Real-Time Procedural Generation in Games

Christopher James Murdoch

BSc (Hons) Computer Science

Final Year Dissertation

Supervised by Dr. Murdoch James Gabbay

Heriot-Watt University
School of Mathematical and Computer Sciences
Department of Computer Science

April 2022

Abstract

The global videogame industry is currently valued at over $138 billion. Countless hours are spent by artists and developers, designing exciting worlds for players to explore. Many videogame developers use *procedural generation* – the algorithmic creation of data – to automatically generate content without the need for manual design. However, due to current limitations in hardware, developers are often forced to choose between content *quality* and *generation speed*.

In this dissertation, I will explore the application of procedural generation to automatically generate infinite landscapes in real time, without sacrificing on quality. I will research and evaluate various procedural generation techniques and their current implementations in published games. I will use this research to design and implement a technical game demo that showcases high-quality, real-time generation of infinite landscapes.

Declaration

I, Christopher James Murdoch confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: 

Date: 20th April 2022

Acknowledgements

I would like to express my gratitude to the following people:

Firstly, I would like to thank my supervisor, Dr. Jamie Gabbay, for his guidance and support throughout this project. You have taught me how to express my narrative in academic work and your feedback has been invaluable in helping me to achieve this project.

I extend my gratitude to my second reader, Dr. Adam Sampson, for reading through my first deliverable and providing valuable feedback that has helped shape this dissertation into a professional piece of work.

I would like to thank my family, partner and close friends for their love and support throughout my degree. You have taught me, supported me and put up with me through the years and I owe you my sincerest gratitude.

Finally, I would like to thank my late grandad, David Gifford, for his love and inspiration throughout my childhood. You taught me to be curious and, without you, I would not be where I am today. Thank you.

Table of Contents

Chapter 1: Introduction

1.1 Motivation

The global videogames industry is expected to value $257 billion by 2025, with over 2.5 billion people playing games worldwide (Dobrilova, 2022). Minecraft – a popular, procedurally generated building game – has sold over 200 million copies since its launch in 2011 (Curry, 2022). More than ever before, consumers have proven their demand for virtual worlds to which they can escape.

Procedural generation allows content creators to focus less on the manual creation of individual results, but more on the desired *properties* of those results. Instead of having to manually sculpt hills and carve rivers into a landscape, a creator can tell the algorithm how to do so, and let it automatically generate limitless instances with the desired properties.

Using procedural generation to create assets can save creators time and resources, but it can also allow them to create things that wouldn't otherwise be possible. Using traditional, manual methods of content creation, infinite worlds would require infinite work. However, with procedural generation creators need only define the properties of an infinite world, and let the algorithm dynamically generate the player's environment on the fly.

Procedural generation is not a universal solution, however; it has multiple limitations:

1. Procedurally generated content can lack the high-level coherency that well-thought-out, handcrafted content often provides.
2. With too few variable properties, content can be repetitive and boring.
3. Procedurally generating content during runtime relies heavily on the player's hardware and can slow down or degrade game performance.

1.2 Aim

The aim of this project is to research, design and develop a game demo that shows how procedural generation can be used to generate realistic, 3D terrain in real time, without sacrificing on content quality. A working final demonstration allows a player to generate and load into a 3D world where they can explore an infinite landscape of diverse biomes.

The final demonstration's source code is hosted on GitHub, available to view here. Additionally, a playable demo is available to download in the repository's releases page.

A guided video of the final demo is available on YouTube. The video showcases each biome individually, demonstrates the generation of infinite terrain, and shows the performance metrics during various scenarios.

1.3 Objectives

In this dissertation, I set out to achieve the following objectives:

1. Research different methods and applications of procedural generation in real time.
2. Design and plan the development of a technical demo that generates terrain in real time.
3. Carry out the implementation of the designed system, documenting difficulties or issues encountered along the way.
4. Test and evaluate the developed system using technical benchmarks.

Each of these objectives were achieved and reflection on them can be found in Section 9.1.

1.4 Document Organisation

This document is divided into nine sections:

1. **Introduction**: introduction of the project, aims, and document organisation
2. **Background**: research of procedural generation techniques and applications
3. **Requirements Analysis**: identification of the primary functions of the application
4. **Design**: technical planning of the application's functionality
5. **Evaluation Strategy**: definition of the metrics used for assessing the final application
6. **Project Management**: overview of how the project and associated risks were managed
7. **Implementation**: chronological section detailing the implementation of the project, as well as some of the issue encountered along the way
8. **Evaluation**: Technical evaluation of the system using previously defined metrics
9. **Conclusion**: discussion of the achievements and limitations of the project, as well as further improvements that could be made in the future

Chapter 2: Background

The algorithms used to procedurally generate content can be split into two subcategories: *Ontogenetic* and *Teleological* (Wikidot, 2020). In a nutshell, Ontogenetic algorithms are fast but rough *approximations*, and Teleological algorithms are slow but accurate *simulations*.

Ontogenetic

Ontogenetic algorithms seek to best approximate a target result, often using few or no intermediate steps. These algorithms are usually fast, scalable and easy to implement, but struggle to produce features that arise from complex emergent properties of physical systems.

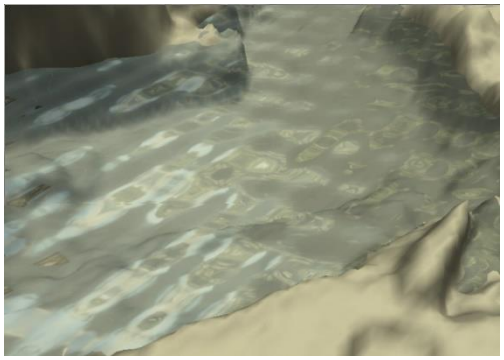


Figure 1: An Ontogenetic algorithm using sine waves to emulate moving water (Conrod, 2010)

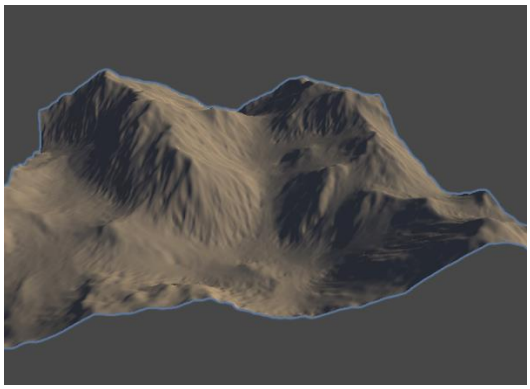


Figure 2: A Teleological algorithm using rainfall to simulate an eroded landscape (Lague, 2019)

Teleological

Teleological algorithms target a result, in the context of the physical systems that create it. By modelling systems like plate tectonics, rainfall and evaporation, features such as canyons, rivers and deltas can be naturally created.

Ontogenetic vs Teleological

Teleological generation algorithms are slow but accurate; they are best suited where high-quality results are needed, and time is not a significant issue. Ontogenetic algorithms are faster but less accurate, making them useful in time-critical situations such as real-time content generation.

Another distinction is that, due to their simulation-based nature, Teleological algorithms often must simulate all of the content at once. Because a raindrop falling at the top of a mountain may affect the state of the land at the bottom, that state can't be ascertained until everything around it has been simulated. Although some Ontogenetic algorithms also have this problem, most do not. This means that Ontogenetic algorithms such as Perlin noise and domain warping can be used on infinite worlds, independently-generating just one, finite section at a time.

2.1 Perlin Noise

Perlin Noise (Perlin, 1985) is an Ontogenetic gradient noise algorithm capable of generating content in an arbitrary number of dimensions. It was originally developed by Ken Perlin to generate realistic visual textures for Disney's 1982 movie, Tron.

Perlin Noise returns just one 'sample' of the desired content at a time. If a texture with a resolution of 100 by 100 pixels were to be generated, the function would be individually called 10,000 times – once for every pixel.

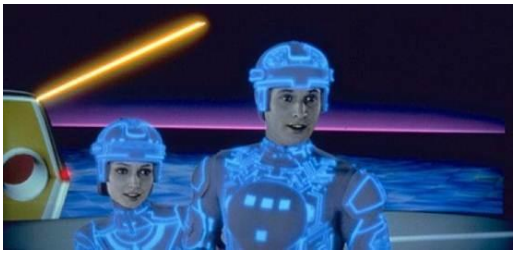


Figure 3: Procedurally-textured landscape in Tron (Vivo, Noise, 2021)

The function provides an explicit mapping from sample position to sample value. This means that, provided with a position, the function will return the value of noise at that location. Because samples can be taken independently like this, Perlin Noise is particularly well-suited to infinite terrain generation.

2.1.1 Perlin Noise Implementation

Perlin Noise uses an n-dimensional grid of pseudo-random vectors to allocate areas of high and low sample values. To calculate the value of a sample, the dot product of each surrounding vector and their sample-relative position is taken (Figure 5), and the final value is calculated as a linear interpolation of these dot products.

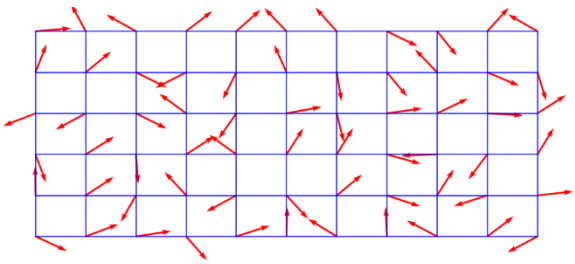


Figure 4: 2D pseudo-random vector grid (Tatarinov, 2008)

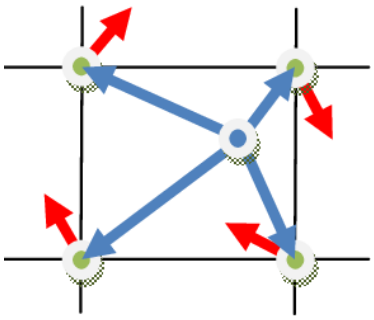


Figure 5: Pseudo-random vectors (red) and their sample-relative position vectors (blue) (Tatarinov, 2008)

The linear interpolation of the dot products is just a blend of the values where physically closer positions have more of an influence on the final output.

A disadvantage of Perlin Noise is that it naturally produces gradient discontinuity along the gridlines (Perlin, 2002). This means that textures or terrain generated would have square-like visual artefacts if the discontinuity is not handled.

2.1.2 Smoothing

In order to rectify the gradient discontinuity along the gridlines in Perlin Noise, a smoothing function is used. This function slightly alters the linear interpolation of grid vectors to produce much smoother, continuous values. Modern implementations of this smoothing use a Quintic function, Smootherstep (Perlin, Improving Noise, 2002):

$$smootherstep(x) = 6x^5 - 15x^4 + 10x^3$$

Because both the first- and second-order derivatives of this function have roots of 0 and 1, it ensures that both the values and gradients of the noise produced are continuous over gridlines.

2.1.3 Simplex Noise

Simplex Noise (Perlin, 2002, pp. 16-17) is an improved version of Perlin Noise with fewer visual artefacts and better dimensional complexity. It achieves this by using a grid of n-dimensional triangles called simplices.

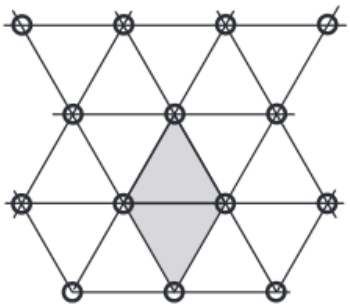


Figure 6: 2D simplex grid (Tak, 2015)

By using triangle-based grids instead of square ones, the number of vertices calculated per dimension reduces exponentially. While n-dimensional square grids require 2^n vertices for a sample, triangle-based simplex grids require only n+1.

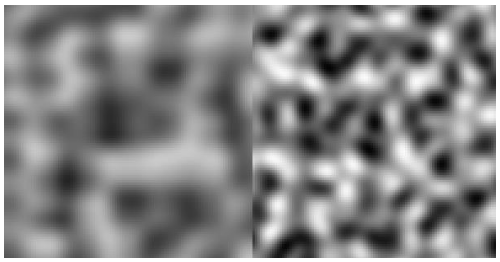


Figure 7: Perlin (left) vs Simplex(right) Noise (www.bit-101.com, 2021)

Another benefit of using Simplex grids is that they naturally produce continuous gradients along the gridlines. This means that the results are smoother, clearer, and don't require the use of a smoothing function.

2.1.4 Augmentation

Perlin and Simplex Noise can be altered to produce interesting effects:

1. Taking the absolute value of a sample for bubbly noise
2. Inverting bubbly noise to generate a ridged effect
3. Rounding sample values to get a stepped pattern

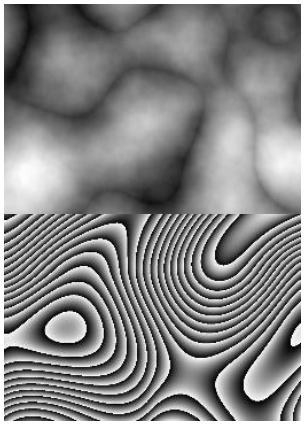


Figure 8: Ridged and Stepped noise (Li, 2018)

2.2 Fractal Brownian Motion

Consider the specific problem of generating realistic terrain. Features like mountains, hills, boulders and rubble should all be represented at different scales. **Fractal Brownian Motion** offers a generalisation for creating these features by combining multiple layers of varying frequencies and amplitudes.

The frequency of a noise function governs how quickly the output can switch between high and low values. For hills, a low frequency is required to represent slow change over several kilometres but, for rocks, the frequency should be high to reflect significant movement over only a few centimetres.

Though rocks 'jitter' much faster than hills, they represent a much smaller portion of the terrain's final height. Where rocks contribute a few centimetres of height for a given point, hills can represent several kilometres.

Fractal Brownian Motion (Vivo, Fractal Brownian Motion, 2015) generalises the combination of the different layers by iteratively adding layers with progressively higher frequencies and lower amplitudes. Two variables control these shifts in frequency and amplitude; lacunarity and persistence.

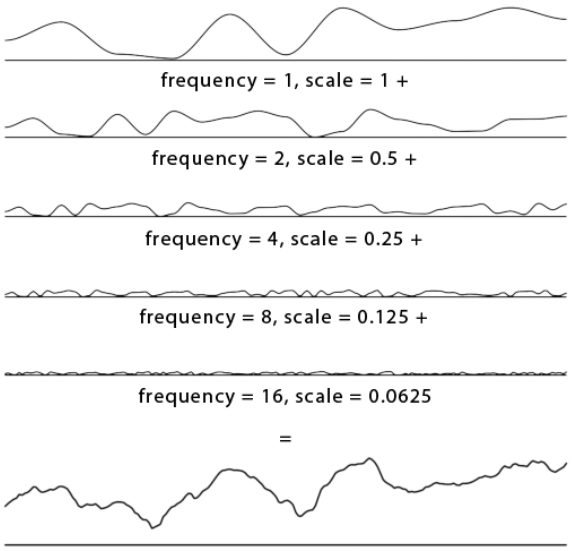


Figure 9: Combination of multiple noise layers (Scratchapixel, 2021)

Lacunarity is usually set to around 2 and persistence is around 0.5. This means that, if the frequency and amplitude are respectively multiplied by the lacunarity and persistence for each new sample, finer and finer levels of detail will be added to the result. Once the desired iterations of noise sampling have been completed, the samples are all added up into one, fractally-layered sample.

The number of sampling iterations is variable but, in order to get good results, it should be high enough that any more samples would have insignificantly low amplitudes.

The lacunarity and persistence can be slightly altered to receive different results; higher lacunarity results in more drastic changes in features like hills, whereas an increase in persistence results in small features like rocks having more of an effect on the resultant landscape.

2.3 Domain Warping

A feature often seen in real-life landscapes is warping. Whether it's sand dunes that have been moved by wind, cliffs shifted by glaciers, or hills carved by rivers; nature often warps terrain out of its original shape.

Domain Warping (Quilez, 2002) is a way to achieve this effect in an Ontogenetic, explicitly-mapped manner; this means we can use it for infinite terrain, where a section of land must be generated without knowledge of all its neighbours.

The term 'Domain Warping' comes from the fact that the method achieves a warped output by warping the domain of the function itself. The function takes in the position of the desired sample, moves that sample by a small amount, and then returns the value of sampling at the new, distorted location:

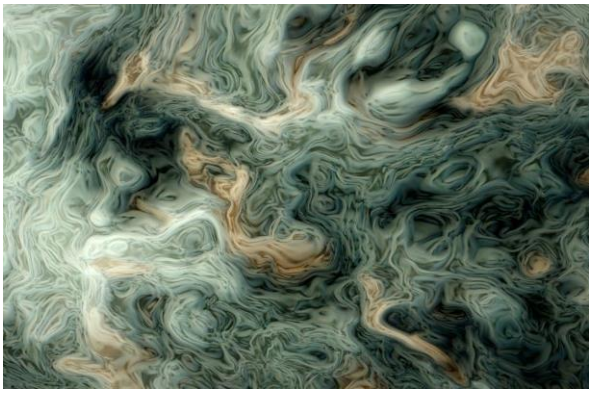


Figure 10: A marble texture generated using Domain Warping (Quilez, 2002)

```
static float domainWarp(float x, float y, function noise) {

    // Use noise function to slightly move sample position
    float xWarp = noise( x,  y);
    float yWarp = noise(-x, -y);

    // Sample at new, warped position
    return noise( x+xWarp, y+yWarp );
}
```

As can be seen in the implementation above, Domain Warping provides a simple way to add more detail and complexity to the generated content. A drawback to this method, however, is that the noise function must be called three separate times for one sample. This is fine for simple noise functions such as 2D Perlin Noise but, for more computationally expensive functions such as fractally-layered 4D Perlin Noise, Domain Warping poses a significantly expensive augmentation.

Because the warping of content is not generally as visible as the content itself, optimisations can be made to speed up the performance of warping without sacrificing noticeable content quality. A simpler noise function can be used to warp the sample location before a higher-quality function is used to sample the calculated location. This means that warping a sample could result in two samples of standard Perlin Noise and one sample of 10-fractal-layer Perlin Noise.

2.4 Content Depth: No Man's Sky

The development of No Man's Sky began in 2011. Trailers and demonstrations of the game showcased a vast, procedurally-generated universe with over 18 quintillion planets to explore; each with their very own ecosystem of unique inhabitant wildlife. Players could fly anywhere in an almost-infinite universe to explore strange landscapes and discover new species.

2.4.1 Gameplay

When the game launched in 2016, it was met with an overwhelmingly negative reception (Metacritic, 2021). Out of 18 quintillion planets, there were only a handful of unique variations. The procedurally-generated species looked less like the product of a complex, biologically-inspired procedural generation algorithm and more like a sliding scale between dinosaur and giraffe. Yes, there were limitless worlds to explore – but they weren't fun.

> *"... each new planet was little more than a minor change of clothes surrounding the same exact things I had already experienced on every other planet."*
>
> - *Philip Kollar, Polygon* (Kollar, 2016)

Figure 11: A procedurally generated creature in No Man's Sky (Michael, 2016)

2.4.2 Summary

Although No Man's Sky has improved significantly since 2016, it demonstrates that the depth and diversity of procedurally generated content is just as important as the scale. Near-infinite worlds are always going to repeat, but the procedural-generation engine should have a diverse enough set of variables and features to produce a full-game's worth of content before players become bored with the repetition.

2.5 Content Depth: Dwarf Fortress

Dwarf Fortress is a colony-management game released in 2006 that uses procedural generation to generate realistic, engaging landscapes. The game uses a complex series of Ontogenetic and Teleological steps to create a world that feels lived in (Hall, 2014).

2.5.1 Landmass

In Dwarf Fortress, the first step in world generation is building a basis for the map. A layer of fractal-layered gradient noise, hereafter referred to as a fractal layer, is generated for the height values across the world. This defines what parts of the land are high up in the mountains and what parts are deep under the sea.

Figure 12: Small island generated in Dwarf Fortress (bay12games.com, 2021)

2.5.2 Validation

Once the heightmap has been generated for the world, a gameplay-balancing algorithm determines whether or not it's suitably diverse enough to play on. If the heightmap is deemed to be too high, too low, too flat or just generally unsuitable for playing on, it's scrapped, and world generation starts over again. This conditional rejection continues until the algorithm is happy with the generated heightmap.

2.5.3 Natural Elements

Once a suitable heightmap has been created, fractal layers for annual rainfall, soil deposition and temperature are generated. These layers are not seen by the player and have no direct impact on gameplay; they exist to enhance the realism of the next step.

2.5.4 Weather

With annual rainfall, soil deposition, and temperature maps generated, the game starts to simulate rain falling and eroding the landscape. This creates the rivers, canyons and deltas that make the world feel real. Once the rain has fallen, a salinity map is generated to help define the natural biomes that make up the world.

2.5.5 History

The next step in Dwarf Fortress' world generation is outside of this project's scope. However, it does well to show how Teleological methods can create genuinely real and interesting features for players to discover.

Once the physical world has been generated, and before the player starts their game, thousands of years of history are played out on the landscape. AI characters explore, build civilisations, form diplomatic relations and wage war upon each other – all before the player has begun their campaign. This means that, when the player salvages or sets up camp in the ruins of an ancient civilisation, it's an actual ancient civilisation. Previously established trade routes may be reopened, abandoned gold mines rediscovered; the world shows signs of genuine life.

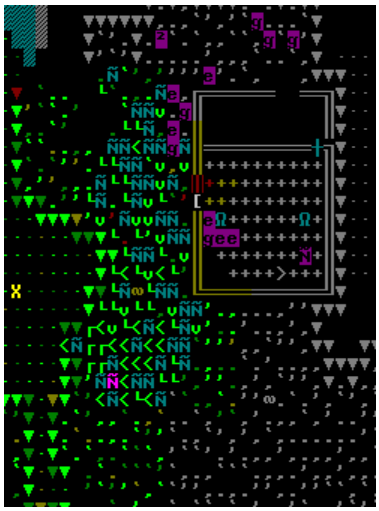


Figure 13: AI-built tower in Dwarf Fortress (Reddit, 2016)

2.5.6 Summary

Dwarf Fortress is a vastly complex game, and this section has only touched on a relatively small amount of the techniques used to generate realistic worlds. Furthermore, generating worlds in Dwarf Fortress is no small task. Larger-sized worlds with long periods of simulated history can take upwards of 20 minutes to generate.

Despite the computational expense of using such in-depth simulation for world generation, Dwarf Fortress demonstrates the vital part that more convoluted, nature-based factors can have in generating lifelike content. The technical demonstration developed in this project should therefore aim to achieve Dwarf Fortress' realism by considering natural factors such as temperature, rainfall and soil deposition.

2.6 Content Depth: Discussion

The previous sections, 2.4 and 2.5, outline how significantly procedurally generated game worlds are affected by diversity of content. With thoughtful and careful design of natural properties, games like Dwarf Fortress can express more life and diversity through ASCII art than games such as No Man's Sky can with modern, AAA graphics.

To ensure that my technical demonstration achieves a similar level of content diversity and realism to Dwarf Fortress, and to avoid the lack of depth initially suffered by No Man's Sky, my system should incorporate geographically inspired properties such as humidity, temperature, and soil deposition.

If time permits, incorporating teleological algorithms such as fluvial or glacial erosion into the technical demo could significantly improve the quality of terrain generated. Unlike Dwarf Fortress, however, these teleological algorithms would need to be performed in real time and would need careful consideration as to how chunks interact with each other while loading (later discussed in 4.4: Teleological Enhancements).

2.7 Technology: Game Engine

This section outlines the research-based design choices made for the engine/graphics API used in the demonstration. Reflection on this choice will be made in later section 7.2.3.

2.7.1 Graphics APIs

To process and render 3D graphics to a screen, a graphics API such as OpenGL or Vulkan must be used to leverage the GPU. These APIs are implemented by the GPU manufacturer and handle tasks such as memory movement, hardware-specific shader compilation, and frame management.

Most modern graphics APIs provide little more abstraction over hardware than is necessary. The developer is responsible for managing low-level tasks such as memory allocation, buffer synchronisation, and writing shader code that runs on the GPU. Graphics APIs can be very powerful for achieving low-level optimisations, but development with them is slower and more difficult than higher level alternatives such as game engines.

2.7.2 Game Engines

Game engines provide a high-level abstraction over graphics APIs, often including extra features that are useful for game development such as physics, animation and sound. Game engines like Unreal, Unity and Godot host a variety of functionalities that enable creators to develop their games without worrying too much about trickier, low-level programming.

Because game engines are higher level, development with them is often much faster and easier, allowing developers to focus more on the content they create. The downside to these high-level abstractions, however, is that low-level performance optimisation can be much more difficult – if not impossible – to make.

2.7.3 Graphics API vs Game Engine

Graphics APIs are difficult and slow to use, but offer higher performance-tuning potential. Game engines are easier and faster to use, but some optimisations can be difficult or impossible to implement.

Given more time to develop my application, I would have chosen to use a graphics API such as Vulkan or OpenGL 4. Procedural generation in real time is very performance dependent and optimisations such as manual memory movement and interoperability between compute and graphics processes on the GPU can make a huge difference.

However, development of this application was scheduled to last only two months – too short to implement a rendering engine from scratch. Additionally, extra features such as animation, sound and physics would have to be programmed, taking development time away from more important features.

For these reasons, I decided that using a game engine would be better as it would allow me to focus more on the high-level design and development of the application.

2.7.4 Game Engine Comparison: Table

There are many different game engines, each with their own strengths, weaknesses and niches. I have created a table listing a few of the more popular ones, along with relevant features of each:

Engine	Unreal Engine 4	CryEngine	Unity 5	RPG Maker
Animation, Sound, Physics	Yes	Yes	Yes	Yes
3D capable	Yes	Yes	Yes	No
Programming Language	C++	C++, C#, LUA	C#	JavaScript
Community Size*	27%	3%	72%	<1%

**Developer community size is measured as how many game developers currently use this engine. Data from Statista* (Clement, 2019).

2.7.5 Game Engine Comparison: Features

Like the majority of game engines, the ones I researched all had support for basic features such as animation, sound and physics. This meant that I could spend less time reinventing the wheel, and more time developing novel functionality unique to my application.

2.7.6 Game Engine Comparison: 3D Capability

Unreal, CryEngine, and Unity all have full support for 3D in terms of graphics, physics and animation. RPG Maker doesn't have native support for 3D graphics, however. Though there are plugins that allow for basic 3D rendering, the engine does not compete with the performance and functionality of the other three. This means that RPG Maker was not suitable for use in my application, and was therefore excluded from further consideration.

2.7.7 Game Engine Comparison: Programming Language

Although many game engines can be used entirely without programming, many offer the ability to natively run programming languages when the developer wants more low-level control.

Unreal Engine uses C++ for programming in-engine. C++ is a relatively low-level, highly performant language providing the developer with access to low-level operations such as memory management and move semantics, and has good interoperability with hardware interfaces such as Cuda and vectorisation.

Unity uses C# for programming in-engine. C# is a relatively high-level programming language with faster development time and good scalability. C# has automatic memory management using garbage collection. However, this can decrease runtime performance in specific cases where developers need fine-grained control over memory operations.

Lastly, CryEngine offers a choice of languages to use in-engine: C++, C# or LUA. LUA is a very high-level scripting language that, whilst powerful for light scripting, is not performant enough to run performance-intensive operations using. C++ and C# both run natively in CryEngine and the choice allows developers to pick between faster development or more performant operation.

Because my application required a significant amount of performance tuning to run in real time, I decided that Unity with C# was not suitable for this specific project.

2.7.8 Game Engine Comparison: Community Size

Unreal Engine and CryEngine are very similar engines with very similar offerings. Both provide good functionality with performance-oriented programming languages.

From experience, I have found that the difficulty of overcoming problems when learning a new language or technology is largely dependent upon the amount of documentation, help and discussion available online. This tends to correlate strongly with the size of the active developer community.

Looking at Statista's (Clement, 2019) data, it's clear that Unreal Engine has a large community consisting of around 27% of game developers, second only to Unity. CryEngine on the other hand, is only used by around 3% of developers.

For this reason, I decided to use Unreal Engine for the development of my application. The high-level functionality of a feature-rich game engine with the low-level performance of C++ strikes a good balance for development of the app within the given timeline. Coupled with a large and active developer community, problems were relatively easy to fix using help from documentation and online forums.

2.7.9 Unreal Engine 4

Unreal Engine 4 was released in 2014 and has been stable for many years now, with a large amount of published games developed using it. Unreal Engine 5 has very recently been released in alpha, but is not recommended for commercial use as it is not yet stable.

Although Unreal Engine 5 provides many extra features and a large performance boost in graphics, I opted not to use it. This is primarily because the use of experimental functionality introduces unnecessary risk of bugs or data corruption.

Chapter 3: Requirements Analysis

This section details the functional and non-functional requirements of the technical demonstration using MoSCoW prioritisations:

MoSCoW Priority	Description
Must	Requirements vital to the success of the system
Should	Requirements that are optional but provide high value to the system
Could	Requirements that add value to the system if time permits
Won't	Requirements that would add value, but can't be implemented in the timeframe

Additionally, I have colour-coded and added a column since the original plan, specifying if the requirement was fulfilled and, if so, in which section it was implemented.

Requirement fulfilled	Requirement not fulfilled	Other

3.1 Functional Requirements

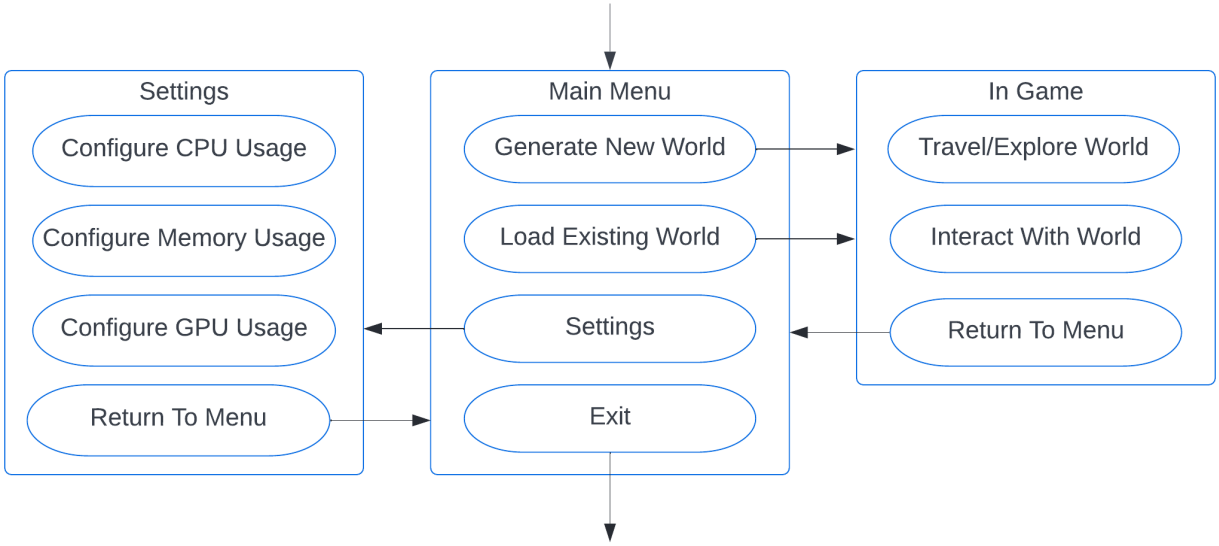
Requirement	Description	MoSCoW	Completed?
FR-1	The system must be able to create a new world with physical 3D terrain	Must	Yes: 7.2.4
FR-2	The user must be able to see the world in first or third person	Must	Yes: 7.3.5
FR-3	The user should be able to walk on and travel the generated terrain	Should	Yes: 7.3.5
FR-4	The system should be able to display framerate metrics	Should	Yes: 7.2.2
FR-5	The system could allow for customisation of hardware usage (CPU cores, memory, GPU acceleration)	Could	Redundant. See: 7.4.3
FR-6.1	The landscape should have at least three distinct biomes	Should	Yes: 7.5.2

FR-6.2	The landscape could have at least five distinct biomes	Could	No
FR-7	The system should have basic gameplay mechanics such as combat, exploration or building	Should	Yes: 7.3.5
FR-8	The system won't have advanced gameplay mechanics such as AI, multiplayer or story progression	Won't	N/A
FR-9	The user won't be able to change the world's physical landscape	Won't	N/A

3.2 Non-Functional Requirements

Requirement	Description	MoSCoW	Completed?
NFR-1.1	The system must have load times less than 1 minute	Must	Yes 8.6
NFR-1.2	The system should have load times less than 30 seconds	Should	Yes 8.6
NFR-2.1	The system's average framerate must be at or above 30Hz	Must	Yes 8.7
NFR-2.2	The system's average framerate should be at or above 60Hz	Should	Yes 8.7
NFR-3	The landscape should be effectively infinite	Should	Yes 7.2.7
NFR-4	The landscape could be enhanced with Teleological generation	Could	No
NFR-5	The system should use multiple CPU cores for generation	Should	Redundant. See: 7.4.3
NFR-6	The system could use the GPU to accelerate generation	Could	Redundant. See: 7.4.3
NFR-7	The system could use custom shaders for visual effects	Could	Yes: 7.3.4
NFR-8	The system should provide a basic GUI for world generation	Should	No
NFR-9	The system could cache world data for faster loading	Could	No

3.3 Use Case Model



When the technical demo is launched, the user will be met with a main menu providing options for creating a new world, loading an existing one, or changing the settings.

If hardware usage customisation (**FR-5**) has been implemented, clicking on settings will provide the user with a menu where they can configure options such as number of CPU cores used, GPU enablement, and memory usage. Clicking 'Return' will place them back at the main menu.

The user will be able to generate a new world from the main menu. After clicking it, they will be met with a loading screen where they should wait no more than 30 seconds (**NFR-1.2**). Once loaded, the user will enter the generated world as a first- or third-person character, free to explore the infinite world.

Once the player wants to leave, they can return to the main menu with a button. If persistent worlds (**NFR-9**) have been implemented, the exited world will be saved in storage. If the user clicks on load world, the saved world will then be entered after a short loading screen.

3.4 Use Cases

Use Case	**Generate new world**
Actor	User
Description	The user wants to generate a new world in which to play
Precondition	User is on the main menu
Trigger	User clicks on 'Generate World' button in the main menu
Main Flow	1. User clicks on 'Generate World' button 2. User may select world generation options 3. System shows loading screen as initial chunks are loaded 4. User enters the new world
Postcondition	After a short wait, the user is now in a newly-generated world, ready to play

Use Case	**Go to main menu**
Actor	User
Description	The user wants to return to the main menu
Precondition	User is in a game world
Trigger	User clicks on 'Return to Menu' button in the game menu
Main Flow	1. User clicks on 'Return to Menu' button 2. System asks: 'Are you sure? Progress will be saved' 3. User confirms 4. World state is saved 5. User returns to main menu
Alternate Flow	1. User clicks on 'Return to Menu' button 2. System asks: 'Are you sure? Progress will be saved' 3. User declines 4. User remains in world
Postcondition	Main Flow: The user is now on the main menu Alternate Flow: The user is still in the game world

Use Case	**Load existing world**
Actor	User
Description	The user wants to load an existing world
Precondition	User is on the main menu
Trigger	User clicks on 'Load World' button in the main menu
Main Flow	1. User clicks on 'Load World' button 2. System shows loading screen as initial chunks are loaded 3. User enters the world in the same state that they left it (player position, inventory etc.)
Postcondition	After a short wait, the user is now in their world, ready to play

Use Case	**Change settings**
Actor	User
Description	The user wants to change their settings
Precondition	User is on the main menu or the game menu
Trigger	User clicks on 'Settings' button in either menu
Main Flow	1. User clicks on 'Settings' button 2. The user changes checkboxes or sliders to their desired settings 3. The user clicks 'Back' 4. System applies settings 5. User returns to the menu in which they started
Postcondition	The settings have now been changed and the user returns to their initial menu

Use Case	**Exit**
Actor	User
Description	The user wants to exit the game
Precondition	User is on the main menu
Trigger	User clicks on 'Exit button in the main menu
Main Flow	1. User clicks on 'Exit button 2. The user waits as the system ends processes gracefully 3. The system closes
Postcondition	After a few seconds, the user has now left the application

Chapter 4: Design

This chapter outlines the main idea for the technical demo as well as covering a high-level overview of how it was achieved.

4.1 Unreal Engine 4

Following from section 2.7, I used Unreal Engine 4 to build the technical demonstration. This helped to handle components such as rendering and physics so that I could dedicate more development time to the implementation of procedural content generation.

Another benefit of using Unreal Engine 4 was that it is industry standard. By using it to build the technical demonstration, I gained valuable experience with tools and features commonly used in the professional development of games.

4.2 Game Demo

To showcase real-time procedural generation in games, I designed an exploration-based game demo. The demo was set on an infinite 3D landscape where the player can explore multiple different biomes.

The landscape is generated as a 2D heightmap using Perlin Noise, Fractal Brownian Motion, and Domain Warping. Like Dwarf Fortress (Section 2.5), fractal layers were generated representing attributes such as moisture, temperature and salinity to create a diverse set of biomes. These biomes each have distinctive colour palettes and physical features such as canyons or sand dunes.

4.3 Infinite World

In order to generate an infinite world for the player to explore, the system only has a finite number of terrain 'chunks' loaded at a time. By loading only the chunks close to the player, the illusion of an infinite world is created while remaining computationally possible.

These world chunks act like a treadmill: appearing in front of the player and disappearing behind them. As long as the chunks load/unload at a sufficiently far distance from the player, they are not aware of the process.

The size of the chunks and the distance at which they load/unload was tested and tweaked to provide the best experience when moving through the world. Chunk loading distance can also be used on a sliding scale to balance performance and visual quality.

4.4 Teleological Enhancements

As detailed in requirement **NFR-4** (Section 3.2), Teleological enhancements could have been used to enhance the realism of the generated landscape. The main issue, however, is that Teleological algorithms classically have to generate all of the content at once – which can't be done on an infinite world.

The reason that a Teleological algorithm such as hydraulic erosion (rainfall simulation) can't be used on one chunk at a time is that the water would erode terrain, move downhill, and press against the boundary of the chunk. This would leave a large amount of sediment being deposited against this border and leaving very obvious visual artefacts between chunks.

My proposed solution is to involve a finite number of each chunk's neighbours when simulating the effects of hydraulic erosion. This would help to mitigate the issue of water pressing against the chunk boundaries.

After loading the 3x3 grid of neighbouring chunks, the system would simulate rain falling on only the centre chunk (blue square in Fig. 14. By allowing the centre chunk's rain to flow and evaporate across the neighbouring chunks (green), much less water should reach the outer boundary (red). If it's observed that too much water is reaching this boundary, the evaporation rate can be increased, or the green zone extended.

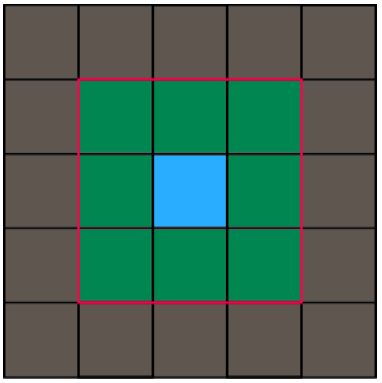


Figure 14: Chunk neighbour involvement

Chapter 5: Evaluation Strategy

5.1 Content quality

There are multiple objective measures we can use to evaluate the content quality:

1. Terrain Resolution (5.1.1) – measured in mesh vertices per metre.
2. Number of attributes used (5.1.2) – attributes such as precipitation, heat etc.
3. Number of simulation iterations (5.1.3) – how long terrain simulations should last
4. Number of distinct biomes (5.1.4) – hills, forest, mountains etc.
5. How long the world takes to load (5.2.1) – measured in seconds
6. Frame rate (5.2.2) – measured in frames per second

5.1.1 Terrain resolution

To render the terrain for the demo, a mesh of triangles is created from the generated heightmap. Each vertex of this mesh corresponds to the height at a given point, and changing how far apart these vertices are provides a control over the 'resolution' of the terrain.

When using image-imported heightmaps, Unreal Engine 4 uses the standard of *1 pixel = 1 metre* (World of Level Design, 2020). This means that resultant meshes have a resolution of one vertex per metre. The system should aim to match this resolution to ensure that landscapes look sufficiently sharp and well defined.

5.1.2 Number of attributes used

The study in section 2.5 highlights the importance of using multiple attributes to generate terrain. Dwarf Fortress uses height, temperature, annual rainfall and soil deposition as the primary components for making realistic-looking terrain.

Like Dwarf Fortress, the system should use a similar number of attributes to improve the realism of the generated terrain.

5.1.3 Number of simulation iterations

A demonstration of hydraulic erosion on terrain (Lague, 2019) showed a heightmap with resolution 255x255 converging after 70,000 rain droplets. The demonstration ran on a six-core CPU in around 750 milliseconds.

Using Lague's simulation as a benchmark for hydraulic erosion, the system should aim to achieve at least 1 ($70{,}000/255^2=1.08$) raindrop simulation per heightmap vertex.

5.1.4 Number of distinct biomes

A large part of what makes real-world landscapes interesting are their diversity. Biomes such as forests, deserts and plains make exploration of these landscapes more enjoyable.

Like Dwarf Fortress (Section 2.5), the system should aim to have multiple different biomes consisting of unique features and colour schemes. The functional requirements FR-6.1 and FR-6.2 outline the requirement for at least 3 – and preferably 5 – unique biomes.

5.2 Performance

To keep the player engaged, the technical demonstration should generate worlds quickly and should maintain good rendering quality while running. There are multiple objective metrics that capture these requirements:

1. Loading time when entering a new world (5.2.1)
2. Loading time when entering a saved world (can benefit from world storage) (5.2.1)
3. Frame rate when standing still (5.2.2)
4. Frame rate when moving (during real-time generation of new chunks) (5.2.2)

5.2.1 Loading time

To keep players engaged, the system should aim to minimise loading times. The system must not have load times exceeding 1 minute (NFR-1.1), and should preferably load in under 30 seconds (NFR-1.2).

5.2.2 Frame rate

The number of frames rendered per second (fps) by the system is important to ensure a comfortable gameplay experience. As outlined by Non-functional requirements NFR-2.1 and NFR-2.2, the system must have a minimum of 30fps and should have 60fps.

This metric is dependant upon the specific task the system is executing, so I will design tools to monitor the framerate over time in the context of measurements such as minimum, maximum, mean and standard deviation.

Chapter 6: Project Management

6.1 Timeline

The Gantt chart found in the appendix (Appendix 1) shows the plan for this project's implementation in semester 2. The plan is broken into development of the application, writeup of the dissertation, and preparation for the final presentation.

The development of the application started in early January, beginning with musts, then shoulds, then coulds. One week was allocated at the end for the writing of documentation and packaging of the distributable for sharing with others.

Throughout the application's development, sections on the issues faced, solutions found, and other useful information were written. One week was allocated once development was done to test and evaluate the system. The remaining time (roughly half the project's duration) was used to write the rest of the dissertation.

6.2 Risk Identification

To identify project risks, the following MoSCoW analysis matrix was used:

	Low impact	Medium impact	High impact
Low likelihood	Low risk	Low risk	Medium risk
Medium likelihood	Low risk	Medium risk	High risk
High likelihood	Medium risk	High risk	High risk

Using the above risk matrix, the following risks were identified and assessed:

Description	Likelihood	Impact	Risk
R1: Loss or corruption of work	Medium	High	High
R2: Illness or temporary inability to work	Medium	Medium	Medium
R3: Requirements turn out to be too challenging	Low	High	Medium
R4: Final demo has performance issues	Low	Medium	Low
R5: Final demo has bugs	Medium	Low	Low
R6: Demo incompatible with standard hardware	Low	Low	Low
R7: Cannot complete work on time	Low	High	Medium

6.3 Risk Mitigation

R1: Loss or corruption of work

Through hardware failure, loss or human error – project data may be lost. Without mitigation, the likelihood of applications crashing or the accidental deletion of files was quite high and could have resulted in setbacks. To mitigate the impact of data loss, all project data was source controlled.

Dissertation documents were synced with one drive every day, and the application code was pushed to a git repository every significant change.

R2: Illness or temporary inability to work

Illness or general inability work is always possible, and especially likely during Covid. To mitigate the impact of this issue, work was scheduled with adequate time before deadlines to cover time lost. Source code and documentation were stored online to allow for working from home.

R3: Requirements turn out to be too challenging

It was possible that the requirements set out initially were too challenging to complete. For this reason, must-have requirements were scheduled to be completed before working on should-haves and finally could-haves. The implementation of the must-haves that form the minimal viable product were scheduled to take only a third of the total development time so there was adequate time to work on backlog if required.

R4: Final demo has performance issues

As the technical demo was likely to be quite computationally tasking, the final product may have had low or stuttering framerate when running. The likelihood of this becoming an issue was reduced by including a variety of controls such as render distance, memory allocation, and hardware acceleration.

R5: Final demo has bugs

Two things were done to reduce the likelihood of bugs in the final demo.

The *KISS principle (Keep it Simple, Stupid)* (principles-wiki, 2021) was used when designing the system. The technical demonstration did not need to be particularly complex to achieve its goals and keeping it simple helped to avoid issues later down the line. The C++ component of the system also uses modern design principles such as *RAII* (cppreference, 2021) and *the rule of 3/5/0* (cppreference, 2021).

Secondly, testing was used to ensure that the individual components of the system behave as intended. The demo was also regularly play-tested to make sure that it played smoothly.

R6: Demo incompatible with standard hardware

With the potential use of low-level technologies such as Cuda and SIMD, the demo ran a risk of incompatibility with certain systems.

To reduce the likelihood of platform incompatibility, platform-specific technologies were designed as an optional extra for the system. Users without NVIDIA cards should be able to disable GPU use and platform-specific SIMD operations should have cross-platform alternatives.

R7: Cannot complete work on time

To mitigate the impact of incomplete work on the project, the timeline was designed to firstly produce a minimal viable product from the must-have requirements. This section of the timeline takes up only a third of the total development time so there was adequate time to implement a basic product that satisfies the must-have requirements, FR-1, FR-2, and NFR-1.1.

6.4 Legal and Ethical Considerations

One ethical consideration of procedural generation in the long term is its ability to replace human designers. Moving forward, game developers should be conscious of this issue and *make sure that their use of automation is ethical* (ThinkAutomation, 2021). Procedural generation can enable designers to build larger worlds faster, but it should not – and currently, *cannot* – be used to replace artists and designers entirely.

Simplex Noise is currently patented by Ken Perlin, expiring in 2022. However, as *fair use* (Wikipedia, 2021) allows for the use of patented work in non-profit research or study, this project is exempt. If I am to continue development and commercialise the application in the future, I will need to instead use a similar noise algorithm such as Perlin Noise or the patent-free, slightly slower implementation of Simplex Noise: Open Simplex Noise (Gustavson, 2005).

All assets and textures used in the technical demonstration are free to use in Unreal Engine, for personal or commercial purposes. The textures used in the project are from Quixel, an Unreal-owned asset scans library (more in 7.3.2 Sourcing Textures).

Chapter 7: Implementation

7.1 Implementation Overview

Over the three months allocated for project implementation, I completed four sprints:

1. Sprint 1: Project infrastructure and basic implementation
2. Sprint 2: GUI, graphics and gameplay
3. Sprint 3: Performance optimisations
4. Sprint 4: High-level interfaces and polishing

During the implementation phase, I also solved several key issues that arose:

1. Mitigated an issue with Unreal's limited standard library compatibility (7.2.3)
2. Identified a mesh generation bottleneck (7.2.8)
3. Implemented a solution for texturing procedural landscapes (7.3.4)

7.2 Sprint 1: Infrastructure

1st January – 15th January 2022

7.2.1 Sprint 1 Overview

The first sprint I completed lasted 15 days and covered the core infrastructure of the project, as well as implementing some of the math and procedural generation functions needed for terrain generation. During this sprint, I completed the following objectives:

1. Set up the project (7.2.2)
2. Mitigated an issue with limited standard library compatibility (7.2.3)
3. Implemented procedurally generated chunks of terrain (7.2.4)
4. Implemented procedural meshing (7.2.5)
5. Implemented Perlin Noise function (7.2.6)
6. Implemented infinite world functionality (7.2.7)
7. Identified a bottleneck in Unreal's procedural mesh generation (7.2.8)

7.2.2 Unreal Project Setup

The first step in setting up the project was to ensure I had the relevant tools:

1. Git – for version-controlling the project and mitigating data loss
2. Unreal Engine Editor – the graphical application required for seeing, editing, and running Unreal projects in development mode (with debugging).
3. Visual Studio – for editing and building C++ source code used in the project.

Once these were installed/updated, I created a new blank C++ game project. This template comes with no starter content such as characters or objects, but specifies that C++-based classes can be added to the project.

One benefit of using Unreal right away, was that frame rate metrics are already available in-editor and in-game. This satisfied requirement FR-4 without having to program additional functionality.

7.2.3 Limited Standard Library Compatibility

The C++ standard specifies what's called the 'standard library', defining core functions and classes such as arrays, maps and strings. Although each C++ compiler is free to implement these functionalities in whichever they like, they must implement these if they want to adhere to the standard. This means that, no matter what compiler they're using, developers can rely on the functionality of the standard library when programming with C++.

Unreal Engine, unfortunately, has limited compatibility with the standard library. Specifically, no functions or classes defined by Unreal (such as Actor) use standard library components; they use their own custom library. This is primarily because Unreal Engine implements automatic memory management in all of their classes – making it similar to the system in C# or Java.

This causes an issue for multiple reasons:

1. The runtime may be slower. If the memory of objects are being automatically managed, C++ written using Unreal's library may have a similar computational overhead to C# or Java.
2. Developers using Unreal have to learn how to use Unreal's standard library which, in places, can differ quite significantly to C++'s standard library.
3. Code written for purposes outside of Unreal can't be reused without significant refactoring to Unreal's custom library.

Slower runtime is an issue that can be partially mitigated. Although Unreal's classes don't use standard library components (and therefore don't accept them as parameters or return them), developers can still use standard library components; they just have to convert them to and from Unreal's custom components before interfacing with Unreal-implemented classes.

As a compromise between the performance of standard library and the compatibility of Unreal's library, I predominantly used Unreal's components, only using standard library components in performance hotspots such as mesh generation.

Having to re-learn the standard library in Unreal Engine's variation was a minor but frequently occurring issue. The library is very comprehensive and, although a few useful functions are missing, all of the necessary classes from C++ are implemented by Unreal Engine. However, having to regularly search for documentation on arrays, maps, sets, tuples, and everything else I've learned over the years, proved to be quite time consuming.

A common design principle in interface design is to leverage the user's existing knowledge. A person going to a website for the first time should not have to read a manual for that specific website: instead, the website should use buttons and icons the user already knows the purpose of. Similarly, I believe that Unreal Engine could leverage developer's pre-existing knowledge of C++'s standard library, and shape their custom library to be more intuitive to C++ developers.

To illustrate this point, I've demonstrated how a developer writes a simple message to the console in standard C++, and in Unreal's implementation:

```
char const *name = "Chris";

// Printing a name in standard C++
std::cout << "Hello, " << name << std::endl;

// Printing a name in Unreal Engine
FString unrealString{name};
UE_LOG(LogTemp, Log, TEXT("Hello, %s"), *unrealString);
```

This pattern of significantly different implementations for the same functionality continues in most of Unreal's classes and functions. Using this library is easy once learned, but learning it took an unexpectedly large portion of the limited development time set aside.

7.2.4 Procedural Terrain Chunk

Anything in Unreal Engine that can be placed in the level (the virtual world) is an *Actor*. Actors are defined by a geographical position, orientation, and scale. As Unreal uses an inheritance-based class system, classes like meshes, particle emitters, and even sounds all extend from the Actor base class.

One specific class that inherits from Actor is ProceduralMeshComponent. Meshes are essentially just collections of coordinates that the engine uses to build up a graphical representation of an object, and they're usually statically loaded from a file. What the ProceduralMeshComponent class implements, however, is the ability to dynamically generate each coordinate of the mesh using C++. This is very important for procedurally generating meshes during runtime.

Extending from Unreal's ProceduralMeshComponent class to allow for dynamic mesh generation, I created a C++ class named ProceduralTerrainChunk. This class encapsulates the visible mesh for a single chunk of terrain (4.3 Infinite World), and allows it to be generated and regenerated in real time.

An important part of the design of this class is that its construction is separate to the point at which the mesh is procedurally generated. This means that generation of the chunk's mesh – a computationally tasking operation – can be deferred if the system is currently under too much pressure. Using this method, chunks will be invisible until the system has time to load them, and the framerate stays stable.

7.2.5 Procedural Mesh Generation

To generate a mesh in Unreal, two primary data arrays are required: **vertices** and **indices**.

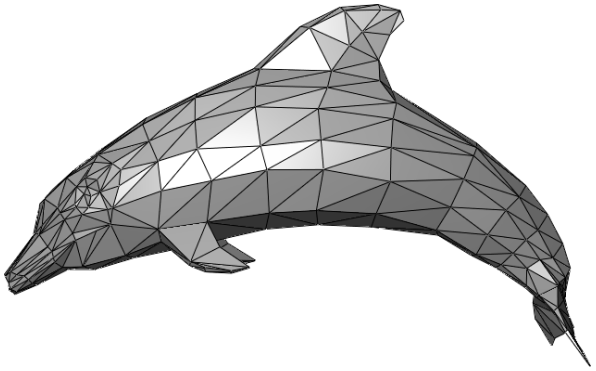


Figure 15: A dolphin mesh made from many vertices (Wikipedia, 2022)

The **vertices** array is an array of 3D coordinates. These coordinates represent every corner of a graphical mesh. Additional to physical coordinates, each vertex can specify extra data specifying such as how they should link to textures, what direction they're facing, or what colour the vertex is.

Almost all modern computers render graphical models using triangles. What the **indices** array defines is what vertices should be grouped together into triangles. The indices array contains

unsigned integers that, when grouped in threes, define the index within the vertex array for the corners of each triangle to be rendered.

The vertex array for 3D, heightmap-based landscapes is quite simple to code: arrange the vertices in a square grid on the X and Y directions, and calculate the Z (altitude) coordinate based on:

Z = perlinNoise(X, Y)

The index array for the terrain can be generated in multiple ways. By splitting the mesh up into quads, and each of those quads into two triangles, the effect seen in fig. 17 is achieved. This mesh works fine, but the lack of diversity in diagonal orientations leaves it looking somewhat stretched.

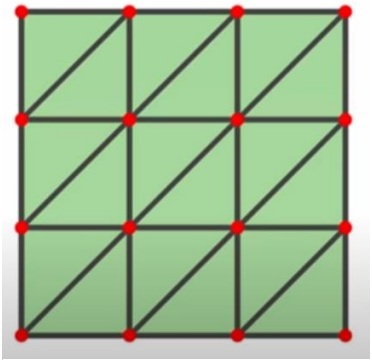


Figure 16: Mesh without quad alternating (ThinMatrix, 2017)

To mitigate the issue of the mesh looking stretched in one direction, I alternated the way in which quads are subdivided into triangles. As seen in fig. 18, alternating the quad division orientation looks much more evenly distributed.

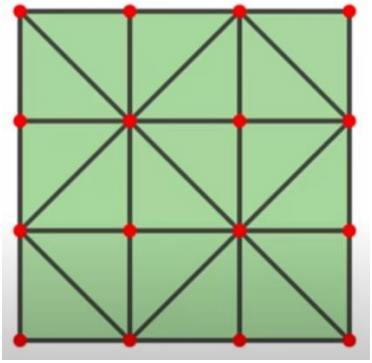


Figure 17: Mesh with quad alternating (ThinMatrix, 2017)

7.2.6 Perlin Noise

Once the code required for generating the graphical mesh was completed, I started on the implementation of Perlin Noise. This went through many iterations over the course of the project, but the final implementation is this (Also hosted on GitHub):

```
float PerlinNoise::standard(float x, float y, int seed)
{
    // Grid cell coordinates
    int16_t X=std::floor(x), Y=std::floor(y);

    // Sample coordinates within cell
    FVector2D sample(x-X, y-Y);

    // Get sample vectors relative to each corner of the grid cell
    FVector2D rBL = sample - FVector2D(0, 0);
    FVector2D rBR = sample - FVector2D(1, 0);
    FVector2D rTL = sample - FVector2D(0, 1);
    FVector2D rTR = sample - FVector2D(1, 1);

    // Generate pseudorandom vectors for each corner of the cell
    FVector2D pBL = randomVector(X,   Y,   seed);
    FVector2D pBR = randomVector(X+1, Y,   seed);
    FVector2D pTL = randomVector(X,   Y+1, seed);
    FVector2D pTR = randomVector(X+1, Y+1, seed);

    // Calculate dot products for each relative and pseudorandom vector
    float dBL = FVector2D::DotProduct(pBL, rBL);
    float dBR = FVector2D::DotProduct(pBR, rBR);
    float dTL = FVector2D::DotProduct(pTL, rTL);
    float dTR = FVector2D::DotProduct(pTR, rTR);

    // Interpolate between corner dot products using smootherstep
    float xLerp=Math::smootherstep(sample.X), yLerp=Math::smootherstep(sample.Y);
    float dB = Math::lerp(dBL, dBR, xLerp);
    float dT = Math::lerp(dTL, dTR, xLerp);
    float d  = Math::lerp(dB,  dT,  yLerp);

    return d;
}
```

Within the namespace *PerlinNoise*, the function *standard* implements one octave of regular Perlin Noise. Using the method explained in 2.1.1: Perlin Noise Implementation, the algorithm leverages Unreal's built-in vector library to efficiently calculate and interpolate the dot products of each pseudorandom cell corner into one, smoothed (2.1.2 Smoothing) output.

The function *randomVector* uses a hashing function (later covered in section 7.4.5) to pseudo randomly generate vectors consistently and efficiently.

The function *lerp* is a basic function that interpolates between the first and second parameter using the third as the interpolant. Provided with the smoothed interpolant, *lerp* returns the smoothed interpolation between two points.

7.2.7 Treadmill

Plugging the Perlin Noise function into the mesh generation code, I was able to create and see a chunk of the terrain in the editor.

The next step was to implement the functionality for creating and destroying chunks based on player position (4.3 Infinite World).

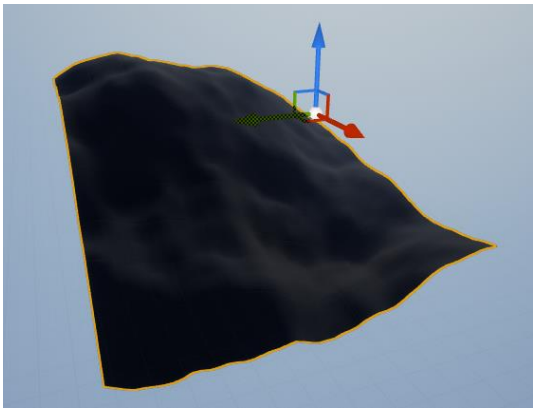


Figure 18: Procedurally generated chunk of terrain. (Screenshot from the working demo)

To encapsulate the collection of ProceduralTerrainChunks, I created a class named ProceduralTerrain. This primarily consists of a HashMap containing all of the currently active chunks, indexed by their coordinates. Every frame, the class checks to see if:

1. Chunks are spawned when they shouldn't be (need to despawn)
2. Chunks aren't spawned when they should be (need to spawn)

To see if chunks should be spawned, I originally just checked if the chunk's centre was within a certain distance to the player. This worked well and meant that only the necessary chunks were spawned at once.

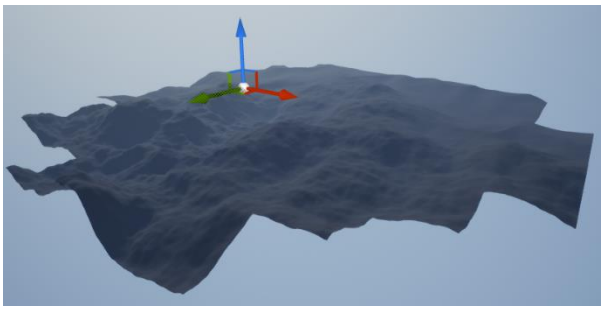


Figure 19: Many chunks spawned around a central point. (Screenshot from the working demo)

This method of spawning and despawning chunks based on one distance check had an issue, however. When moving back and forth quickly over the boundary of a chunk, distant chunks would be rapidly spawning and despawning, using up unnecessary amounts of computing power to keep generating and deleting, multiple times a second.

The solution for this was to have separate spawn and despawn radii. Chunks would spawn in at a closer distance and despawn at a farther distance. This meant that if the player steps over a boundary to spawn in a new chunk, taking one step back no longer immediately despawns the chunk.

Fig. 21 shows how certain chunks (dark blue) are outside the spawn radius, but have not yet reached the despawn radius.

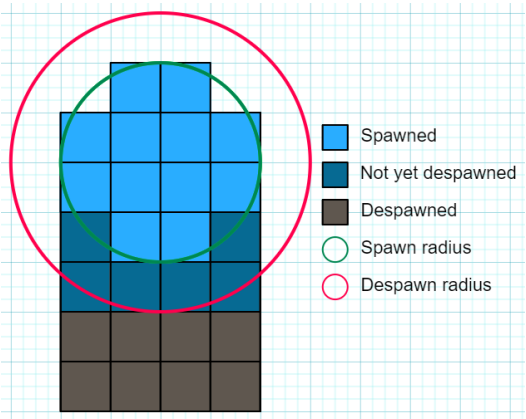


Figure 20: Illustration of separate spawn and despawn radii as the player moves north

One further optimisation made at this stage was selecting the optimal chunk size for a fixed view distance. Iterating multiple chunk sizes, I found that the optimal size for speed is around 50x50 vertices, each 0.4 metres apart (7.4.4). Loading these 400m^2 chunks initially each took around 14ms, with later improvements cutting the time down to under 7ms (7.4.2).

7.2.8 Mesh Creation Limitation

After implementing treadmill functionality for spawning lots of chunks at once, I noticed that the generation was taking longer than expected – about 14ms per chunk. Profiling the code, I found that around 7ms was being used to procedurally generate the data, and around 7ms was being used on Unreal's *CreateMeshSection_LinearColor* function.

The 7ms being used to procedurally generate the data for each chunk is fine. It's expected as it's a computationally tasking operation, and it can also be parallelised or run on a worker thread quite easily (7.4.2).

The 7ms being taken on the *CreateMeshSection_LinearColor* function, however, is much more concerning. The function's main purpose is to take the vertex and index arrays and turn them into a visible mesh. In a graphics API like OpenGL, this would only consist of sending the data to the GPU but, as Unreal Engine is not open source, I was unable to find exactly what Unreal does within this function.

The reason that Unreal's function taking 7ms to run is concerning, is because all Unreal function calls must be made from the main thread. Almost nothing in Unreal Engine is thread safe, and functions will throw runtime exceptions if not called from the main thread. Because of this, there is an unavoidable hard limit on how many meshes can be created per tick.

If the game is to run at 60 frames per second, each frame gets a maximum of 16.67ms to complete all operations before moving onto the next tick. These operations include:

1. Unreal's internal operations. Memory management, hardware polling etc.
2. The developer's custom operations. Primarily generating and displaying chunks.
3. Unreal's rendering operations. Sending data to the GPU, updating uniform variables etc.

With operations like these all having to run on the same thread, I later had to make compromises on graphics to ensure the terrain can load fast enough (7.4.4).

7.3 Sprint 2: Graphics

16th January – 31st January 2022

7.3.1 Sprint 2 Overview

The second sprint I completed lasted 16 days and covered the graphical aspects of the project; primarily using Unreal's material interface to design the landscape's texturing. During this sprint, I completed the following objectives:

1. Sourced textures for PBR materials (7.3.2)
2. Created materials using Unreal's node editor (7.3.3)
3. Implemented gradient-based material blending (7.3.4)
4. Added an animated, third-person player character (7.3.5)

7.3.2 Sourcing Textures

Textures are images that wrap around a graphical mesh. Their main purpose is to define the colour of the object, but modern graphics can use many other texture types to define characteristics such as the roughness, bumpiness and metallicness. The use of these additional textures is called PBR (Physically Based Rendering).

To add fine-grained detail to the generated landscape, I decided to use multiple PBR textures:

1. Albedo: Defines the colour of the object when evenly lit.
2. Normal: Defines the direction the material is facing. Used for fine-grained lighting.
3. Roughness: Defines how reflective the material is.
4. Ambient Occlusion: Pre-calculated shadow detail for intricate lighting.

Using these four textures, we can give the rendering process extra information that helps it to more accurately apply light to the material:

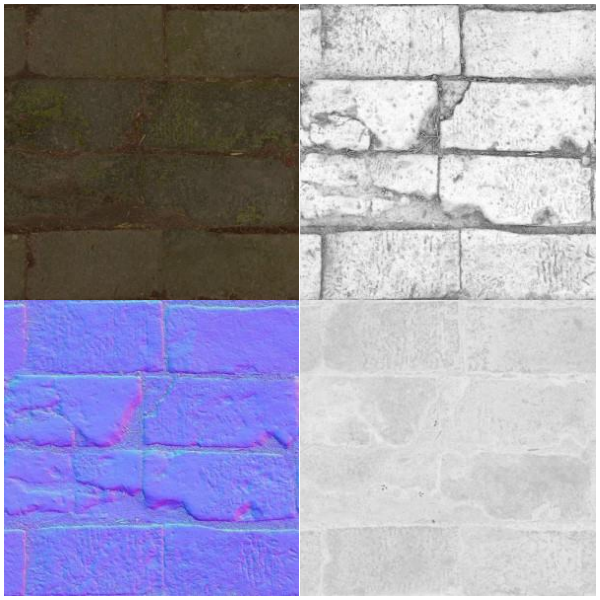


Figure 22: Albedo, Ambient Occlusion, Normal, and Roughness textures (Quixel, 2022)

Figure 21: Final rendered image (Quixel, 2022)

To source high-quality, free-to-use PBR textures, I used a website called Quixel Megascans. Quixel is an asset library that was recently acquired by Epic Games – the developers of Unreal Engine. The assets are completely free for use within the Unreal Engine, for both personal and professional use.

Quixel materials also feature a displacement texture. These textures define how much the material 'pops out' at any given point. Although vertex meshes usually define geometric data such as this, using a low-vertex-count mesh with a displacement texture allows the system to optionally subdivide the mesh for higher geometric detail. This process is called Tessellation and, although I tried it in the application for a while, the benefits did not justify the heavy performance cost.

7.3.3 Creating Materials

Materials in Unreal Engine define how a mesh is rendered. The primary components of materials are the different texture sources it uses, but it can also define complex rules to achieve effects such as gradient-based rendering.

Creating a basic material in Unreal is very simple. After importing the textures to the project, they can be dragged onto a newly created material, and linked to the correct output node (Right box in Fig. 24).

Unreal Engine automatically converts a wide range of different image files into compatible formats for textures, so I had very little issue importing different textures to build up the materials.

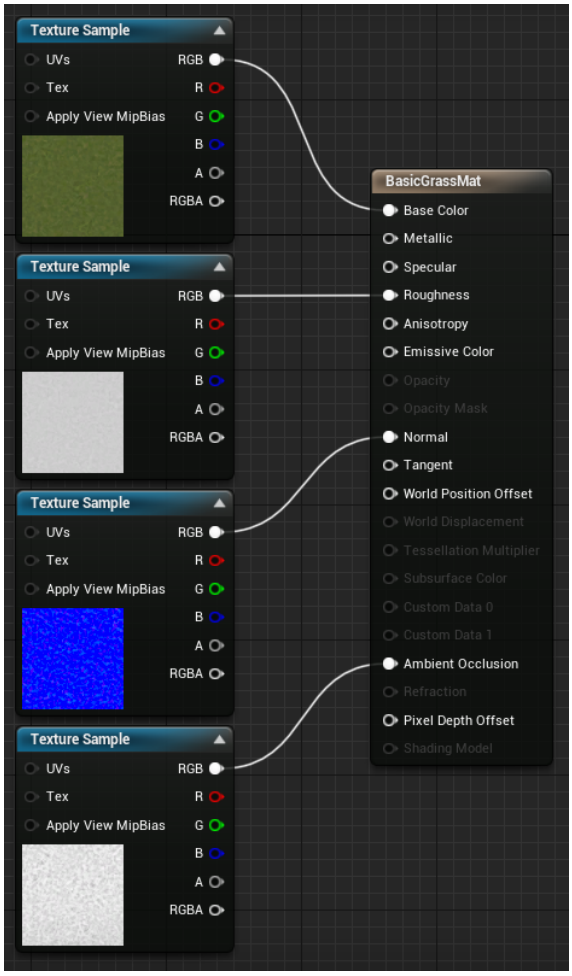


Figure 23: A basic UE4 material, utilising PBR textures. (Screenshot from the working demo)

Materials in Unreal can also use parameterised variables. Parameterising certain variables, like the roughness multiplier seen in fig. 25, allows these variables to be quickly changed from the engine's GUI, and avoids having to recompile the material every time the variable is changed.

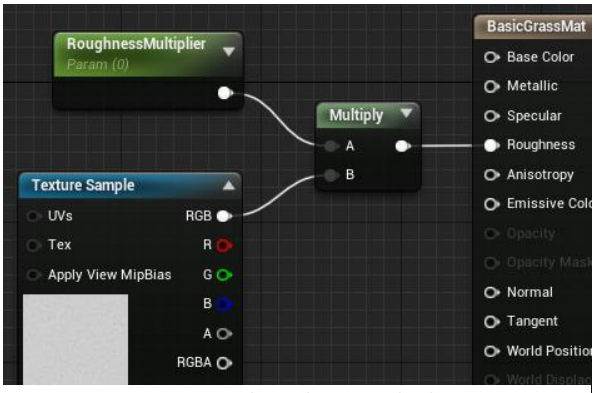


Figure 24: Parameterised roughness multiplier. (Screenshot from the working demo)

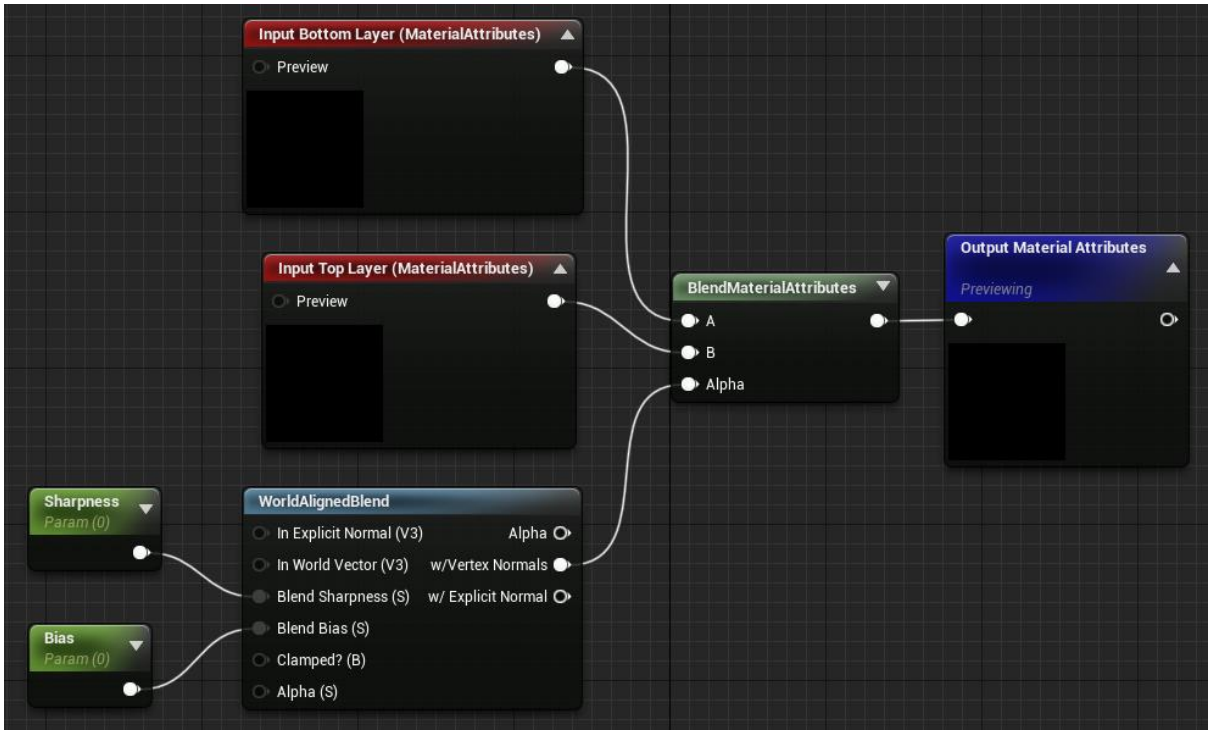
The example shown with the roughness multiplier allows the developer to quickly change how much the roughness texture affects the final roughness of the material. A value of zero makes the material completely smooth, and a value of two makes the material twice as rough as the texture defines. Adding parameterised controls like this helped me quickly make small adjustments to the materials in-engine.

7.3.4 Gradient-Based Blending

In the real world, flat terrain often looks very different to sloped terrain. This is because vegetation such as grass can easily grow on flat surfaces, but sloped surfaces tend to be less hospitable for growth and end up looking muddy or rocky.

Gradient-based blending draws from this phenomenon and defines a method to automatically interpolate between a steep texture and a shallow texture, based on the gradient of the terrain.

Using Unreal's *WorldAlignedBlend* to blend between two textures based on the world-space alignment (terrain slope), a blend function can be defined:



The parameter, sharpness, defines how sharply the material switches from one texture to another, and bias allowed me to adjust the ratio of shallow to steep texturing for certain gradients.

Plugging basic snow and rock materials into the gradient-based blend function, I was able to achieve this result:

Figure 25: Gradient-based blending of textures. (Screenshot from the working demo)

7.3.5 Third-Person Player Character

Up until this point, the terrain had only been viewable from Unreal's editor using the editor camera. To allow players to walk on the terrain and view it up close, I added a third-person player character.

Unreal has a large library of assets that are free for use within the engine, including a selection of rigged character models, animations, and player controllers. I selected one of the base characters used in a lot of starter content, and added a jump animation so that players can scale steep areas of the terrain.

Although the use of a high-level game engine had been limiting in some aspects of development, adding and animating a character into the project was very simple. Where similar tasks could have taken days or weeks using a graphics API, this took no longer than 20 minutes to set up in Unreal.

Figure 26: Jump animation for Unreal's player character model. (Screenshot from the working demo)

7.4 Sprint 3: Performance

1st February – 15th February 2022

7.4.1 Sprint 3 Overview

The third sprint I completed lasted 15 days and was spent optimising the application for better performance in world generation and graphics. During this sprint, I completed the following objectives:

1. Implemented multithreaded world generation (7.4.2)
2. Identified performance bottleneck in the application (7.4.3)
3. Implemented measures to balance quality and performance (7.4.4)
4. Made low-level performance improvements to my Perlin Noise implementation (7.4.5)

7.4.2 Multithreaded World Generation

Creating and meshing terrain takes 14ms when both on the main thread – precious time that can be used for other processes like rendering. While 7ms of that time is unavoidable (7.2.8: Mesh Creation Limitation), the 7ms taken to procedurally generate the vertex array for each chunk can be moved onto other threads.

Generating a visible mesh with Perlin Noise has two main steps:

1. Build up a vertex array (coordinate list), using Perlin Noise
2. Provide that vertex array to Unreal's mesh function, *CreateMeshSection_LinearColor*

Although *CreateMeshSection_LinearColor* must be called on the main thread, generating the vertex array does not have the same restriction. Using a thread-safe priority queue to queue up generation tasks, we can allocate the first step to one or more worker threads using the following steps:

1. Chunk is constructed on the **main thread**, but the deferred generation (7.2.4) means that it remains invisible until the system is ready to generate the mesh.
2. As soon as the chunk is constructed, the **main thread** adds a task to a thread-safe queue, signalling that the chunk is waiting for its vertex array.

3. One or more **worker threads** constantly dequeue these tasks and generate the vertex arrays, off of the main thread. Once finished, they toggle a boolean to let the chunk know it's ready to create the mesh.
4. Each tick, the **main thread** polls each chunk waiting for mesh generation and, if the vertex array is ready, it will call *CreateMeshSection_LinearColor* on the min thread.

7.4.3 World Generation Bottleneck

After moving vertex array generation off of the main thread, there were visible improvements in the speed of world generation. Profiling the code, I found that 7ms was still being used to create the meshes, but queuing and utilising the vertex arrays generated on worker threads took just nanoseconds. This decrease in chunk generation time from 14ms to 7ms doubled the speed of world generation but, as the remaining 7ms is unavoidable, this was the absolute limit of how fast the chunks can be generated.

Profiling the worker threads, I found that even a single worker thread dedicated to vertex array generation was idle around 40% of the time. Because the main thread is bottlenecked and can't consume vertex arrays any faster, one worker thread was more than enough to generate vertex arrays fast enough for maximum speed up.

Because of my previous experience using OpenGL, I had assumed that generating the vertex array would be the bottleneck, not 'registering' it with Unreal. In lower-level graphics APIs, the vertex array is immediately ready to use, only having to be sent to GPU memory. Unfortunately, Unreal seems to have much more overhead when registering the vertex array for use on the GPU.

My previous could-have requirements, outlining hardware configurations (FR-5) and GPU usage (NFR-6) depended upon the assumption that vertex generation would be the primary bottleneck. However, because this isn't the case, and because world generation is bottlenecked by an unavoidable part of Unreal, these requirements were rendered redundant.

I spent a lot of time looking for a mitigation, but I was able to find very little on the subject. The use of ProceduralMeshComponent is quite rare in the community and there's no documentation for the functions other than basic parameter descriptions (Epic Games, 2022).

7.4.4 Performance-Quality Balance

Modern games recognise that players value different aspects of a game, and provide controls so that users can choose whether they prioritise aspects such as 4k textures and 100-mile render distances, or whether they don't mind sacrificing on visual quality for fast and responsive frame rates.

So that players can customise their experience, and so that evaluating performance is quick and easy to perform, I built in multiple parameters for world generation and graphics quality. There are two main controls for this:

1. Terrain resolution: How close together vertices of the terrain are
2. View distance: how far away terrain chunks are spawned in

Terrain resolution defines controls how intricate the terrain detail is. High resolution terrain picks up all the tiny bumps and imperfections of the landscape, but comes at the cost of high mesh generation times. Low resolution terrain loads fast, but it doesn't show the same intricacy as high-resolution terrain. The comparisons can be seen below:

Figure 27: Terrain with varying levels of resolution. ***Top left****: 1.25 vertices per metre.* ***Top right****: 2.5 vertices per metre.* ***Bottom Left****: 5 vertices per metre.* ***Bottom right****: 10 vertices per metre. (Screenshot from the working demo)*

Each step up in resolution seen in fig. 28 comes at the cost of quadrupled generation time per chunk. Although vertex array generation can be done asynchronously and on multiple threads (7.4.2), mesh generation is locked to the main thread (7.2.8). Because of this, 2.5 vertices per metre is the highest resolution that can be achieved while maintaining a stable frame rate above 60Hz.

View distance has two limiting factors:

1. Higher view distance means more chunks per second must be generated on the CPU
2. More chunks loaded in at once means more polygons that the GPU has to render

When moving, the CPU is having to generate new meshes for the chunks, using up CPU time and potentially limiting the frame rate. Using a terrain resolution of 2.5 vertices per metre, a view distance of **200m** can be achieved while maintaining a stable frame rate of 60Hz.

When standing still, the CPU is not having to generate new meshes and the only limiting factor is how many polygons the GPU can render in time. Using the same 2.5 vertices per metre resolution, an extended view distance of **250m** can be achieved while maintaining a stable frame rate of 60Hz.

Figure 28: Top: 200m view distance. Bottom: 250m view distance. (Screenshot from the working demo)

For rough reference, Minecraft uses a default view distance of 192m (Mojang, 2022) – just under what my technical demo achieves while the player is moving.

Finally, as the player should not be able to see distant chunks popping in and out, fog can be added to the background so that terrain gradually transitions in and out. Using Unreal's exponential height fog scene component, we can easily add fog that covers up chunks spawning and despawning far away:

Figure 29: Fog hiding distant chunks loading in and out. (Screenshot from the working demo)

7.4.5 Performance Improvements

As discussed in Section 7.2.8, the generation of vertex arrays using Perlin Noise is bottlenecked by Unreal's efficiency in consuming the data to generate meshes. Because of this, improving the performance of my initially written code does nothing to improve the application's performance and I have therefore focussed on other features of the project.

However, as my initial reason for exploring this area was largely influenced by my interest in high-performance code, I made some optimisations to particularly slow parts of the Perlin Noise generation functionality.

The primary component of Perlin noise left up to the implementer is the pseudorandom vector generation. Many implementations use a hash table to map integer coordinates to pre-calculated vectors, stored in a large array. This is fast, but it depends on the assumption of shared memory. All CPU/GPU processes must either have shared access to the same hash table, or keep copies of their own. As GPU acceleration was initially considered, I decided that shared memory was not an assumption the system could rely on.

As an alternative, my implementation directly hashes the X and Y coordinates – along with an optional seed – into the outputted vector. This requires no persistent memory allocations and, if the hashing function is efficient, can be just as fast.

To develop the application quickly, I initially just used C++'s built-in hashing function. While it worked well, there were two main concerns:

1. The hash is general purpose, which means that purpose-oriented optimisations can't have been made and the algorithm is not particularly fast.
2. The hash implementation varies between compilers. This means that speed and output can vary. Some integer hashing implementations just use a single modulo operation.

I tried to build a custom hashing function of my own, but most were slow or left visual artefacts in the resultant noise. Researching online, I found a resource exploring many different performance-oriented integer hashing functions. A blog (Jenkins, 2022) by Bob Jenkins, building on previous work by Thomas Wang, described a half-avalanche, integer-hashing function (Hosted on GitHub):

```
uint32_t Math::intHash(uint32_t a)
{
    a = (a+0x479ab41d) + (a<<8);
    a = (a^0xe4aa10ce) ^ (a>>5);
    a = (a+0x9942f0a6) - (a<<14);
    a = (a^0x5aedd67d) ^ (a>>3);
    a = (a+0x17bea992) + (a<<7);
    return a;
}
```

The function uses minimal and computationally lightweight operations to achieve a high amount of entropy for hashing integer keys. Specifically, this algorithm is half avalanche.

Avalanche is the measure of entropy in a hashing function, specifically looking at how many bits change in the output when one bit in the input is changed. A perfect avalanche effect results in an average of half the output bits changing whenever only one bit is changed in the input. The outputs of this function change an average of a quarter of the bits when the input changes, leaving it with a half avalanche score.

Jenkins' blog also details a full-avalanche hash with more computation steps. I tested this and, while it worked perfectly, the results were no different to those using the faster half-avalanche hash. Because of this, I ended up using the half-avalanche hash to generate random vectors.

```
FVector2D PerlinNoise::randomVector(int16_t X, int16_t Y, int32_t seed)
{
    // Shift signed ints into positive range
    uint16_t x=(1<<15)+X, y=(1<<15)+Y;

    // Calculate hash of combined coordinates
    int32_t hashedSeed = Math::intHash(seed);
    uint32_t combinedCoordinates = (static_cast<uint32_t>(x)<<16) | (y);
    int32_t hash1=Math::intHash(combinedCoordinates^hashedSeed),
            hash2=Math::intHash(combinedCoordinates^hashedSeed^0x11111111);

    // Generate unit vector
    FVector2D vector(hash1, hash2);
    vector.Normalize();
    return vector;
}
```

Shown above (Hosted on GitHub) is the code I wrote to generate random vectors using the hash function. A 32-bit integer is formed from concatenating both 16-bit coordinates. Then, using the bitwise XOR of the integer and seed, two hashes are calculated, one with the bitwise inverse of the input. Finally, the two hashes are used as the X and Y coordinates of the output vector, which is normalised (magnitude scaled to 1) before being returned.

Using 16-bit coordinates for chunks allows the player to walk 1,310km before encountering an overflow error in terrain generation. I deemed this to be more than enough for the demonstration, but the function can be easily modified if more walking distance is required.

7.5 Sprint 4: High-Level Function

16th February – 28th February 2022

7.5.1 Sprint 4 Overview

The final sprint I completed lasted 13 days and focused on the higher-level aspects of terrain generation, such as biome blending and graphics. There were fewer specific tasks allocated to this sprint so that there would be time to work on generally polishing the existing features of the project. During this sprint, I completed the following objectives:

1. Implemented the different biomes of the world (7.5.2)
2. Implemented functionality and controls for high-level biome blending (7.5.3)

7.5.2 Biomes

Up until this point, all of the terrain generated used basic, fractal-layered Perlin Noise. The last big step was to generate biomes with different characteristics. Using nature-inspired features like Dwarf Fortress does (2.5) I generated four distinct biomes:

1. Cold and rainy: Temperate biome with moderately-sized hills
2. Cold and dry: Tundra biome with large, ridged mountains
3. Hot and rainy: Marshy biome with dark flat terrain, covered in streams
4. Hot and dry: Desert biome with sand dunes

Figure 31: Temperate biome. (Screenshot from the working demo)

```
float Biomes::temperate(float x, float y, int octaves)
{
    // Return standard, fractal-layered Perlin Noise
    return Math::fBm(x, y, PerlinNoise::standard, octaves, 1);
}
```

Figure 30: Temperate biome code (Hosted on GitHub)

The **temperate biome** was the simplest to construct. The biome uses fractal-layered Perlin Noise to create hills of varying sizes and detail. Shown in fig. 31, the math and Perlin Noise functions are easy and straightforward to use with minimal code.

Figure 32: Tundra biome (Screenshot from the working demo)

```
float Biomes::tundra(float x, float y, int octaves)
{
    // Blend together 1 part Perlin Noise with 1 part ridged noise
    return Math::fBm(x, y, PerlinNoise::ridge,    4,       1) / 2 +
           Math::fBm(x, y, PerlinNoise::standard, octaves, 1) / 2;
}
```

Figure 33: Tundra biome code (Hosted on GitHub)

To create the **tundra biome** with large, ridged mountains, I initially used fractal-layered ridge noise (2.1.4) on its own. However, this resulted in unnaturally sharp terrain. The ground was littered with tiny spikes that were impossible to traverse.

To benefit from the high-level, mountainous features of ridged noise while still being able to walk across the ground, I blended ridged and Perlin Noise. As can be seen in fig. 34, the ridged noise uses a constant four octaves, ensuring that small ridges aren't included in the final result. The ridged noise is mixed with equal parts Perlin Noise, resulting in the landscape seen in fig. 33.

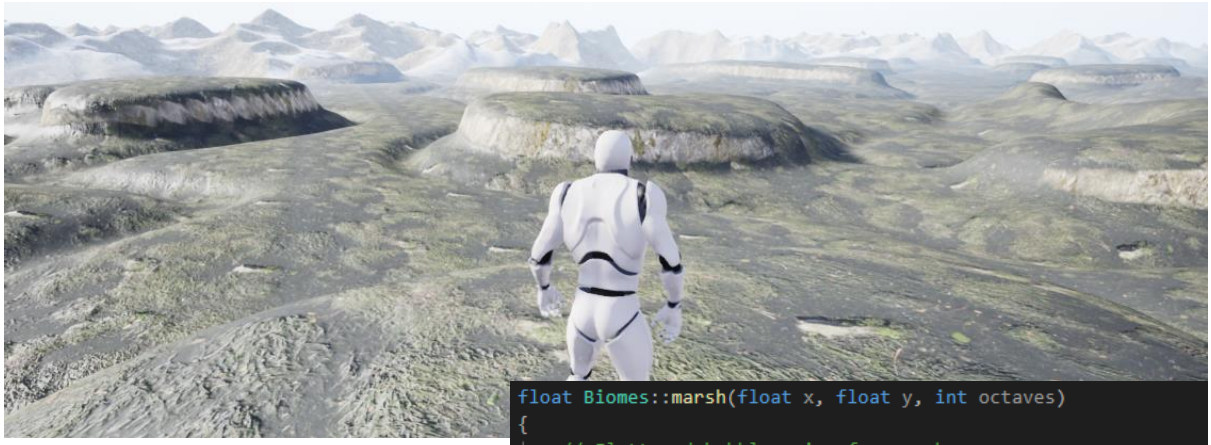


Figure 34: Marsh biome with small mesas. (Screenshot from the working demo)

The **marsh biome** was the hardest to design. I hadn't initially considered that marsh biomes are very flat and, without foliage, look very plain. To make the biome a little more interesting, I added small mesas.

```
float Biomes::marsh(float x, float y, int octaves)
{
    // Flattened bubble noise for marsh
    float marsh = Math::fBm(x, y, PerlinNoise::bubble, 1, 1);

    // Alter distribution for more drastic streams
    marsh = Math::kSigmoid(marsh, 0.9f);

    // Standard noise for mesas
    float mesa = Math::fBm(x, y, PerlinNoise::standard, 1, 1);

    // Alter distribution to lift mesas from ground
    mesa = Math::kSigmoid(mesa, 0.95f);

    // Blend marsh and mesa layers
    return marsh/60.0f + mesa/20.0f;
}
```

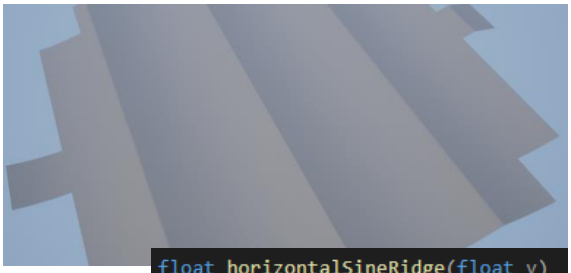
Figure 35: Marsh biome code (Hosted on GitHub)

To create the mesas, I created a layer of Perlin Noise and altered the distribution with a *tuneable sigmoid* (Emery, 2022) function I found online. The kSigmoid function pushes values above 0 closer to 1, and values below 0 to -1. By pushing terrain height values outward, mesa-like plateaus of high and low altitude can be created.

The streams for the marsh can be created by taking bubble noise, altering the distribution with the tuneable sigmoid, and flattening it. This creates small channels carved into the landscape.

Figure 36: Desert biome. (Screenshot from the working demo)

The **desert biome** is the most unique and complex biome I designed. It uses a wide array of techniques and is calculated in multiple parts.

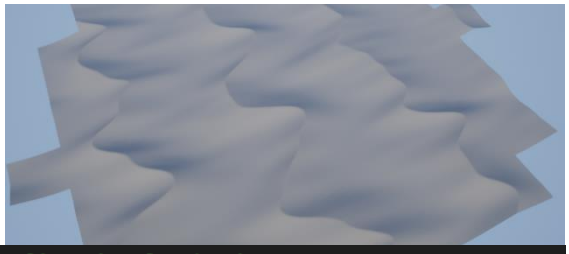


```
float horizontalSineRidge(float y)
{
    // Return sine-wave ridges
    return 2 - abs(sin(y*3.14))*2;
}
```

Figure 37: Sine wave ridges (Hosted on GitHub)

First, I created long, straight ridges extending infinitely across the map. As seen in fig. 38, this is done with *sine ridge* noise – using the same method as Perlin ridge (2.1.4). This is better for sand dunes than standard ridged noise as there are no loops.

Once the ridges have been created, I domain warp (2.3) them using a layer of Perlin Noise (fig. 39). This creates the effect of wind-shifted dunes of sand in a desert.



```
// Get Perlin Noise for domain warp
float dwX = Math::fBm(x,y, PerlinNoise::standard, 2, 1);

// Get two, separately domain-warped sine wave ridges
float frontOne = horizontalSineRidge(y + dwX);
```

Figure 38: Domain-warped ridges (Hosted on GitHub)

Although the result looks quite realistic in shape, it's unrealistic that all of the dunes are organised in infinite, unbroken lines.

To fix this issue, the algorithm does the whole process again, creating an entirely independent second dune layer. The algorithm then generates another layer of Perlin Noise which is used to interpolate these two dune layers into a final, consolidated output, as seen in fig. 40.

```
// Create third layer of noise for interpolating between sine ridges
float noise = Math::fBm(x, y, PerlinNoise::standard, 2, 1);

// Return noise based on noise interpolation
return Math::lerp(frontOne, frontTwo, noise);
```

Figure 39: Noise-interpolated dunes (Hosted on GitHub)

7.5.3 Biome Blending

Once all of the biomes were created, I implemented functionality for blending them into one world. Using one noise layer for temperature and one for precipitation, I interpolated between biome height functions based on the following table:

Biome Attributes	Low Temperature	High Temperature
Low Humidity	Temperate	Desert
High Humidity	Tundra	Marsh

By interpolating between the four biome height functions based on heat and precipitation, different biomes adopted different shapes. However, they still needed to be visually textured differently.

The texturing is calculated as an Unreal Engine material, so I needed a way to tell the material what heat and precipitation each vertex has. As I was using textures to colour the terrain, the vertex colour attribute was going unused. I encoded heat into the red channel of the vertex colour attribute, and precipitation into the blue channel. The Unreal Material then interpolates between the four biomes' textures using the red and blue vertex channels as interpolants.

Finally, so that the sharpness of biome blending can be precisely controlled, both the heat and precipitation attributes are put through the tuneable sigmoid function. Once again, this pushes the values to higher and lower limits, making sharpness of the blends between biomes controllable.

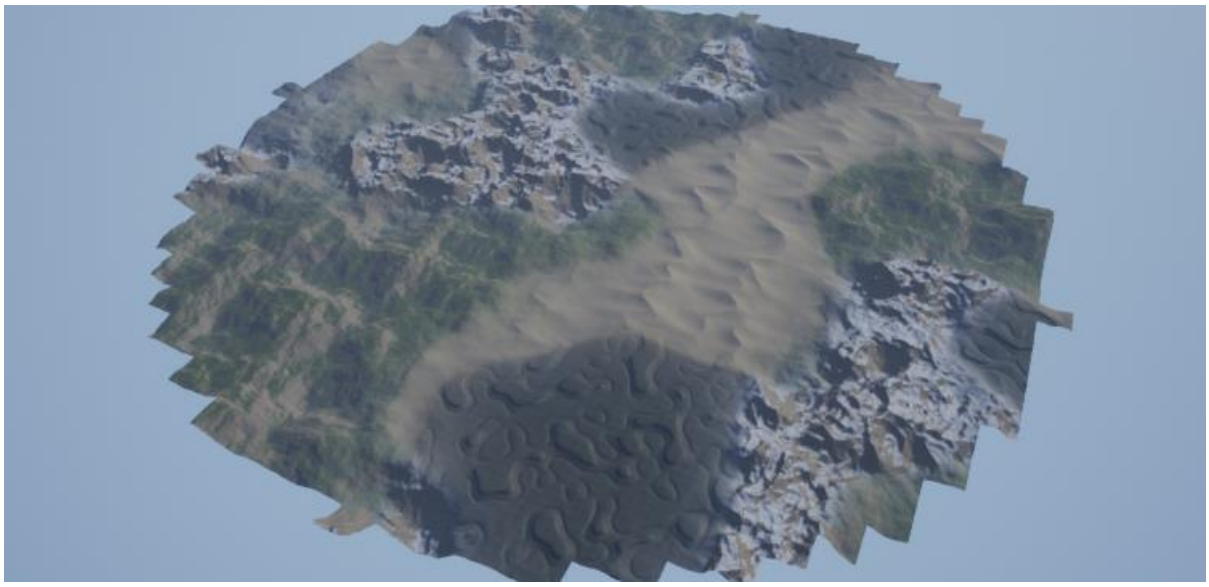


Figure 40: Bird's-eye view of a segment of multi-biome world. (Screenshot from the working demo)

Chapter 8: Testing and Evaluation

8.1 Evaluation Overview

This section aims to evaluate the developed system based on the following metrics, defined in Chapter 5: Evaluation Strategy:

1. Terrain Resolution (5.1.1) – measured in mesh vertices per metre.
2. Number of attributes used (5.1.2) – attributes such as precipitation, heat etc.
3. Number of simulation iterations (5.1.3) – how long terrain simulations should last
4. Number of distinct biomes (5.1.4) – hills, forest, mountains etc.
5. How long the world takes to load (5.2.1) – measured in seconds
6. Frame rate (5.2.2) – measured in frames per second

To ensure a fair evaluation of the system, I only used one parameter configuration for all cases of the testing. This means that all tests used the same terrain resolution, view distance etc.

The application was tested on two systems, using the lower of each score for results such as loading time and frame rate:

System	**CPU**	**GPU**	**Memory**
2018 MSI GE63 Laptop	Intel Core i7-8750H	Nvidia GTX 1060 6GB	16GB
2020 MSI GS66 Laptop	Intel Core i7-10750H	Nvidia RTX 2070 8GB	16GB

As expected, the newer GS66 system outperformed the GE63 in all performance benchmarks. However, the difference was never more than 10% and the older system still passes all performance requirements set out in Section 5.2.

8.2 Terrain Resolution

The target terrain resolution for the demo was set in Section 5.1.1 and aimed to match Unreal's default resolution of **1 vertex per square metre**. The final demonstration I implemented uses a much higher resolution of **6.25 vertices per square metre**. This significantly exceeds what I had planned to support while still achieving good performance (8.7).

8.3 Number of Attributes Used

To match Dwarf Fortress' realism, Section 5.1.2 outlined the number of attributes used in terrain generation as an evaluation metric. The final demonstration uses 2 independent attributes: heat and precipitation. These attributes are then used to generate the four biomes. Although not as many as Dwarf Fortress', these attributes helped to split up the world into natural-looking biomes with randomly generated borders.

8.4 Number of Simulation Iterations

The number of simulation iterations was an evaluation based on the could-have requirement of teleological enhancement (NFR-4). As I didn't have enough time to implement this requirement, there was no need to evaluate the application in this way.

If teleological methods were to be used for terrain enhancement, the computation time would likely be affordable. As discussed in Section 7.4.3, custom calculations such as vertex generation are bottlenecked by mesh generation. Only two CPU cores are used for this demonstration so most modern computers would have a large amount of power left to compute teleological simulation on one or more additional cores.

8.5 Number of Distinct Biomes

The initial requirements stated **three biomes as a should have** (FR-6.1), and **five biomes as a could have** (FR-6.1). The final demonstration implements **four distinct biomes**, fulfilling the should-have requirement. Although a fifth biome would have been reasonably easy to implement, implementing five-biome blending functionality would have taken much longer and I decided that the time could better be spent elsewhere.

8.6 World Load Time

The requirements NFR-1.1 and NFR-1.2 specified a must-have load time **under 60 seconds** and a should-have load time **under 30 seconds**, respectively. The final implementation has a *100% load time* of **5 seconds**.

The 100% load time is the time it takes to load all chunks in view. The chunk the player is standing on is ready in around 20 milliseconds, allowing the player to drop into the game while the farther chunks are still loading.

8.7 Frame Rate

The project was set up to limit frame rate at 60Hz. This is so that both of the computers used for testing could run it with the same configuration. Additionally, limiting the frame rate and using double buffering for graphical frames improved the overall frame rate stability.

As discussed in Section 5.2.2, there are two different scenarios in which the frame rate should be tested:

1. While standing still – no chunks are being generated
2. While moving – chunks are being generated in real time, putting strain on the system

In both scenarios, and on both systems, a median frame rate of 60Hz was achieved. The following table displays the breakdown of this metric over time:

	Standing Still			Moving		
System	Median FPS	Mean FPS	Lowest 5th Percentile FPS	Median FPS	Mean FPS	Lowest 5th Percentile FPS
2020 GS66	60.0	59.2	56.7	59.9	58.9	55.8
2018 GE63	59.8	58.8	54.2	59.6	57.9	54.0

The stability of the frame rate is largely due to the application's *deferred generation* system. If chunks still need to be loaded when the frame's time is up, the generation will be deferred (7.2.4) until there's more time. This results in the frame rate remaining constant, while the chunk loading time (8.6) varies with system load.

Chapter 9: Conclusion

9.1 Achievements

I set out with four initial objectives for this project, and each has been achieved over the course of semester two:

1. I have researched various methods and applications of real-time procedural generation (9.1.1).
2. I have designed and planned the development of a technical demo that generates terrain in real time (9.1.2).
3. I have carried out the implementation of the designed system (9.2.3).
4. I have tested and evaluated the developed system using technical benchmarks (9.2.4).

Additionally, I have published the project online for others to use and learn from (9.1.5).

9.1.1 Research

Over the course of this project, I have carried out research on various articles, papers and blogs, detailing the different techniques used in procedural generation and how they can be applied to games.

I have deepened my knowledge in the fundamentals of procedural generation such as Perlin Noise and Fractal Brownian Motion. I have also broadened my knowledge of new techniques such as Domain Warping and Simplex Noise.

Researching and reviewing games like No Man's Sky and Dwarf Fortress have demonstrated to me the amount of detail put into procedurally generated games, and how the smallest details can make such a large difference in the final game.

9.1.2 Planning

To ensure that the project stayed on track with minimal issues, I laid out an extensive plan, detailing technical tools, code architecture, requirements analysis, risks and ethical concerns.

Spending the time to plan the development ensured that, when I did have issues, I knew what mitigations had to be done to stay on schedule.

9.1.3 Implementation

The following table shows how many of each type of requirement, laid out in Chapter 3, were completed during the implementation stage:

MoSCoW Rank	Implemented	Not Implemented
Must Have	4	0
Should Have	7	1
Could Have	1	3

A detailed breakdown of each requirement can be found back in Chapter 3: Requirements Analysis, with the specification of each requirement and links to the sections that detail their implementation.

9.1.4 Evaluation

I have evaluated my application using various metrics set out in Chapter 5 such as frame rate, loading time and terrain resolution. These evaluations have helped to outline the strengths and weaknesses of the application, and where future work may be applied.

Using these evaluations, the next section (9.2) outlines some of the areas where I believe could be worked on in the future.

9.1.5 Open Source

The final application is open source, licensed under the GPL3 license. Other creators are free to use and learn from my work with procedural generation in Unreal Engine 4.

All of the functionality of procedural terrain in my project is bundled up in a standard Unreal component. That means that, for other people to use my infinitely generated terrain in their game, all that needs to be done is to click and drag the bundled component into their level. No extra plugins or configurations are required.

9.2 Limitations and Future Work

This section details some of the limitations of the application, as well as future work that may be implemented to further improve it. The primary limitations of the system are as follows:

1. Currently, there is no way to generate chunks of terrain any faster (9.2.1)
2. View distances beyond 220 metres result in significantly reduced frame rate (9.2.2)
3. The system only supports ontogenetic approximation for terrain generation, not teleological simulation (9.2.3).
4. The system only generates the shape of the terrain. It does not provide a way to populate biomes with features such as foliage, rocks or water (9.2.4).

9.2.1 Terrain Generation Speed Limit

The only limitation that's largely unavoidable is the terrain generation speed limit. While this could be avoided by using another engine or graphics API, the project designed in Unreal Engine 4 can't avoid the bottleneck created by real-time mesh generation (7.4.3). This issue does not stop high-quality terrain being generated in real time, but it does mean that concessions such as shorter view distances must be made to maintain adequate loading time and frame rates.

As much of the work involved in implementing this project was based in C++ and not Unreal, a lot of functionality could be moved to another engine or API without too much work. The primary challenge in porting the project to another environment would be changing the usage of Unreal's proprietary library (7.2.3) to either the C++ standard library, or another engine's proprietary library.

9.2.2 View Distance

Currently, the system can handle a view distance of around 220 metres before the frame rate falls below the target 60Hz. As detailed in Section 7.4.4, this is adequate but it could be much more.

While this limitation is in part due to the time it takes Unreal to generate meshes, the number of polygons on screen at once has a big impact on rendering time. One solution to this limitation would be to implement a *level of detail system* (shortened to LOD).

A level of detail system would dynamically change the terrain resolution (5.1.1) based on how far away the chunk of terrain is, resulting in far fewer total polygons having to be rendered each frame. Such a system was considered during development, but I decided against it in favour of

implementing other functionality in the limited time. In theory, a highly efficient LOD system could improve view distances by an exponential factor.

9.2.3 Teleological Simulation

One of the initial could-have requirements of the system was the use of teleological simulation in real time. I detailed in Section 4.4 how this simulation could be done while the world is being generated. However, time constraints in terms of delivering the demo meant this could-have requirement was rejected. While interesting landscape features can be made with ontogenetic approximation alone, the use of teleological simulation would allow for the generation of much more complex, physically based features.

I believe that the use of teleological simulation could be implemented in real time on an infinite world. This would allow for more interesting features such as waterfalls and cliffs to be formed in a realistic and natural way.

9.2.4 Foliage, Rocks and Water

I had initially deemed the generation of features such as foliage, rocks and water to be outside the scope of this project, thinking that realistic terrain can be made with only the shape and texture of the ground. However, after designing different biomes reflecting real-world environments, I believe that the population of landscapes with these features is very important for generating interesting and lifelike worlds.

With future work, I would like to research how procedurally generated landscapes can be algorithmically populated with these features, and explore how they can be incorporated into the environment in a realistic and believable way.

9.3 Project Benefits

On a personal level, undertaking this project has provided me with invaluable knowledge and experience managing the development of a complex system from start to finish. I have learned how to plan both the technical and logistical aspects of development, how to identify and assess risk, and how to test and evaluate a finished system.

I have learned that nothing exists in a vacuum – *especially in the field of videogame development*. Through practical experience developing my game demo, I have discovered the importance of game engine choice, and the tight coupling it has with the designed system. Although I set out to implement my technical demo in an environment-agnostic manner, I have instead realised the specific strengths and weaknesses of procedural generation using Unreal Engine 4.

On a technical level, I believe that my open-source project has identified some areas of importance for developers seeking to use Unreal Engine 4 for real-time procedural generation. In the future, I intend on continuing this project by publishing material documenting the specific strengths, limitations and mitigations of Unreal Engine 4 in the context of procedural generation.

References

bay12games.com. (2021, 11 20). *bay12games.com.* Retrieved from Bay 12 Games: https://www.bay12games.com/dwarves/screens.html

Clement, J. (2019, August). *What game engines do you currently use?* Retrieved from Statista: https://www.statista.com/statistics/321059/game-engines-used-by-video-game-developers-uk/

Conrod, J. (2010, 06 02). *Water simulation in GLSL.* Retrieved from jayconrod.com: https://www.jayconrod.com/posts/34/water-simulation-in-glsl

cppreference. (2021, March 21). *RAII*. Retrieved from cppreference: https://en.cppreference.com/w/cpp/language/raii

cppreference. (2021, October 15). *The rule of three/five/zero*. Retrieved from cppreference: https://en.cppreference.com/w/cpp/language/rule_of_three

Curry, D. (2022, January 11). *Minecraft Revenue and Usage Statistics*. Retrieved from Business of Apps: https://www.businessofapps.com/data/minecraft-statistics/

Dobrilova, T. (2022, March 14). *How Much Is the Gaming Industry Worth in 2022?* Retrieved from Tech Jury: https://techjury.net/blog/gaming-industry-worth/#gref

Emery, D. H. (2022, 04 12). *Normalized Tunable Sigmoid Function*. Retrieved from GitHub Pages: https://dhemery.github.io/DHE-Modules/technical/sigmoid/

Epic Games. (2022, 04 09). *UProceduralMeshComponent::CreateMeshSection_LinearColor*. Retrieved from Unreal Engine: https://docs.unrealengine.com/4.27/en-US/API/Plugins/ProceduralMeshComponent/UProceduralMeshComponent/CreateMeshSectio-/2/

Gustavson, S. (2005, September 30). Simplex noise demystified. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/OpenSimplex_noise

Hall, C. (2014, July 23). *Dwarf Fortress will crush your CPU because creating history is hard*. Retrieved from Polygon: https://www.polygon.com/2014/7/23/5926447/dwarf-fortress-will-crush-your-cpu-because-creating-history-is-hard

Jenkins, B. (2022, April 10). *Integer Hashing*. Retrieved from burtleburtle.net: https://burtleburtle.net/bob/hash/integer.html

Kollar, P. (2016, August 12). *No Man's Sky Review*. Retrieved from Polygon: https://www.polygon.com/2016/8/12/12461520/no-mans-sky-review-ps4-playstation-4-pc-windows-hello-games-sony

Lague, S. (2019, February 28). *Coding Adventure: Hydraulic Erosion.* Retrieved from Youtube: https://youtu.be/eaXk97ujbPQ?t=254

Li, V. (2018, 12 28). *Perlin Noise Implementation.* Retrieved from 1000 Forms Of Bunnies: http://viclw17.github.io/2018/12/28/Perlin-Noise-Implementation/

Metacritic. (2021, November 6). *No Man's Sky*. Retrieved from Metacritic: https://www.metacritic.com/game/pc/no-mans-sky/user-reviews

Michael, J. (2016, December 7). *6 of the Most Awkward Looking Creatures in No Man's Sky.* Retrieved from Game Skinny: https://www.gameskinny.com/b8u00/6-of-the-most-awkward-looking-creatures-in-no-mans-sky

Mojang. (2022, 4 10). *Minecraft Options*. Retrieved from Minecraft Wiki: https://minecraft.fandom.com/wiki/Options

Perlin, K. (1985). An Image Synthesizer. *SIGGRAPH Comput. Graph.*, 1.

Perlin, K. (2002). Improving Noise. *SIGGRAPH Comput. Graph.*, 1-2.

Perlin, K. (2002, July 29). Noise Hardware.

principles-wiki. (2021, October 20). *Keep It Simple Stupid*. Retrieved from principles-wiki: http://principles-wiki.net/principles:keep_it_simple_stupid

Quilez, I. (2002). *Domain Warping*. Retrieved from iquilezles: https://www.iquilezles.org/www/articles/warp/warp.htm

Quixel. (2022, 04 09). *MOSSY STONE FLOOR*. Retrieved from Quixel: https://quixel.com/megascans/home?category=surface&category=historical&assetId=veilfjxn

Reddit. (2016, September 25). *Necromancer tower.* Retrieved from Reddit: https://www.reddit.com/r/dwarffortress/comments/54dx7e/attacking_that_annoying_necromancer_tower/

Scratchapixel. (2021, 11 20). *Value Noise and Procedural Patterns: Part 1.* Retrieved from scratchapixel.com: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/procedural-patterns-noise-part-1/simple-pattern-examples

Tak, A. (2015, July 1). *Building Procedural Terrain.* Retrieved from abhimanyutak.blogspot.com: Perlin noise in Real-time Computer Graphics

Tatarinov, A. (2008). *Perlin noise in Real-time Computer Graphics.* Retrieved from Semantic Scholar: https://www.semanticscholar.org/paper/Perlin-noise-in-Real-time-Computer-Graphics-Tatarinov/b49da45b19f6ad6c28b3748223b715810711d15f/figure/0

ThinkAutomation. (2021, 11 19). *Is automation ethical?* Retrieved from thinkautomation.com: https://www.thinkautomation.com/automation-ethics/is-automation-ethical/

ThinMatrix. (2017, Sept 12). *OpenGL Low-Poly Terrain Tutorial*. Retrieved from YouTube: https://www.youtube.com/watch?v=l6PEfzQVpvM

Vivo, P. G. (2015). Fractal Brownian Motion. *The Book of Shaders*.

Vivo, P. G. (2021, 11 20). *Noise.* Retrieved from The book of shaders: https://thebookofshaders.com/11/

Wikidot. (2020, July 31). *Teleological vs. Ontogenetic*. Retrieved from Wikidot: http://pcg.wikidot.com/pcg-algorithm:teleological-vs-ontogenetic

Wikipedia. (2021, November 12). *Fair use*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Fair_use#United_Kingdom

Wikipedia. (2022, February 6). *Polygon mesh*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Polygon_mesh

World of Level Design. (2020, April 29). *UE4 Heightmap Guide: Everything You Need to Know About Landscape Heightmaps for UE4*. Retrieved from worldofleveldesign.com: https://www.worldofleveldesign.com/categories/ue4/landscape-heightmap-guide.php

www.bit-101.com. (2021, July 17). *PERLIN VS. SIMPLEX.* Retrieved from www.bit-101.com: https://www.bit-101.com/blog/2021/07/perlin-vs-simplex/

Appendices

Appendix 1: Project Plan Gantt Chart