School of Computer Science

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

Final Report

Procedural Terrain Generation and Optimisation for Infinite Virtual Environments

Nicholas Mann

Submitted in accordance with the requirements for the degree of
BSc Computer Science

2024/25

COMP3931 Individual Project

The candidate confirms that the following have been submitted*:*

Items	**Format**	**Recipient(s) and Date**
Final Report	*PDF file*	*Uploaded to Minerva (30/04/2025)*
Link to online code repository	*URL*	*Sent to supervisor and assessor (30/04/2025)*

(Signature of student) [illegible]

Summary

This report investigates the development and optimisation of real-time, procedurally generated terrain for virtual environments, with a focus on achieving high-quality terrain that can be expanded infinitely. This paper examines various methods of generating terrain Perlin noise, hydraulic and thermal erosion models, and hybrid techniques. The goal was to improve the realism of the terrain while maintaining real-time performance.

The methodology presented addresses the problems and challenges that come with dividing the terrain into chunks that can be independently processed. These chunks are then seamlessly placed together to create the illusion of a single continuous terrain. Perlin noise serves as the base for the terrain generation and acts as a reference point when trying to sync the chunks. Hydraulic erosion is applied to enhance realism by simulating natural weathering effects.

The techniques implemented in the paper were done using C++, and continuously validated by generating and rendering a terrain at various points of the process to check that things are working as intended. The system's performance and visual quality were evaluated, revealing significant improvements in terrain aesthetics. However, performance benchmarks indicated that adding erosion and smoothing effects led to substantial increases in processing time, which could hinder its practical application on lower-end systems.

Future work will focus on refining chunk continuity, optimizing performance, and exploring new erosion techniques to enhance both the visual quality and computational efficiency of the system. Despite current limitations, the approach presented in this report holds promise for developing more detailed and expansive virtual environments in real-time applications.

Table of Contents (example of how to format)

Summary .... **iii**

Table of Contents (example of how to format) .... **iv**

Chapter 1 Introduction and Background Research **1**

1.1 Introduction 1

1.2 Literature Review 2

1.2.1 Realistic and Textured Terrain Generation using GANs 2

1.2.2 Realtime Procedural Terrain Generation 2

1.2.3 Terrain Generation Using Procedural Models Based on Hydrology 2

1.2.4 Terrain Amplification using Multi-scale Erosion 3

Chapter 2 Methodology **5**

Chapter 3 Implementation and Validation **7**

Chapter 4 Results, Evaluation and Discussion **15**

5.1 Evaluation 15

5.1.2 Terrain Appearance 15

5.1.2 Performance 17

5.2 Conclusions 17

5.3 Future work 17

List of References **19**

Appendix A Self-appraisal **20**

A.1 Critical self-evaluation 20

A.2 Personal reflection and lessons learned 20

A.3 Legal, social, ethical and professional issues 21

A.3.1 Legal issues 21

A.3.2 Social issues 21

A.3.3 Ethical issues 21

A.3.4 Professional issues 21

Appendix B External Materials **22**

Chapter 1
Introduction and Background Research

1.1 Introduction

Virtual environments have been used in games as early as 1973, with the original Maze game also known as Maze Wars. This included a default maze environment but also allowed the player to define their own at the start of a new game. Since then, there have been massive improvements in computer power and new developments in computer graphics. This has enabled new games to be able to create vast and unique environments, as demonstrated in No Man's Sky, released in 2016 but has been continuously developed to improve the environments as late as 2024.

These improvements have resulted in the development of programs like Unreal Engine 5, which is capable of creating environments so detailed that they are being used in the background of films to create environments that would otherwise not exist on Earth. A good example of this is the 2021 Dune, which used computer-generated environments for the majority of the background.

Although the environments used in No Man's Sky and Dune are both virtual, they are not directly comparable because they use different techniques for generating and rendering the terrain. No Man's Sky's environments are procedurally generated, which means the terrain is generated and rendered in real-time. This enables the environments and, therefore, the terrain to be completely random each time a new one is generated. The terrain used in Dune is pre-generated with the help of a human, altering some aspects to improve the result. Later, it is rendered for the film, taking anywhere from a few minutes to a few days to render out a single scene. This approach is used for film since it allows for more detailed terrain, but it is simply not practical for games since it would take too long to render each frame, making the game unplayable.

A middle ground used in games is to have pre-generated and sculpted terrain like in films, which allows for additional detail in the terrain that is not possible with real-time terrain generation. Then, a quicker, less detailed renderer is used to allow it to still be rendered in real-time. This enables some very convincing environments, as seen in the game The Last of Us Part II. However, these environments are not suitable for all situations and tend to only lend themselves to more story-driven games.

With improvements in terrain and rendering generation techniques, the divide between these three different methods will get smaller. This report focuses on procedural real-time terrain as used in No Man's Sky to investigate ways of improving the quality of the terrain without sacrificing the ability to expand infinitely.

1.2 Literature Review

Since the report is on improving procedural terrain, the research was focused on what techniques were already available for producing terrain and how easily they could be adapted to allow for infinite generation. Below are a handful of notable papers using different techniques:

1.2.1 Realistic and Textured Terrain Generation using GANs

The paper "Realistic and Textured Terrain Generation using GANs" by Ryan Spick and James Alfred Walker, introduces an approach for realistic and textured terrain generation using Generative Adversarial Networks (GANs). The aim was to automate the process of generating and texturing 3D terrains used in virtual environments. The idea was to leverage real-world data, including NASA’s SRTM (Shuttle Radar Topography Mission) elevation data and satellite images, to train a Spatial GAN (SGAN). This would generate both the heightmaps and corresponding textures in a single model. The method improves upon previous techniques such as DCGAN (Deep Convolutional Generative Adversarial Network) by offering enhanced visual fidelity and structural similarity to real-world regions. The results show that the SGAN model can produce varied and realistic terrains, making it a valuable tool for developers seeking efficient, automated terrain generation. This paper also included discussions on optimising techniques used to train GANS, and potential limitations with using this method with highly Varied terrain [1].

1.2.2 Realtime Procedural Terrain Generation

Jacob Olsen’s study presents fast methods for generating eroded fractal terrain suitable for real-time applications like computer games. The paper defines an "erosion score" to assess terrain quality based on slope characteristics and evaluates traditional terrain synthesis techniques such as spectral synthesis, midpoint displacement, and Voronoi diagrams. He goes on to analyse two traditional erosion models, thermal and hydraulic erosion, and compares their performance. He finds that while hydraulic erosion produces better results, it is too slow for real-time use. Olsen proposes a new algorithm that modifies thermal erosion to rapidly flatten low slopes, achieving higher erosion scores significantly faster. This method was successfully implemented in the games he developed, Tribal Trouble, demonstrating the feasibility of real-time procedural terrain generation [2].

1.2.3 Terrain Generation Using Procedural Models Based on Hydrology

The focus of this paper was an approach for using procedural models inspired by hydrological principles to generate terrain. This starts by generating a network of rivers which is used to form the foundations for the terrain's features, such as valleys, hills and mountains. The river network is formed using a geometric graph. Features are extracted from this graph by matching predefined graphs representing a certain feature to areas in the main graph. The result is terrain that is consistent with the geometry of terrain found in the real world. The resulting terrain is one that is largely scalable, supports multiple resolutions, and has a fast computation time compared to other methods [3].

1.2.4 Terrain Amplification using Multi-scale Erosion

This paper presents the idea of using multi-scale terrain amplification methods using erosion simulation to enhance lower resolution terrains, generating new higher resolution ones. The approach blends physics-inspired erosion models, such as stream power erosion and thermal erosion, with fractal procedural terrain. Fractal terrain is the process of stacking heightmaps with differing levels of detail on top of each other so that the larger detail affects the overall terrain more than the smaller ones. The techniques used for erosion allow the designers to control the intensity, location, and scale of erosion effects, and preserve important terrain features like peaks and ridgelines. The method iteratively amplifies the terrain by applying different erosion processes at each resolution, ensuring realistic landforms and drainage patterns. This simulation can be performed in real time, allowing for an interactive terrain editor [4].

1.2.5 Perlin Noise

Perlin noise is a mathematical algorithm developed by Ken Perlin in 1983 for generating noise maps, where each pixel in the image goes between 0 and 1 with smooth but random transitions. This works by generating a grid of random vectors, each of length one. This grid is overlaid on top of the image in which the noise is to be generated. For each pixel in the image, you find the closest four vectors on the grid and compute the dot product between that vector and the normalised vector from the grid position to the pixel being checked. The resulting values are interpolated based on the distance from the pixel to each point. This results in an image with random noise. With the standard implementation of Perlin noise, two potential problems can arise. The first is that, depending on the random function used to generate the vectors, the repeatability of the noise could be compromised, resulting in a different noise each time it's generated. The second problem is that there are limitations on the size of the grid that can be used. The vector grid can be bigger than the image size, but also can't be smaller than one. E.g. having a single cell of the grid bigger than the image. Both problems would need to be addressed in order for this to be suitable [5].

1.2.6 Analysis of Methods

Each of the methods discussed above brings its own approach to the problem. But no single method achieves the results needed for infinite procedural terrain generation. To this end, the best approach is a blended one, taking elements from different papers to make a more suitable method. The resulting method will use the fractal procedural terrain discussed in 1.2.4 with Perlin noise as the noise function for a base terrain mentioned in 1.2.6. On top of this base terrain, hydraulic erosion from 1.2.2 will be added to improve the terrain's visuals.

Chapter 2
Methodology

Choosing a terrain technique:

This report aims to investigate ways of adapting techniques to generate infinite terrain. Since it is infeasible to generate an infinite amount of terrain at the same time, the terrain needs to be broken into smaller chunks that can be processed individually to form a continuous terrain when put together. These chunks are then generated at request as you move around to give an illusion of infinite terrain. Each chunk needs to be able to generate without knowing any of its surroundings but still keep continuity with any chunks that surround it.

Due to this concept of keeping the chunks isolated, not every terrain generation technique is suitable for adaptation without massive algorithm changes. Some problems with techniques are listed below:

Any technique that requires the user to define a starting landscape or guide is not suited since it is not feasible to specify an infinite size area.

Methods that use AI (Artificial Intelligence) or GANs have an issue with continuity between chunks without having knowledge of the existing terrain. Since, ideally, the chunks can be generated without knowledge of their surroundings, this means that using AI or GANs is a bad idea.

Terrains generated using graphs could be possible for adaptation since it is feasible for the graphs to have the ability to grow when new chunks are needed, but this was deemed to be too ambitious to finish within the timeframe.

On the other hand, procedural techniques are well-suited for adapting with little to no modification needed in most cases. This is because they primarily rely on cellular noise. Cellular noise can be generated by using continuous mathematical functions, meaning neighbouring chunks can be inferred just by sampling the same point, thus keeping their independence.

Using procedural terrain as a base, physics-based erosion can be applied to improve the final result of the terrain. Physics-based erosion aims to emulate environmental factors found in nature. These include thermal erosion (freezing and thawing of the ground), hydraulic erosion (erosion caused by rainfall moving across terrain), and many more factors. Generating the terrain this way, the procedural terrain becomes a reference point that prevents each chunk from getting too far away from continuity, even when erosion effects modify the terrain. Any differences introduced can be corrected with some post-processing of the terrain to allow the terrain to retain its improved look while also being contiguous.

With this in mind, the process for this report will use Perlin noise, a type of procedural noise, as the base for the terrain. Hydraulic erosion will be applied afterwards to improve the terrain's look. The edges will be corrected by first smoothing out the differences between the chunks, and then the chunks will be blended back on top to retain some of the detail originally found in the chunk.

Code Structure:

From the start, the project was designed to be as modular as possible, using abstraction to break sections into their core processes and reduce duplicate code. The code is written using C++, which enables the code to be split up into individual classes, keeping elements contained and restricting communication between classes only to use functions. This means that the content of the class can change, but as long as the function names don't, then the rest of the project is not affected. With the help of inheritance, this is taken a step further, allowing one class to copy the functions of another class but overriding their content. This limits repeated code and enables many different terrains to be handled at once without explicitly stating them. This flexibility allows for quicker prototyping and a more robust and versatile code base.

The designed flow of data through the program is as follows. The main class that the user interacts with is the Terrain Manager class. This is responsible for handling all user inputs and passing any relevant data to the terrain when needed. Terrains are intended to manage the chunks and decide whether chunks should be generated or deleted. This is also where the functions used for generating the heightmaps are stored. Storing the generation functions here allows the chunks to be simplified, only needing to store information relevant to that specific chunk, like the height map and position, along with some basic functions needed for rendering. This also enables the terrain class to be overwritten, enabling the single interface to be used to produce multiple different terrains.

Code Management:

During the development of the code, in order to keep track of code revisions and prevent loss of code, GitHub was chosen as a version control system. Snapshots of the code are uploaded along with a description of changes made and any known problems with that build. Unless otherwise stated, each build is a working version of the code base that allows it to be used in the event of a rollback or as a reference to how a function has changed since it was last working. The ideas and procedures discussed in this report were developed alongside a 3D rendering program. This program enabled the testing of the algorithms during development while also providing a way of benchmarking elements of the code to identify areas that could be improved.

Chapter 3
Implementation and Validation

Storage and Core System Flow:

Before any development was done the supporting systems needed to be developed and tested. This includes the how the user will interact with the terrain and any data structure needed to for storing data related to the terrain.

A heightmap is used to store the height of each point in the terrain. This is a two-dimensional array of numbers, where each number represents the height at that point. It is a compact way of encoding a point's position and height in three-dimensional space. It was decided to use floating-point numbers to store the height, allowing enough accuracy to keep details without needing too much memory, since the heightmaps can get extremely large depending on settings.

The general flow for generating terrain is as follows: The Terrain Manager receives input to generate a terrain of a specific type. This causes the Terrain Manager to create the terrain class that matches that type on a new thread and passes through the terrain settings. This thread is then added to a list for tracking later. Once the terrain class and the settings are transferred, the terrain will be independent of the Terrain Manager and start generating terrain.

There are three main stages needed when generating the terrain for a chunk. First, the heightmap must be created in memory and registered with OpenGL for rendering. Next, the heightmap is generated; this is done in a new thread to allow the terrain to handle multiple chunks at the same time. The exact process for generating the heightmap differs depending on the terrain type. Once the heightmap is finished being generated, the thread is closed, and the terrain proceeds to inform OpenGL of the new heightmap, allowing it to be seen on the screen.

Core system testing:

A simple terrain needed to be generated from start to finish to validate that the core system was working as intended. This ensured that all the connections between components were correct and that no data was lost or corrupted as it was passed through. If the terrain was rendered correctly to the screen, then all the intermediate components must be working.

Complex Terrains:

Now that the core system is proven to be working correctly, the next step is to start developing more complex terrains. Due to the complexity of these terrains, multiple steps are needed for more complex terrain generation; each chunk stores the index of the step it is currently processing. By storing this value, other chunks can now know the progress of the surrounding chunks, allowing them to share resources at specific points in the process.

Perlin Noise:

The base terrain that was chosen is fractal Perlin noise. The standard algorithm for Perlin noise is as follows:

- Generate a grid of points at a given size, e.g., 100x100. This grid must be equal to or smaller than the image size you wish to generate.
- At each point in the grid, generate a vector of size one pointing in a random direction. This will be referred to as VG.
- For each of the pixels in the image, repeat the following steps.
    - Calculate vectors from the closest four points in the grid to the pixel being checked in the image, then normalise the vectors. These will be referred to as VPs.
    - Calculate the dot product for each pair of VP VG and then interpolate between these values using the distance from the pixel to each point in the grid. The resulting value is the value used in the image.

For fractal Perlin noise, the process above is repeated with increasingly larger grid sizes and added to the image but with a decreasing weight so that the smaller details influence the image a smaller amount each time.

Fractal Perlin noise builds upon this process by layering Perlin noise each time the grid size and amplitude of the noise change. This allows for the small noise layer to construct the shape of the terrain while allowing the smaller noise to add details. Typical values for this are to scale the grid size by (base grid size * i) and the image weight by (1/i), where i is the number of iterations, also known as the octaves.

Two problems must be addressed to enable the Perlin noise to work with the terrain system. The first is that while we can use a normal random number generator, it would be impossible to get the adjacent grid to match since these values are not repeatable. The second problem is that ideally since the world is infinite, the grid size for the Perlin noise should be larger than a single chunk to allow terrain features to span multiple chunks.

Both problems can be addressed by changing the algorithm to a hash function that takes in a position in world space and returns a random number between -1 and 1. Since we can use world space to relate the point in the height map to the grid position needed for Perlin noise, we no longer need to store the points in a grid, and they do not need to share the same origin; this solves the second problem. The hash function used to do this is as follows:

sin((x^(3) *12.9898 + y^(3) * 78.233 + seed * 545.7145) * 1455.546) +

cos((x^(3) *12.9898 + y^(3) * 78.233 + seed * 545.7145) * 5640.465)) / 2

X and Y are the parts of the grid position to be sampled. The seed allows completely different terrains to be produced by changing a single value. The values of the constants were chosen at random with the intent to make a high enough frequency that even points chosen close together would have completely different values, making it sufficient to use as a random number generator. The Perlin algorithm can be altered as follows:

For each point in the height map:

- Find where the point is located in world space.
- Calculate the coordinates of the corners of the cell where the point is for the current octave. For each of these coordinates, use the hash function described above to produce random numbers, which can then be used to create an angle for the vector at each point.
- Using the world coordinates for the point and each corner, create vectors going from each corner to the point and normalise them.
- Calculate the dot product using each pair of coordinates and interpolate the same way as in the original Perlin noise algorithm.

Similar to fractal Perlin noise, this process can be repeated at different levels of detail to improve the effect.

This algorithm does take longer to compute than the original Perlin noise because it needs to compute the random vectors for each pixel rather than store them, but it reduces memory usage.

Fixing Lighting:

During the testing of the Perlin terrain, it was noticed that the way the terrain's normals are calculated leaves artefacts between the chunks, preventing them from being seamlessly connected. The solution to this problem was to increase the height map size by two and make the chunks overlap by a triangle in each direction; this allowed the normals to be generated correctly since it now knows the height of the terrain in the adjacent chunks. The overlapping terrain is cut off when rendering to prevent clipping, allowing for seamless chunks.

Hydraulic Erosion:

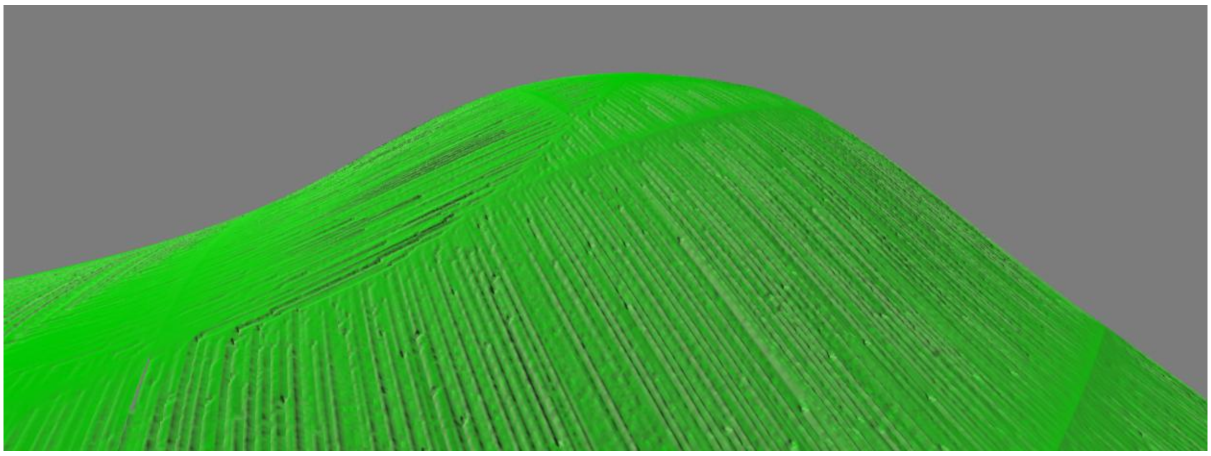
To improve the accuracy of the terrain, hydraulic erosion was applied. There are several different ways of performing hydraulic erosion. The process that was implemented was a combination of different algorithms and is detailed below:

Repeat the process below until a sufficient number of raindrops have been applied. A good starting point is around one raindrop per point on the terrain.

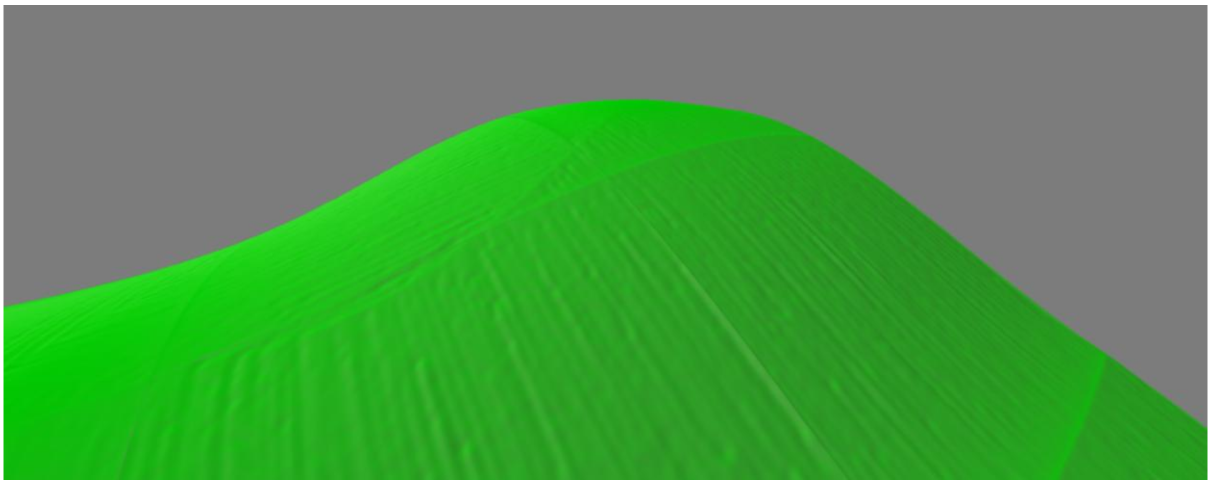
- Choose a point on the heightmap at random. This is the starting point for the raindrop. The raindrop starts with a capacity of one.
- Check the surrounding points to find the lowest neighbour; this will be the location where the raindrop will flow.
- If the water capacity is above zero, reduce the water capacity to simulate evaporation. The water capacity can never go below zero.
- Erode the terrain at the current point by the erosion factor, ensuring that the terrain is not eroded more than the lowest neighbour. Add the amount of terrain eroded to the sediment stored in the raindrop.
- Check to see if the amount of sediment stored in the raindrop exceeds the raindrop's capacity. The raindrop's capacity is a function of the maximum capacity of the raindrop multiplied by the water capacity; this means that as the raindrop evaporates, its capacity reduces. Any sediment over the capacity is deposited back into the terrain, making sure that the new height of the terrain does not exceed the highest neighbour.
- Deposit a small amount of sediment at the current rain position. This ensures that most sediment is evenly deposited back into the terrain and prevents too much sediment from being removed.
- The position of the raindrop moves to the lowest neighbour, and the process is repeated until both the water and sediment are empty or the maximum number of iterations is hit.

Terrain Smoothing:

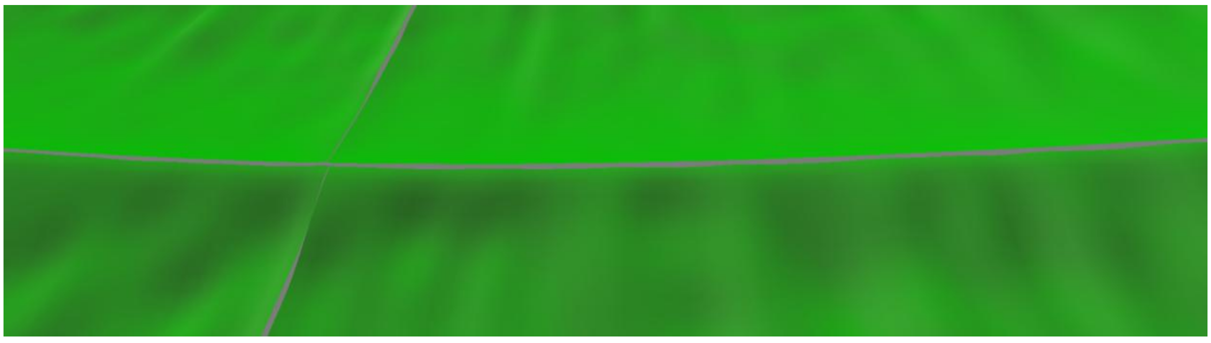
After applying the hydraulic erosion, the terrain's overall shape is closer to what you would expect to see in the real world. But in the process, it introduced artefacts to the terrain in the form of ruts left behind from the rain flowing down the terrain.



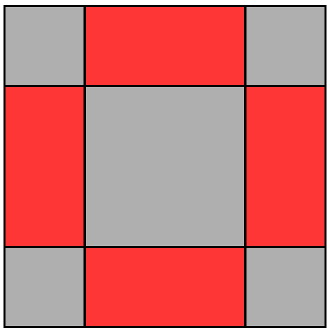
After trying a few different solutions, the one that gave the best results was simply to apply a smooth effect over the terrain a few times to even out the terrain and remove the worst of the artefacts while still keeping a believable amount of roughness to the terrain. This was done by iterating across each point in the terrain and taking the average of all adjacent points. The resulting value is saved to a new heightmap so that it does not affect the calculation of other points. The new height map is then copied back to the original at the end of each smoothing loop.



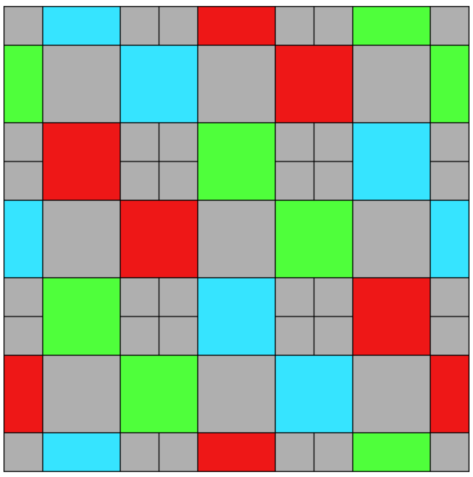
As a result of the hydraulic erosion and smoothing transformation, the chunks are no longer synchronised, causing holes to form across the terrain. This problem was expected to happen. The proposed solution is to have a two-step approach to smoothing out the transition between adjacent chunks to bring back the continuity.



The first smooth effect is for adjacent edges to bring them back in line. During this, the corners should be ignored, leaving the original terrain in place since they need to be handled differently. Area shown in red below.



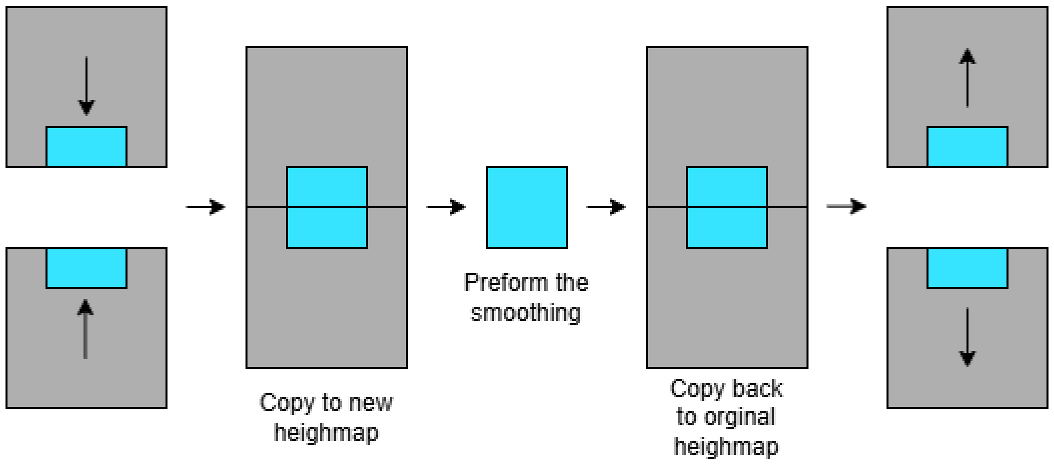
To allow the chunks to be processed in parallel, each chunk only processes the edges to the north and east of itself and leaves the other edges to adjacent chunks to process. This ensures that no two chunks are trying to work on the same section at the same time, removing the possibility of a race condition. The diagram below demonstrates how this would still cause every edge to be processed.



Before an edge can be smoothed, it first checks that the two chunks that share the edge have finished processing the Perlin noise and hydraulic erosion and are thus ready for smoothing together. The process used for doing this is detailed below:

- Create a new heightmap and copy the regen that needs to be smoothed to this new heightmap from each chunk.
- With this newly generated heightmap, perform a smooth over the terrain using the same method as before. Integrate over the terrain, and for each point, take the average of the surrounding points and save this value to a new height map. At the end of the iterations, copy the heightmap back to the original and repeat this process to get a suitable smooth between the terrain.
- Now that the smoothing is done, the heightmap can be copied back to the relevant chunk to apply the changes.

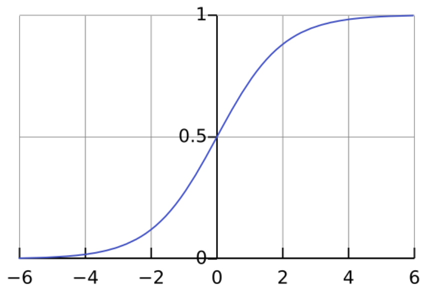
The diagram below demonstrates this process to help aid in what is being described.



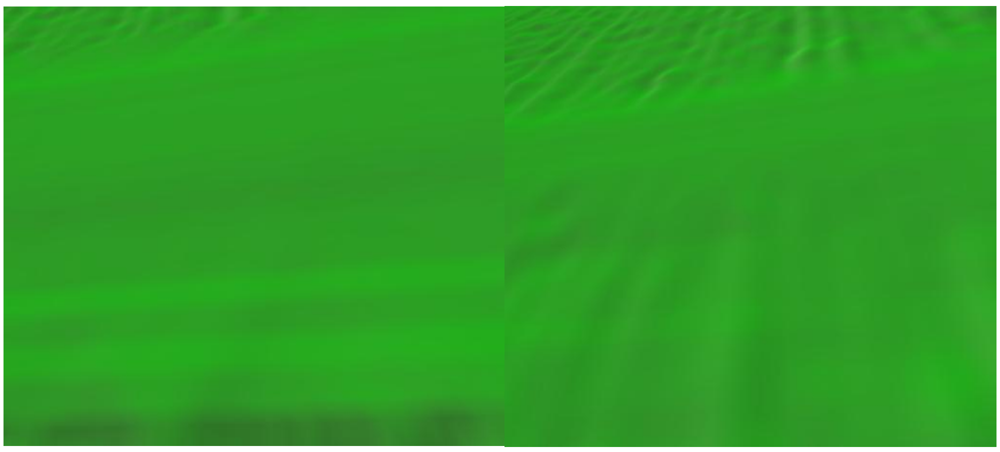
This process is good at creating a smooth transition between the chunks but results in a loss of detail, making the join stand out. To rectify this, we must add details from the original terrain to this newly smoothed terrain. So before copying the smoothed terrain back to the chunks, we add the following process:

Iterate over the smoothed heightmap and at each point:

- Compare the height of the smoothed heightmap and the original terrain and store the difference in height.
- Using a sigmoid function, multiply the difference in height by the function such that the weight is one at the edge of the heightmap, but in the centre, it is zero. This causes the terrain to blend into the join gradually.
  A Sigmoid function is a mathematical function that takes in any positive or negative value and returns a value between 0 and 1. Below is an example of a sigmoid function.



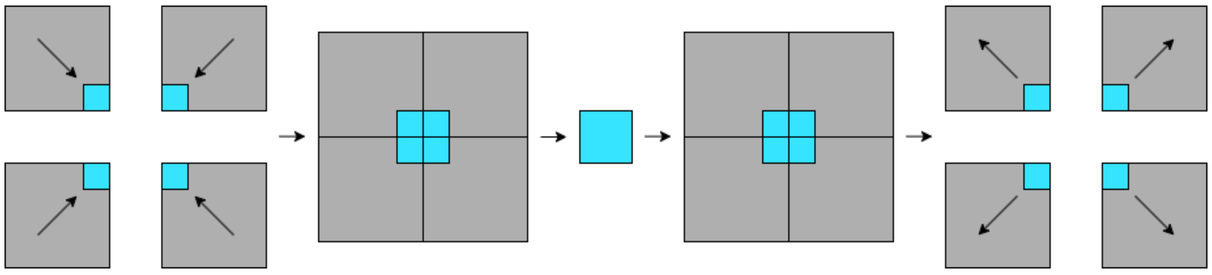
Below shows the terrain before and after applying the blending to the terrain. The left image is smoothing only, the right image is smoothing and blending.



Once the edges have been smoothed, the corners are next. The process is fundamentally the same, just with the corners. All four chunks that share the corner need to be smoothed at the same time. Similar to before, we wait for all four chunks to finish smoothing the edges before we can start processing the corner. A brief breakdown of the process follows, detailing where the process differs from the above:

- Copy the four corners of the to a new heightmap, keeping their relative positions.
- Apply the same smooth as detailed above.
- The concept for blending is the same, but now, instead of blending across a single axis, the blend needs to be applied on both axes such that each corner has a weight of one and the centre has a zero weight.
- Copy the newly smoothed terrain back to each corner.

With the terrain now smoothed out between the chunks, all the holes have been moved, and the terrain is back to being contiguous.

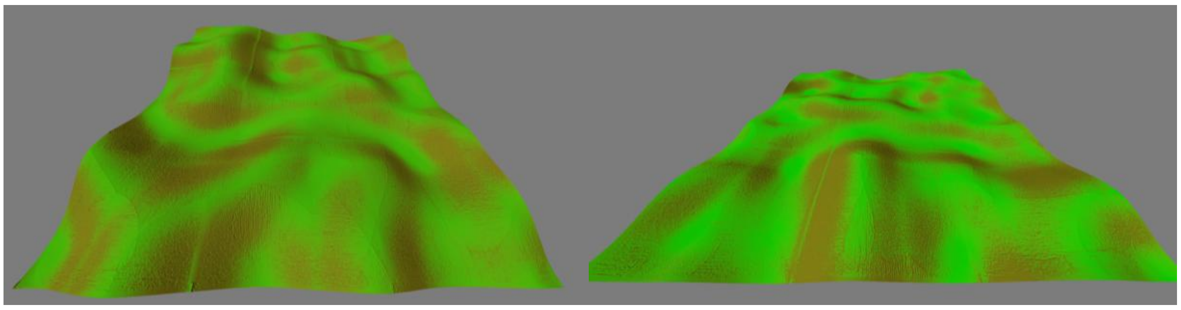


Chapter 4
Results, Evaluation and Discussion

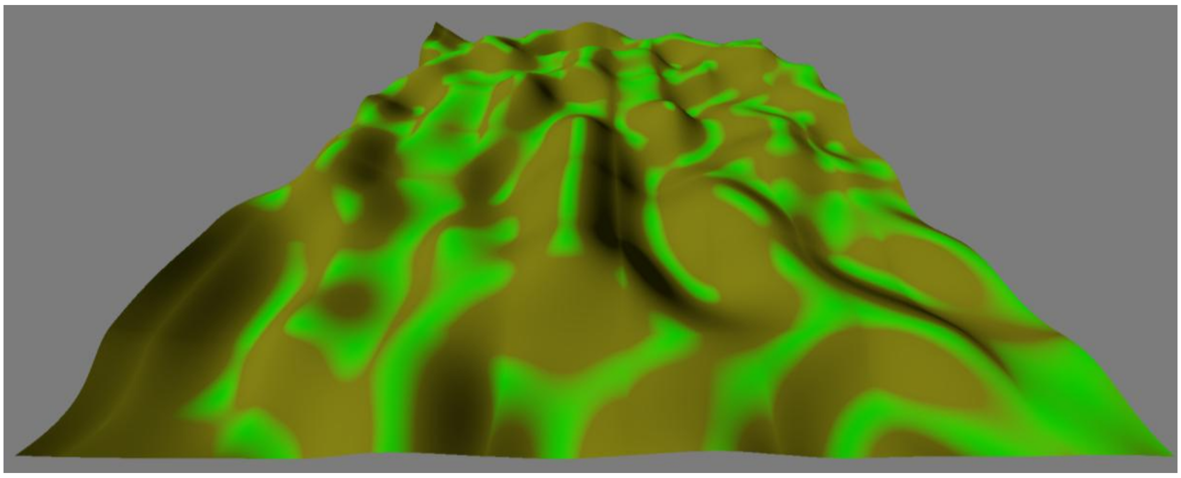
5.1 Evaluation

5.1.2 Terrain Appearance

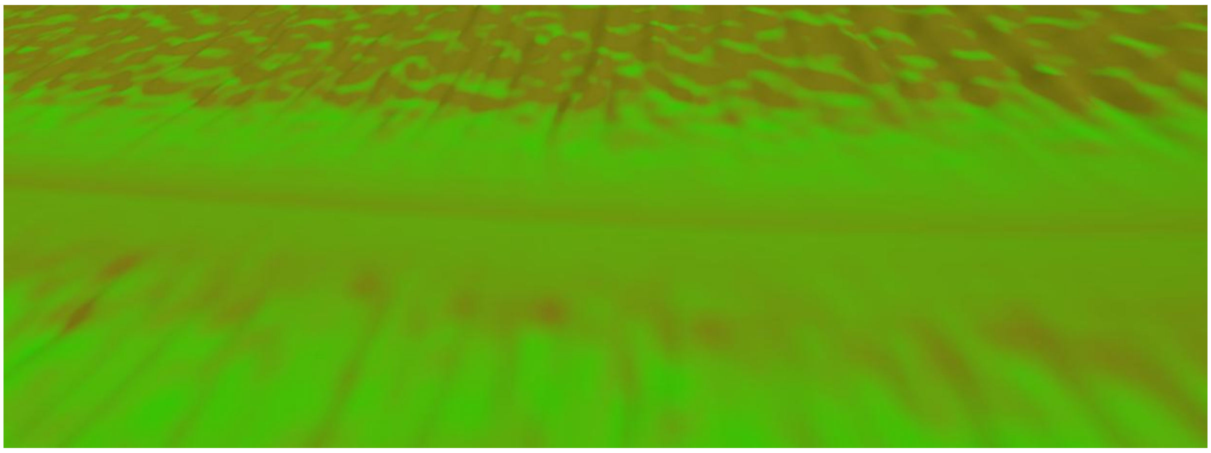
During this report, we have discussed a way of achieving procedurally generated terrain with a higher level of realism. A few examples of the resulting terrain are below.



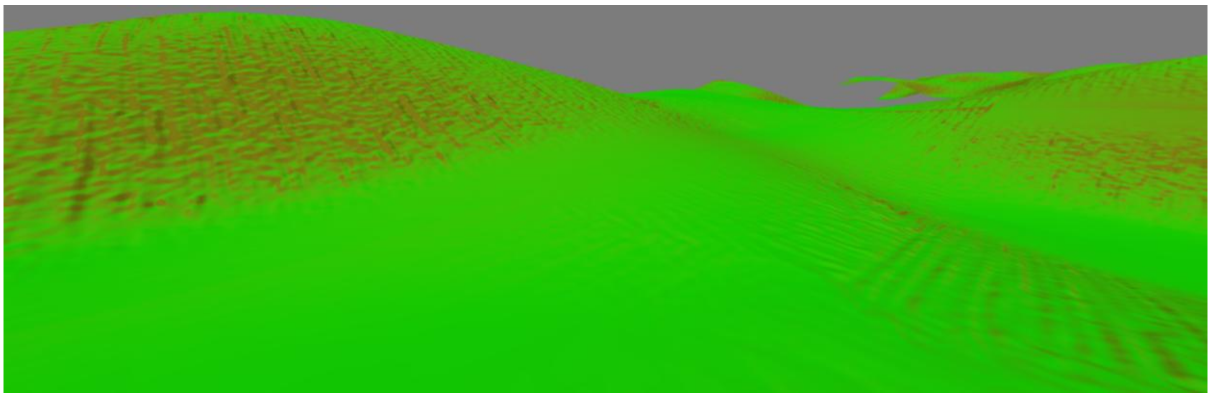
Compared to just using the base terrain Perlin noise, the resulting terrain has a more realistic look with smoother transitions between the peaks and valleys. The peaks become more defined and sharper, forming a ridge line as seen in real mountains. Similarly, the valleys created as a result of the rain deposits form flatter-bottomed valleys, which more closely resemble real valleys.



This is not to say that the terrain is without fault. When stitching the terrain back together after applying the hydraulic erosion, some artefacts still remain. Since the difference between the terrains is smoothed to make them fit back together, it causes the terrain to have a clear boundary between them. Even when you blend back some of the detail, this boundary persists, causing a clear line to form.



This line is not always as obvious, depending on the location and angle of the terrain at the joining point. Below is a boundary of the same terrain, just at a different location.



5.1.2 Performance

For the following performance stats, a 16-core CPU was used with a clock speed of 5.00 GHz. Each terrain generated a 7x7 chunk area, with each chunk generating a 250x250 heightmap. That makes a combined heightmap size of 1750x1750 or 3,062,500 points. Each chunk was processed in parallel on its own thread.

Generating a base Perlin noise terrain took an average of 0.153595 seconds to generate a chunk. Compared to the full terrain generation, including hydraulic eroding and post-processing of the terrain, it took 3.261012 seconds to generate a chunk. That’s an increase of 3.107517 seconds or a 2024% increase in time taken. So it's clear that adding these extra details comes at a massive cost in performance.

5.2 Conclusions

After reviewing the results, it's clear that this is not a perfect solution. As the terrain is now with its current problem, it's unclear whether it has a use case. But that’s not to say it's completely useless. This report showed that with some more work, this could be a viable solution for generating more detailed terrain without sacrificing world size.

With the performance as it is, low-end machines would have trouble generating the terrain with any speed, making it not viable to use in things like games, which are widely circulated across a large variety of hardware. This would be made worse when extra features are added on top of the terrain system to add actual gameplay elements.

To conclude, the methods used in this report have some potential to be used for real-time terrain generation with some additional work and optimisations. Below, I will be detailing some ideas that may help improve the terrain if implemented.

5.3 Future work

There is a lot of room for improvement in the methods discussed in this report, both in visuals and performance.

To improve the visuals, we need to first improve the results of joining the chunks. Two potential ideas to improve this are listed below:

The first idea was to add a layer of noise back to the boundary area to help break up the clear line that exists. This wouldn’t work for all terrain types and may cause more problems than it solves. But for Perlin noise, you could add a layer of noise equal to the lowest octave.

The other idea would be to increase the heatmap size so that they overlap more with each other. This would give you two points of reference in which you could perform some blending so that any difference between them is removed. This would give a better result than the current solution, but it has two clear downsides. Firstly, the cost of generating the extra terrain around each chunk would cause a noticeable hit in the performance, which is already less than desirable. The second point is that this introduces a large amount of wasted time and space since the overlapping areas will be disposed of after the chunks are finished generating, making this method rather wasteful.

These mentioned areas would be a good starting point for fixing the current problems with the terrain. But it will most likely decrease performance even more. Making it more important that the performance is increased. Using the current process, it is uncertain whether there is still a large amount of performance gain or whether a different erosion technique will be needed to get the desired performance.

List of References

1. Ryan RS Spick and James Walker, 2019, Realistic and Textured Terrain Generation using GANs, In Proceedings of the 16th ACM SIGGRAPH European Conference on Visual Media Production (CVMP '19). Association for Computing Machinery, New York, NY, USA, Article 3, 1–10. https://doi.org/10.1145/3359998.3369407
2. Jacob Olsen. 2004. Realtime Procedural Terrain Generation. https://web.mit.edu/cesium/Public/terrain.pdf
3. Jean-David Génevaux, Éric Galin, Eric Guérin, Adrien Peytavie, and Bedrich Benes. 2013. Terrain generation using procedural models based on hydrology. ACM Trans. Graph. 32, 4, Article 143 (July 2013), 13 pages. https://doi.org/10.1145/2461912.2461996
4. Hugo Schott, Eric Galin, Eric Guérin, Axel Paris, and Adrien Peytavie. 2024. Terrain Amplification using Multi Scale Erosion. ACM Trans. Graph. 43, 4, Article 145 (July 2024), 12 pages. https://doi.org/10.1145/3658200
5. Wikipedia, 2025, Perlin noise, [Online] [28th April 2025] https://en.wikipedia.org/wiki/Perlin_noise
6. Mariano, 2005, Valles Calchaquíes in Salta Province, [Online] [28th April 2025] https://commons.wikimedia.org/wiki/File:Salta-VallesCalchaquies-P3140151.JPG
7. Alpsdake, 2002, Ridge from Mount Otensho to Mount Tsubakuro, [Online] [28th April 2025] https://commons.wikimedia.org/wiki/File:Tsubakurodake_from_Otenshodake_2002-8-22.jpg
8. Qef, 2008, Sigmoid Function Plot, [Online] [28th April 2025] https://commons.wikimedia.org/wiki/File:Logistic-curve.svg
9. H. Zhou, J. Sun, G. Turk and J. M. Rehg, "Terrain Synthesis from Digital Elevation Models," in *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 4, pp. 834-848, July-Aug. 2007, doi: 10.1109/TVCG.2007.1027. https://ieeexplore.ieee.org/abstract/document/4293025
10. Wikipedia, 2025, *No Man's Sky*, [Online] [28th April 2025] https://en.wikipedia.org/wiki/No_Man%27s_Sky
11. Wikipedia, 2025, *Maze* (1973 video game), [Online] [28th April 2025] https://en.wikipedia.org/wiki/Maze_(1973_video_game)

Appendix A
Self-appraisal

A.1 Critical self-evaluation

As a whole, I think this project went quite well. The exact scope of the project did have to change as the project went on, mainly due to the original scope being too broad to achieve in the given time frame. Due to this, the areas of the project had to change, including how many different example techniques were explored.

In terms of time management of the project, this was handled relatively well, although more time should have been allowed for the development of the software. This is because implementing the techniques from the research papers turned out to be harder than I originally thought, leading to a lot more troubleshooting and development to get the desired result.

I did go into this with the hope that this would produce a more viable solution, but even with the extra time spent trying to fix the problem, it just wasn't possible to fix them all in time for submitting the report. This meant that the resulting solution was not the desired outcome, but the best it could have been, and thus I'm happy with the outcome overall.

A.2 Personal reflection and lessons learned

During this project, I was reminded that nothing will go as expected and that extra time should always be allowed for the unknown. I did know this going in, and thought that the project planned would be complete within the given time. So, going to the world, I should adjust my estimate of how long I think things will take. This will help me not have the problem of running out of time and needing to adjust what I plan to achieve in order to fit it within the time frame.

During this project, I had the opportunity to better learn different rendering techniques and interact with rendering API like Opengl. This was something that I had wasted time learn but never had a project I could implement it in. This knowledge can be used in future projects and in my career.

A.3 Legal, social, ethical and professional issues

A.3.1 Legal issues

Intellectual property: The use of third-party libraries, datasets, and algorithms may require certain licensing agreements to use. For instance, the use of publicly available data, such as NASA’s elevation data, requires the acknowledgement of the source to be included to ensure compliance.

Copyright concerns: When using code from the internet in your project, there may be problems where the code is not for free use and thus should not be included in the project. So it's important that all code used is your own work or used following the correct procedure, for example, referencing the origin of the code.

A.3.2 Social issues

User impact: Depending on the use case of the terrain generated, it could affect end users. To this end, a certain level of care must be taken to ensure the quality of the terrain produced so as not to disrupt the user.

A.3.3 Ethical issues

In terms of ethical issues, there isn’t much to mention since you aren't directly responsible for any user data. The only paramount problem that arises is that when using a data set, either public or private, you should only use it for the intended purpose, and you should not misuse the dataset.

A.3.4 Professional issues

Code Quality: There is a professional obligation to produce code that is of a high standard and quality to the best of your ability. Making your code easy to maintain and store correctly using a version control system is a must-do item for the longevity of any project.

Appendix B
External Materials

No external materials were needed during the development of this report.