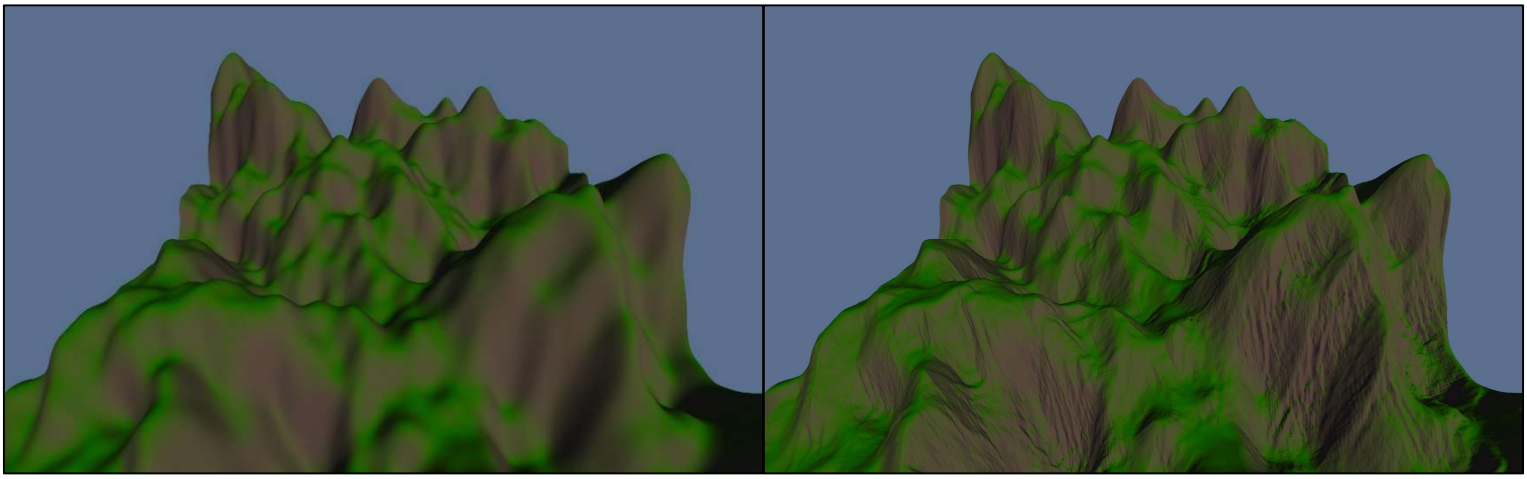


Figure 10: Simulation running on a 1024 X 1024 grid.

In Figure 10, my simulation is running on a 1024 X 1024 terrain. This is the size of terrain which I developed my simulation on. I created the simulation using this size as it has a reasonable scale but can still run my simulation fairly quickly, meaning testing does not take overly long. The variables of my simulation have been set specifically for this size, and I believe this terrain looks better than others because of this. I think that there is a good amount of detail without the landscape becoming too messy and unrealistic. The surface has lots of texture and the overall shape of the terrain has been altered fairly substantially.

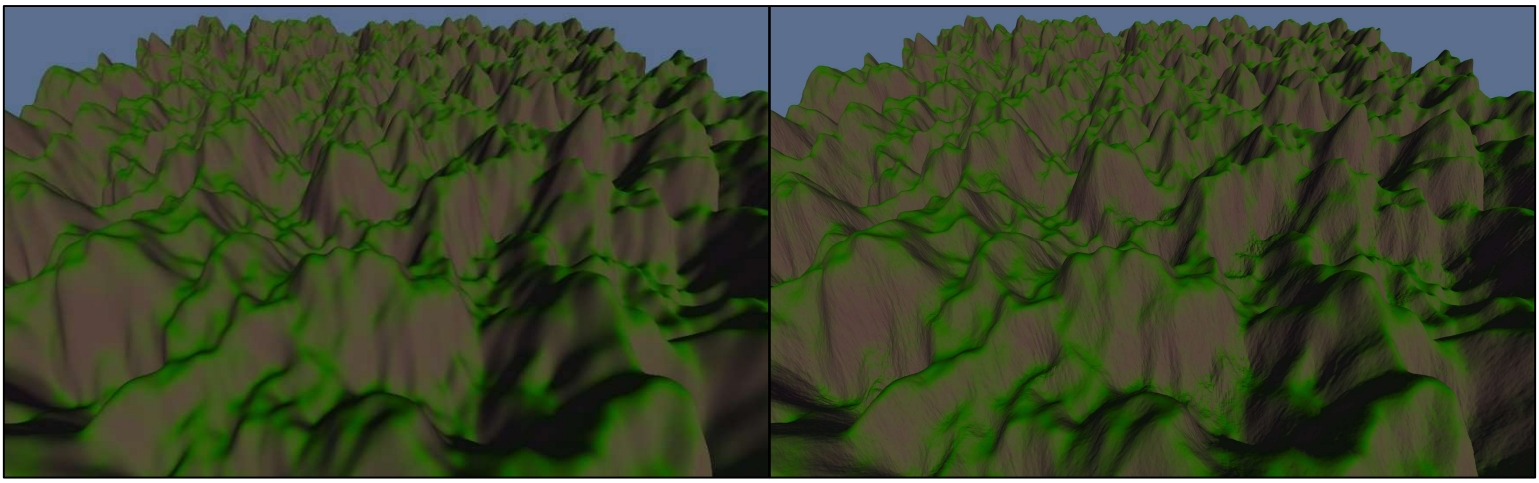


Figure 11: Simulation running on a 4096 X 4096 grid.

Figure 11 is running a simulation on a much larger terrain of 4096 X 4096. For these simulations I have kept the scale of the noise the same as I was testing on so that the variables I used still work to an extent, and to create a more direct comparison. It would be possible to significantly reduce this noise scale to create a smaller environment of a much higher resolution, but this would mean all the variables would have to be completely reset. As the scale is the same as

previous, I feel that the effect of the simulation is still quite good. There are some issues, and it is not as detailed as previous, but to fix these would again mean tweaking the variables. This is possible, but would take a substantial amount of time especially due to the scale of the simulation, where running it can take up to a minute, and many tests would need to be done.

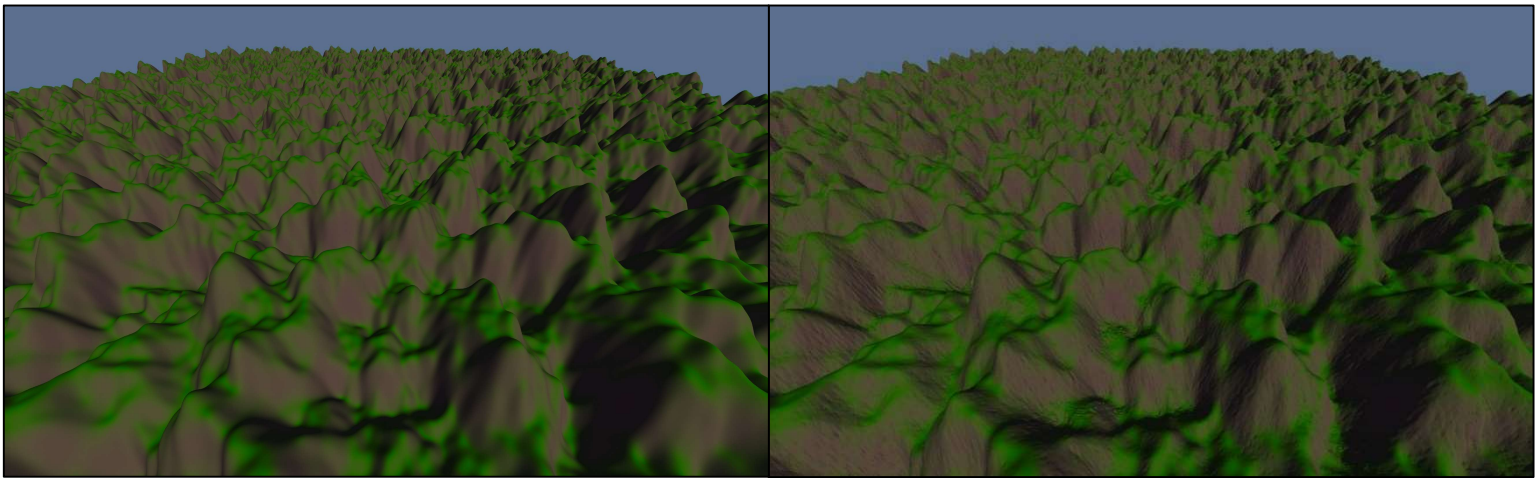


Figure 12: Simulation running on an 8192 X 8192 grid.

Figure 12 is a much larger scale again, running at 8192 X 8192. It is important to note for these size increases, that they are not simply doubling in size. As each edge is twice as large as previous, this makes it a four times increase in the number of points on the mesh, in comparison to figure 11. In this simulation I believe that there is slightly less detail than previous, due to this increase in size. It also seems more prone to making small holes in the terrain rather than smooth slopes down it. However, this could theoretically be fixed by experimenting with the numerous variables and running for even more drops. More examples of my simulation being ran on both the GPU and CPU can be found in Appendices 7.2: Simulation Examples.

Overall, the aesthetics of an environment is difficult to measure objectively. But I am very pleased with the look of my final terrains. I believe that although it can be difficult to set up with the variables, when done correctly it creates terrains which are undoubtably more realistic and detailed in comparison to the fractal noise approach.

4.2 Evaluation

Advantages

I believe that the method I chose for creating a realistic and visually appealing terrain works very effectively, especially in some circumstances. There are some occasions when it would not be advised to use my simulation, which I will cover in the disadvantages section, but here I will focus on the project's strengths.

Realism

The benefits of simulating a hydraulic erosion simulation on a terrain are mostly visual. The simulation takes additional compute time on top of the noise-terrain, but it adds a lot to the environment. The key difference is the realistic looking landscapes that form from this process. My simulation leads to the formation of many real-world features such as mountain ridges, canyons and re-entrants. The edges of the mountains and hills are eroded away to form mountain faces, sediment is deposited around the base of the hills which smooths the landscape.

Aesthetics

While much of the aesthetic advantage given by my method is a direct result of its realism, there are some other reasons why my terrain can look more visually appealing. The erosion formed by drops creates large-scale realistic changes to the terrain, and also adds lots of detail and texture to the surface. Because each drop is individually simulated, the end result is highly detailed, especially in the GPU approach as this simulates many more drops of a smaller impact. This leads to a uniform or flat mesh becoming much more visually interesting for a player in a game world, and along with the increased realism, would certainly add to immersion and engagement.

Disadvantages

There are a few disadvantages to my method for creating terrain. These vary in scope, but I will cover what I believe to be the three key problems in using this approach to create a visually appealing terrain in a game environment.

Processing Time

Even with the optimisation that I have completed, the process of simulating millions of drops eroding a terrain is still very time consuming in comparison to other methods. While maybe only taking a few hundred milliseconds for my GPU approach, this can still be a very large amount of time in a game engine. This processing time could lead to some stuttering if the simulation was used in a procedural setting. If an entire terrain were generated on startup, the erosion step could take significantly longer due to the massive scale needed for such a terrain. In comparison to the purely noise based approach, the erosion simulation adds a much larger amount of processing time required, on top of the noise terrain which still needs to be generated.

While the GPU approach cuts down on this time significantly, it is still slower than alternative methods, and can cause problems in itself. This is because many systems for running games do not have discrete GPU's, especially in the mobile gaming industry. This not only includes games running on a phone, but also on mobile gaming devices such as the *ROG Ally*, and many laptops. Without a dedicated GPU, the code for running the simulation may not work on these devices. Even if it were ported to them, it would still run significantly slower. This is especially relevant for my method, as the performance gain from the GPU approach is massive, and without this speed increase it is certainly infeasible to run this erosion method for games in real time.

Chunk Integration

Many alternative methods for generating terrains use chunks to help reduce stuttering and allow for fully procedural game environments without much time needed to generate the landscape at startup. This is because the entire terrain does not need to be generated at the time the world is created, due to the use of these chunks. Instead, only a relatively small section of the terrain is generated around the player, and terrain chunks are added and removed as the player moves through the game environment. This feature is very important for certain scenarios, and without it many games such as *Minecraft* would not be possible.

The problem here for my erosion method, is that it is not possible to use my simulation with a chunk-based approach. Due to the nature of my simulation, it cannot be broken up into chunks and only simulated a small part at a time. This is because the water drops need to be able to move between the entire terrain as an integrated system. If it were only one chunk being simulated there would be no way of knowing what drops would enter this chunk and affect it

from its surrounding neighbours. It is also not possible to just simulate these direct neighbours, as these would have the same problem and also need their neighbours to be simulated. Overall, the simulation can only run on the entire terrain at once and has no way of being broken up into sections. This means that my method cannot benefit from these chunk based advantages.

Precise Variables

While creating and testing my project I have mainly worked on a 1024 X 1024 grid, with a set scale for my initial fractal noise terrain. This was done to get all the variables correct for the simulation to look realistic. The problem here, is that the process of adjusting these variables to work optimally is very difficult and time consuming. I have many variables which all need to be correct, such as friction, volume, and evaporation.

As well as this, when creating my layering technique for simulating drops of different radii, I inadvertently made this variable problem even worse. This is because each layer has its own value for each of these variables, and these must often differ between each layer. It also makes changing each variable very hard to test without taking large amounts of time. This is because you could simulate one layer at a time, adjusting the variables of just that one layer for simplicity. But this method does not work due to the snowball effect that the value of a variable on one layer can have on all proceeding layers, because they are run consecutively. This issue is, in itself, not too large of a problem, as I spent the time necessary to properly dial in all these settings during development. The main issue occurs when attempting to change any of the external variables for the simulation. If the size of the terrain or scale of the noise is changed in any meaningful way, all of the variables have to be recalculated through extensive manual testing.

In summary, this makes the viability of using this erosion simulation as a simple plugin questionable, as it could not just be added onto any existing terrain. It would instead have to be tuned specifically for each terrain, making the process much more difficult. The many variables can, in some way, be an advantage. It allows for massive amounts of customisation, but the cost of this is a large increase in complexity for using my simulation.

Alternate Approaches

When discussing the strengths and weaknesses of my project, it is important to consider alternative options for creating a realistic and aesthetically appealing terrain. Below, I outline some other methods and their advantages and disadvantages in comparison to my project.

Fractal Noise Terrain

This is the method that I used to create my initial terrain. However, for many applications, it is unnecessary to run an entire erosion simulation on top of this, as the noise itself will be enough. A key advantage of this technique is its ability to be created in chunks, allowing for procedural generation around a player. It is also very quick in terms of processing time and has many variables which can be changed to alter the look of the terrain, without being as particular about the values as my erosion method.

The downside of a purely noise based approach, is that the terrain does not look particularly realistic and lacks detail, even with the addition of fractal layering. This is the main problem I was attempting to overcome in my simulation, and I believe that my erosion has a significant advantage in terms of aesthetics over using just the noise based method.

Diffusion-Limited Aggregation

There are some other methods which are much less similar to my own for creating terrain. One of these methods is called Diffusion-Limited Aggregation (DLA). This theory was proposed in the paper *"Diffusion-Limited Aggregation, a Kinetic Critical Phenomenon" [8].* The overall basis of this method is a process where particles move randomly using Brownian motion and, when colliding, group together to form patterns. This can start as one particle in place and another doing a random walk. Once this second particle collides with the first, it stays in that position and another particle is created which will undergo the same process. If this is repeated many times it can form complex random patterns.

Using this technique, it is possible to create a texture similar to a realistic terrain, which can be used as a heightmap. In the blog post *"Voxel Game Engine Development: Procedural Terrain Heightmap Generation using DLA (Diffusion Limited Aggregation)" [9],* an example of using the DLA technique to create landscapes is presented. The advantage of this method is that the landscapes created are quite realistic looking, and the generation can be edited to create very custom terrains. It is also possible for the terrain to be generated in chunks, as opposed to my erosion method where this is not possible. This process can operate more quickly than my erosion technique, but still slower in comparison to the noise approach.

Importantly however, DLA is not well suited to being ran on a GPU, due to the nature of the generation process. Each particle must have the ability to collide with the next, meaning many particles cannot be computed at once in parallel, as they would be unable to affect each other. This means that while the technique is usually quicker to run on the CPU than my erosion method, it is still much slower than my GPU approach as it cannot take advantage of this massive increase in performance. DLA is therefore more suited to use-cases wherein the program cannot be run on a discrete GPU, such as in mobile gaming.

5 CONCLUSIONS

5.1 PROJECT SUCCESS

Objectives

My first objective was to research and evaluate various methods and techniques such as the Navier-Stokes equations to find the best approach for my simulation. I believe I have completed this objective as my research found that the more particle-based systems such as Navier-Stokes would not be a good fit for my project, which led me to instead focus on a grid approach. This was a key insight for creating my project as, had I attempted to use such approaches, the simulation would have been less efficient and overly complex. I instead focused on researching methods which used a similar approach for inspiration, and created the calculations and equations that I would use myself.

My second objective was to initialise an adequate and suitable terrain to run my simulation on. I succeeded with this objective relatively quickly by creating the custom mesh, and with the use of the FastNoiseLite library, the fractal Perlin noise heightmap was also relatively simple. The terrain I created is randomly generated using a seed which effectively proves that my simulation is capable if working on a range of landscapes, and has the ability for tests to run on the same landscape given they have the same seed. The use of layering the noise in a fractal method allowed me to add some detail into the landscape, while keeping it simple enough for my simulation to properly run on it.

My third objective was to implement the CPU-based erosion simulation. This is the step which took the longest time, as it is the core of my project. Overall, I am very happy with the end result of my simulation; I believe it looks realistic and visually appealing. There are some small changes I would have liked to make in order to create an even more realistic or efficient simulation, however within the timeframe these were not possible.

My fourth objective was to implement the GPU based erosion simulation. This step involved less calculations and deriving equations than the previous, as I was mostly attempting to translate my code into a GPU system. The main challenges here came from the differences when programming for a GPU, but I tried to make my code as similar to the CPU method as possible to allow for a direct comparison in efficiency, and I believe I achieved this to a satisfactory extent.

My fifth and final objective was to evaluate and compare the performance of my CPU and GPU simulations. To complete this, I conducted many tests at varying numbers of drops simulated, measuring the time each program took to run the simulations. I then used these times to find the average for each drop amount and plotted them in a graph to show the differences between the two programs.

I believe that this clearly shows the performance differences and thereby achieved this objective, however I would have liked to conduct even more tests wherein the GPU and CPU were directly comparable. My CPU test, however, cannot handle nearly as many drops without taking significant amounts of time to process. This means that it is infeasible to be able to run a CPU test at the same level needed to properly test the limits of my GPU program. I do still believe that, with the inclusion of the drops per millisecond calculations, the performance difference between these programs is very clear.

Overview

Having covered each objective individually, I will now outline my general thoughts on the success of this project.

Considering firstly, that all of my objectives were fully completed. Some of these I would have liked to put additional work into, but all of which were completed to the specification I outlined in my project proposal. As well as this, my end goal for the entire project was to create a realistic terrain, detailed to a level above that possible through just fractal noise. I believe that this has also been completed, and that the created terrains are of a high enough standard wherein they could be used in real-world settings such as in use creating a virtual environment for a video game.

The performance of the simulation is also an important factor in its success. To this end, I was initially doubtful that my simulation could realistically be used in game development due to its high processing time for suitably detailed terrain. However, this was an observation of my CPU program and, when I finished my GPU simulation, I had no such concerns. The GPU approach is able to create massive-scale, highly detailed terrains in seconds, and so is definitely a viable option in the real world in terms of performance.

Having said this, there are some significant drawbacks of my method of which cannot be ignored. The higher processing time, in comparison to the noise-based approach, makes real-time terrain generation much less possible with my method. Also, the inability to utilise chunk integration means my method cannot be used for fully procedural world generation around the player. The variables, which all need to be precisely correct both in comparison to each other

and the initial terrain, are also a big issue for real-world use. This makes my program much more difficult to integrate into a workflow than just working with the press of a button, as I had initially hoped.

It is important to note when evaluating the success of this project, that many of the aforementioned drawbacks are systematic issues ingrained in the nature of my method. These are issues that all projects focusing on creating an erosion simulation for the creation of a terrain will encounter, many of which are unavoidable. I have attempted to mitigate against some of the issues, especially the performance cost of my simulation, by optimising the efficiency of my program as much as possible. But even with these improvements it is impossible to overcome the core issue that simulating millions of drops in an erosion pass will always take a significant cost in performance.

5.2 Personal Improvement

Over the creation of my project, I have learned many techniques and insights into a range of computing skills. Here I will cover some of the most impactful of these in greater detail.

Grid-Based Systems

The technique I used for creating my initial terrain and simulating drops required a very in-depth knowledge of grid-based systems. With all of my heightmap points being stored in a single Vector3 array in a very specific order, fully understanding how to navigate this array was vital in creating my simulation. Throughout the course of my project, I calculated many factors from this array such as finding the points around an index in a radius. I also needed to derive a direction of travel using the index of two points, and find where a drop's simulation should end at the edges of my mesh according to its radius.

To complete these, I was required to understand very specific forms of mathematical calculations and equations relating to the order of my grid points. I believe that I now have a full understanding of how to navigate and edit a grid-based system and would feel comfortable in creating and using similar systems again in the future.

GPU Programs

When planning this project, one of my largest concerns was my lack of experience with GPU programming. This means that I had to learn many things about not just the programming, but also how the underlying systems work in order to fully understand how my simulation would run. Throughout the creation of my GPU program, I researched this topic thoroughly and I believe that I now have much better knowledge of how these programs work. I believe that this particular skill may be the most useful for my future work as I plan on continuing working in game development, where GPU based programming is very common and is a useful tool for many situations.

Natural Sciences

When deciding to create a simulation of hydraulic erosion, I knew that I would have to put some time into researching aspects of natural sciences. For example, I have learned a lot about geology throughout my project, specifically relating to how certain features such as mountain ridges and talus slopes are formed. I also researched how erosion effects different kinds of rock and the differences in the landscapes depending on this. This specific subject is something I would like to explore in future work, but for now I believe I have a good baseline of understanding for these processes.

I have also studied applied physics, in the form of fluid mechanics. As I was creating my own simplified version of fluid dynamics equations, I needed to have a good understanding of the underlying concepts in order to accurately simulate my drops. While my equations are far less accurate than others such as Navier-Stokes equations, they suit my needs well and are more tailored for my project, focusing more on efficiency over complete realism. The knowledge I have gained from this may be less directly related to traditional computing, however I believe it will be very useful in my future game engineering work. Many games emulate the real world in their systems, such as fluid simulations and physics engines. I thereby believe that a base knowledge of the natural sciences is a valuable tool in game engineering.

Research and Literature Reviews
In order to learn many of the required skills, I undertook extensive research into relevant subject areas. I used scientific papers for this task as much as possible, to ensure that my dissertation was underpinned by sufficient academic research. During this time, I improved my researching skills through experience. This is undoubtedly a key skill which will be important to me during my next year at university completing my master's degree. It will also be invaluable when working in computing, especially for game development. In this industry it is common to work on subjects or areas you may be initially unfamiliar with, and so organised and effective research can substantially cut down the time it takes to learn these new topics.

5.3 Future Work

There are a few options which I would like to explore to further this project. Because of time limitations, I was unable to complete some of the additional ideas I had while undertaking the project. Areas of interest which emerged during the project, but were not in my proposal or project plan, are outlined below, including a description of why they would be useful or relevant going forward.

Creating a Library
The most obvious next step for my project would be to use it to create a library or plugin which others could use to generate terrain with my simulation. There are a few changes I would need to make for this to be more useable for developers. This includes integrating my simulation with the built-in Unity terrain asset, so that it does not need my custom mesh to be used. This would make the library much easier to integrate into a developer's workflow as they may have already made a terrain using this asset and wish to simply enhance it.

I would also need to make the UI and layout of my variables more understandable, as well as provide in-depth explanations of what each variable does and how to adjust them all together to create a simulation which works for a terrain. Unfortunately, this would still take a developer some time to properly set these variables for their terrain. I had hoped I could create a system that could simply be added to any terrain and work well, but I do not believe this to be possible with my method. Nevertheless, my simulation could still be a very useful tool for developing game environments and creating a library or plugin would allow others to use it relatively easily for such purposes.

Experimenting with Rock Types
As mentioned in my personal improvement section, I spent time researching geology during the course of this project. During this research I discovered the vast differences in which different types of rock change due to hydraulic erosion. I theorised that an extension onto my current simulation could involve changing the way the erosion works dependent on the rock's

attributes, such as how porous it is. For this I would have to add many more variables which would change the end result of my erosion depending on the rock.

It would also be interesting to attempt to simulate multiple kinds of rock within the same landscape, with the simulation eroding each type in a slightly different way. This would, however, involve far more extensive and in-depth geological knowledge than I currently have. While this addition may not be as useful for game development, it would push the project more towards a realistic scientific simulation as opposed to a tool for creating landscapes, as is my current goal. My current project has focused more on efficiency and performance at a level of realism that is adequate for a terrain in a game environment. This change, however, would lead the project to focus more on this realistic simulation, changing the overall goals of the project towards something more similar to a scientific visualisation.

Tectonic Initialisation

In a similar subject area to differing rock types, I could also include another more realistic approach for initialising my terrain. Currently I use random noise for a suitably detailed and random terrain. However, it is possible to simulate the movement of tectonic plates in order to create an initial terrain.

As shown in *"Sculpting Mountains: Interactive Terrain Modeling Based on Subsurface Geology" [10]*, a basic mountainous terrain can be formed with the simulation of two tectonic plates moving towards each other, called continental collision. Using this system allows for the creation of more complex geographical features, as explained in the paper: *"terrains modeled using our system capture important landform features of mountain ranges, such as the asymmetry in landform resulting from the presence of a crustal scale fault and multi-scale folding of crust layers appearing in eroded regions"*. Using this system to initialise my terrain would enhance the realism significantly, although this would come at a significant performance cost. It could also be used in conjunction with the different rock types, as the tectonic simulation would dictate the location of each kind of rock.

Expanding the Erosion Scope

My current project has focused entirely on hydraulic erosion, exclusively in the form of rainfall. A real landscape, however, is eroded from a number of sources. This includes erosion from wind, glaciers, vegetation and freeze thawing.

If I were to focus more on the scientific side of my simulation as mentioned above, I would want to include as many of these erosion types as possible. For example, creating a wind erosion using vector fields and using this to alter the terrain accordingly. Taking into account vegetation would also work well with my previous rock types work, as the level of vegetation on a given surface has a large impact on how it is affected by erosion. This is due to the roots of plants holding the soil together and absorbing the ground water, decreasing the effect of hydraulic and wind erosion.

Including all of these erosion types in one simulation, as well as tectonic initialisation and differing rock types, would be a significant task, and involve fundamentally changing how my original erosion simulation works. It would also require extensive research and expertise in how these geological processes work in order to be as accurate as possible. This is clearly well beyond the scope of my current project, and even that of a master's dissertation, but would present an interesting challenge for a PhD thesis.

6 REFERENCES

[1] R. Dwiar, “We asked a landscape designer to analyse The Witcher 3, Mass Effect and Dishonored”, Eurogamer.net, 2017.
Available: https://www.eurogamer.net/a-landscape-designers-take-on-the-witcher-3-mass-effect-and-dishonored

[2] N. Chiba, K. Muraoka, and K. Fujita, “An erosion model based on velocity fields for the visual simulation of mountain scenery”, Faculty of Engineering, Iwate University, Morioka, 1998.
Available: https://doi.org/10.1002/(sici)1099-1778(1998100)9:4%3C185::aid-vis178%3E3.0.co;2-2

[3] Bedřich Beneš and R. Forsbach, “Visual simulation of hydraulic erosion”, 2002.
Available: http://wscg.zcu.cz/wscg2002/Papers_2002/F23.pdf

[4] X. Mei, P. Decaudin, and B.-G. Hu, “Fast Hydraulic Erosion Simulation and Visualization on GPU”, 15th Pacific Conference on Computer Graphics and Applications, 2007.
Available: https://doi.org/10.1109/pg.2007.15

[5] T. Le, “Procedural Terrain Generation Using Perlin Noise”, California State University, 2023.
Available: https://scholarworks.calstate.edu/downloads/m900p2577

[6] J. Stam, “Stable fluids”, Proceedings of the 26th annual conference on Computer graphics and interactive techniques - SIGGRAPH ’99, 1999.
Available: https://doi.org/10.1145/311535.311548

[7] P. Diaconis and F. Mosteller, “Methods for Studying Coincidences”, Journal of the American Statistical Association, vol. 84, no. 408, 1989.
Available: https://doi.org/10.1080/01621459.1989.10478847

[8] T. A. Witten and L. M. Sander, “Diffusion-Limited Aggregation, a Kinetic Critical Phenomenon”, Physical Review Letters, vol. 47, no. 19, 1981.
Available: https://doi.org/10.1103/physrevlett.47.1400

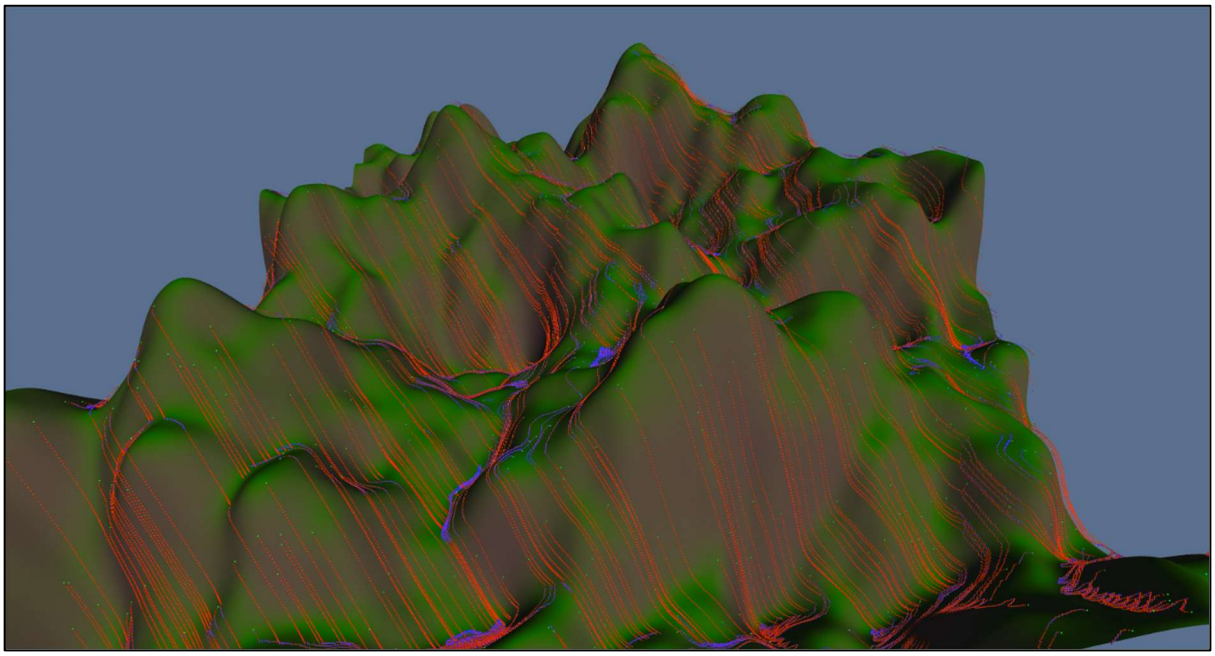
[9] Spacerat, “Voxel Game Engine Development: Procedural Terrain Heightmap Generation using DLA (Diffusion Limited Aggregation)”, Voxel Game Engine Development, 2014.
Available: https://voxels.blogspot.com/2014/01/procedural-terrain-heightmap-generation.html

[10] G. Cordonnier, M.-P. Cani, B. Benes, J. Braun, and E. Galin, “Sculpting Mountains: Interactive Terrain Modeling Based on Subsurface Geology”, IEEE Transactions on Visualization and Computer Graphics, vol. 24, no. 5, 2018.
Available: https://doi.org/10.1109/tvcg.2017.2689022

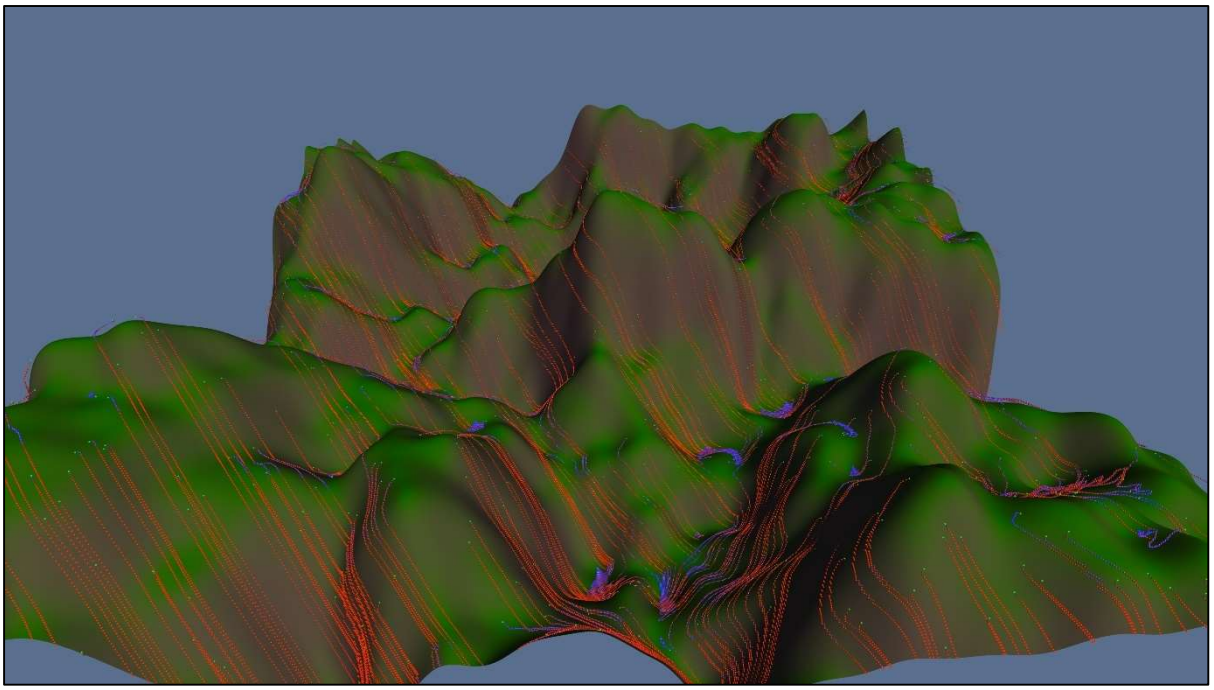
7 APPENDICES

7.1 MOVEMENT VISUALISATIONS

1)

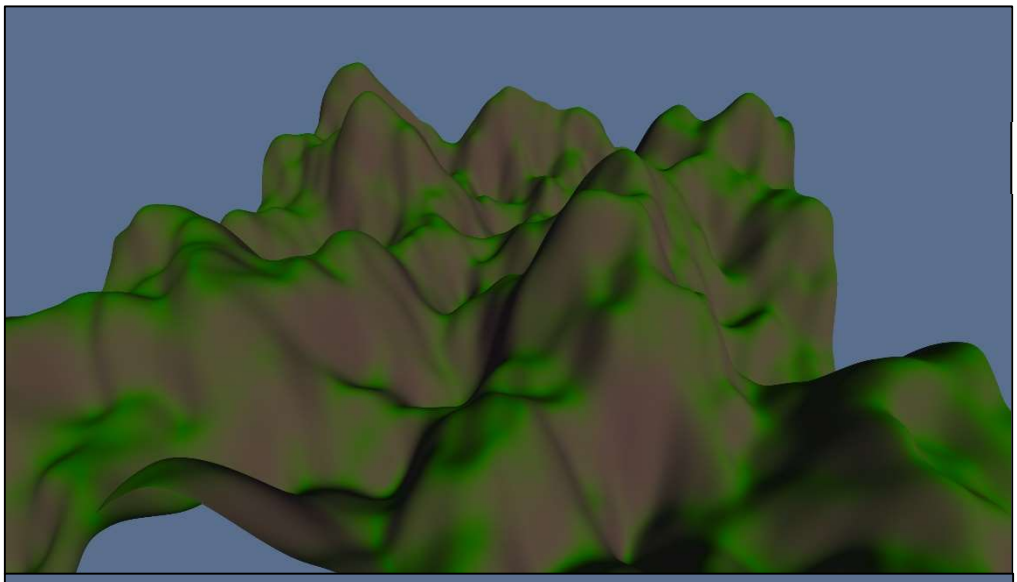


2)

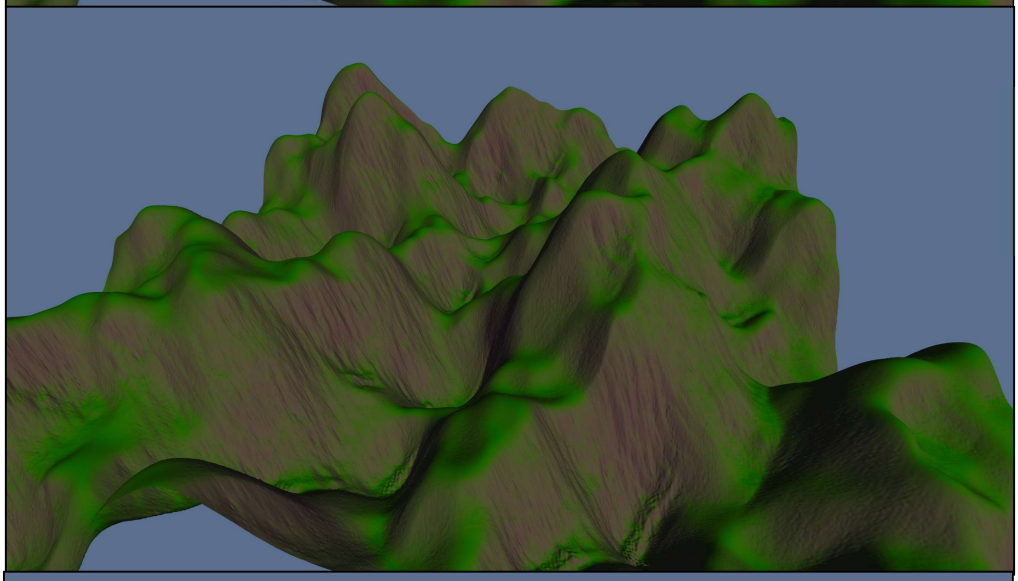


7.2 Simulation Examples

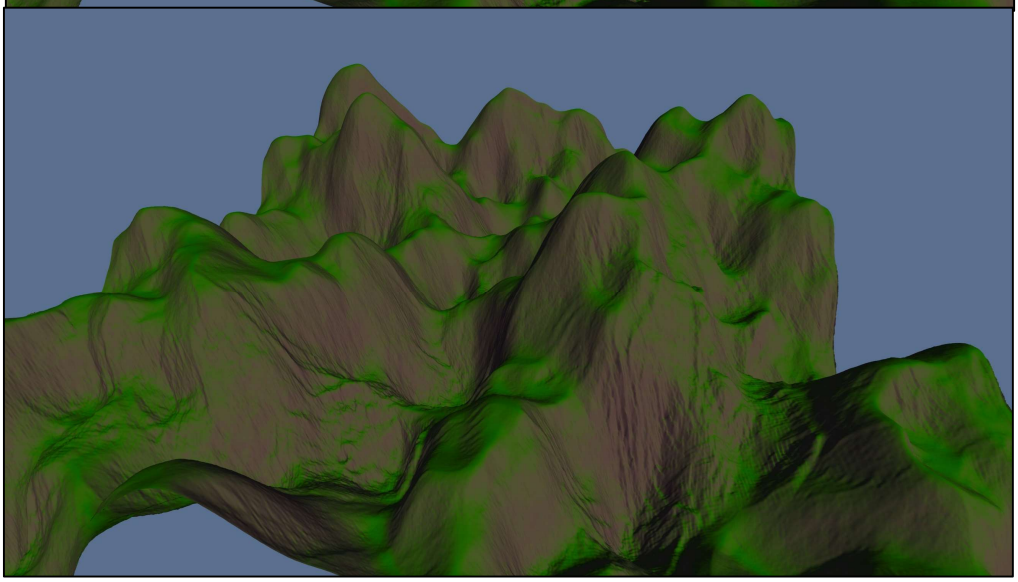
1)



Initial Terrain
Size: 1024 X 1024

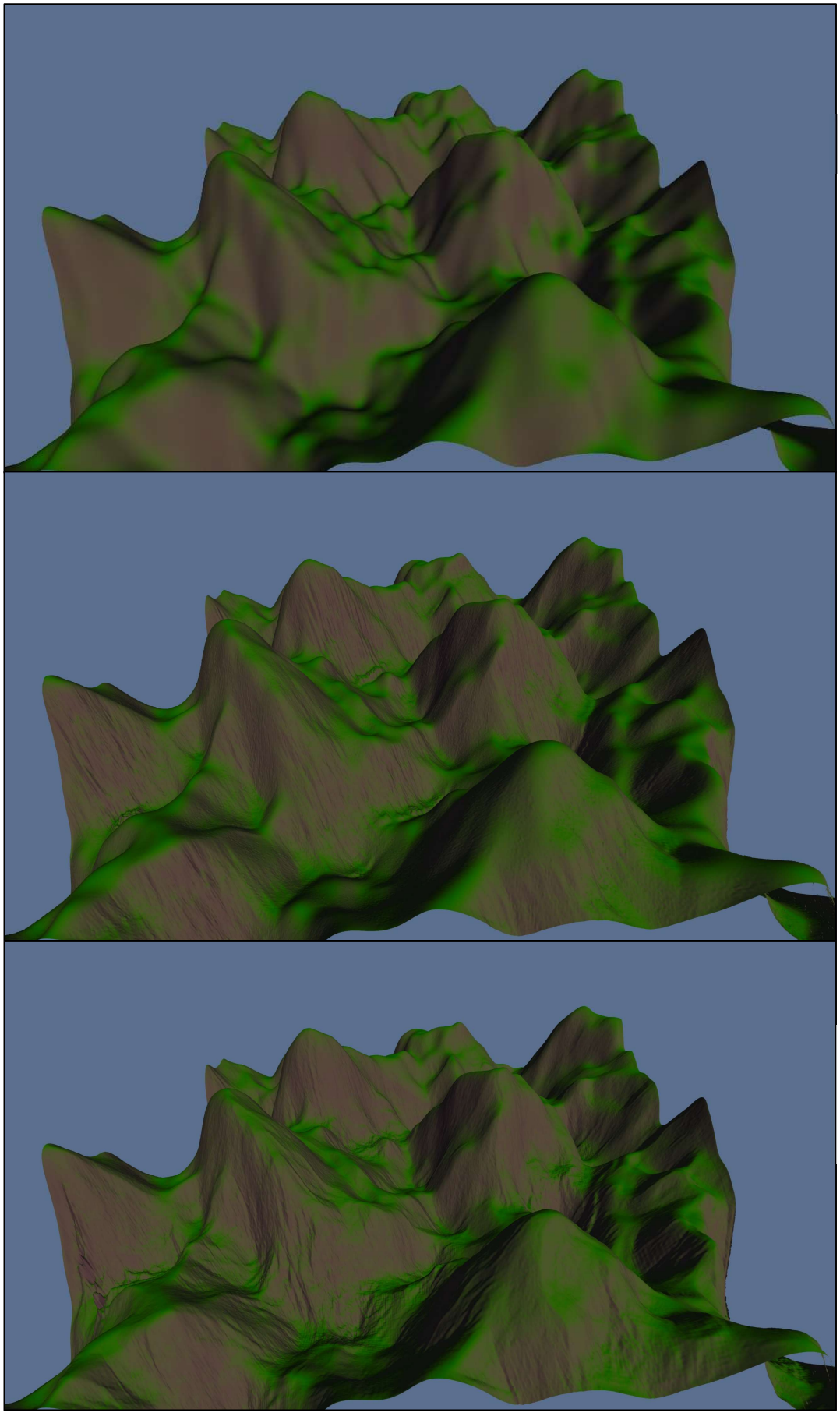


CPU Simulation
Drops: 327,680
Compute Time: 29,526 Ms



GPU Simulation
Drops: 24,576,000
Compute Time: 1,662 Ms

2)

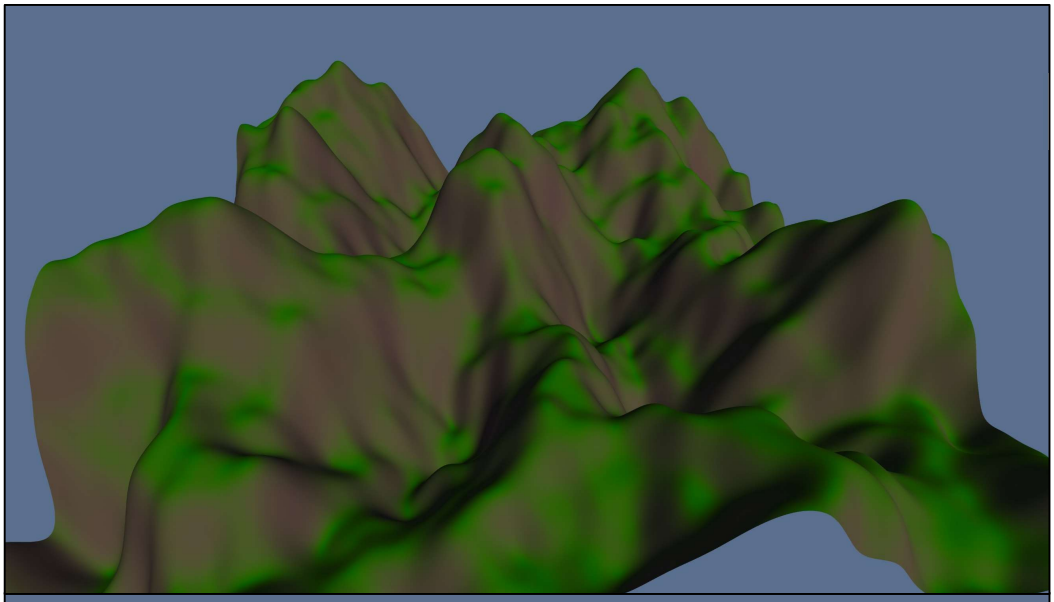


Initial Terrain
Size: 1024 X 1024

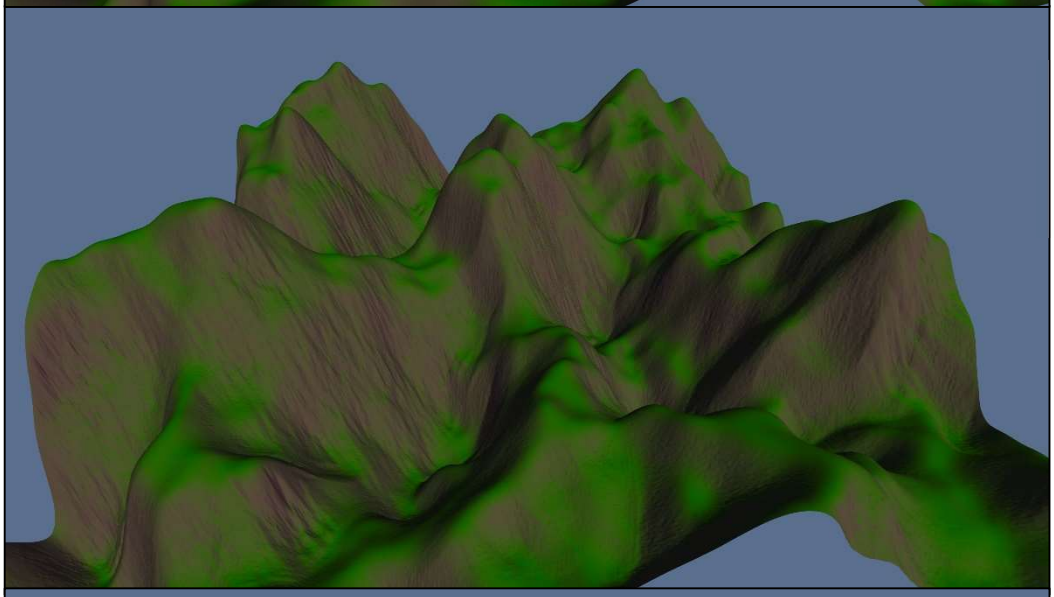
CPU Simulation
Drops: 327,680
Compute Time: 28,573 Ms

GPU Simulation
Drops: 24,576,000
Compute Time: 1,734 Ms

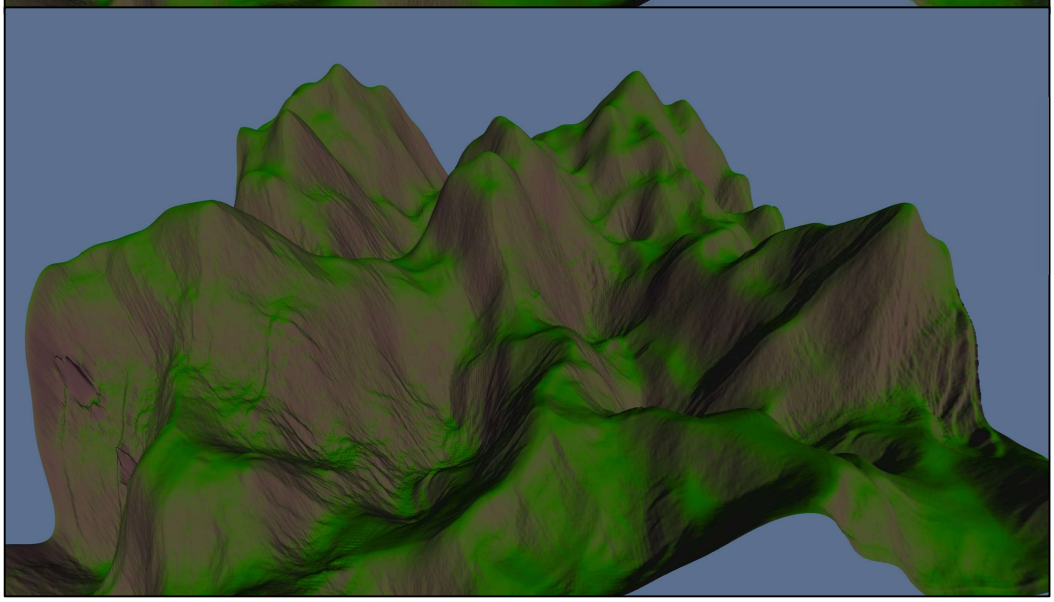
3)



Initial Terrain
Size: 1024 X 1024



CPU Simulation
Drops: 327,680
Compute Time: 29,147 Ms



GPU Simulation
Drops: 24,576,000
Compute Time: 1,719 Ms

7.3 Recording of Simulation Running

Available:
https://drive.google.com/file/d/1OwGJ3qOrgH1KKPgNI0Lw7334Zv7VHsWz/view?usp=sharing

7.4 Project Repository

Available:
https://drive.google.com/drive/folders/1-wSm_g80Buxtml9u0XQsb723G_UiTIIc?usp=sharing