Comparative Analysis of Algorithms in Procedural Generation for Endless Terrain Creation

Abubakker Asim (21021406)

May 2024

G450 BSc Computer Science (Game Eng)

Project Supervisor Prof Graham Morgan

Word Count : 13,496

Abstract

This Dissertation provides a detailed analysis of the procedural generation algorithms, made to develop endless terrains in Unity3D, has been presented. The main motivation behind this paper was to develop procedural terrains that could achieve high graphical fidelity without sacrificing computational power. An experimental methodology was applied to run algorithms, including Perlins Noise and revolutionary algortithms such as Procedural Content Generation algorithms (PCG). The results showed which of them may be used to create visually pleasing and wide-scale digital terrains. Furthermore, it was essential to develop a prototype in Unity3D, in which the algorithms were implemented as a showcase of their direct application. The extensive testing, including performance evaluation on different systems and several simulations of environmental situations allowed drawing the necessary conclusions about the impact of each algorithm on CPU and GPU utilization and memory consumption. The results showed that procedural generation allows achieving a high level of scalability and efficiency and using both Perlin Noise and probabilistic models based on GANs may contribute to a significant increase in terrains' realism and details, preserving system performance. Research should benefit game developers as a direct comparison of several procedural generation techniques to improve graphical quality.

Declaration

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

Contents

1 Introduction

This section introduces the research undertaken in the procedural generation of expansive terrains within Unity3D, outlining the interaction between advanced algorithms and enhanced graphical fidelity and structuring this dissertation accordingly.

1.1 Purpose

In the fast-paced world of digital media, the autonomous generation of realistic game environments of huge sizes has been crucial for ensuring fully immersive user experiences. Following the groundbreaking work on noise by Ken Perlin [1] is allowing noise algorithms to influence the kind of realism that could be found in nature. Nevertheless, such an achievement inevitably has a price, the price being high computational cost, seemingly, at times, requiring the choice between graphical detail and computational efficiency. The motivation for this dissertation stems from the necessity to develop a procedural generation tool to produce a terrain that not only looks aesthetically pleasing but does not strain computational resources.

1.2 Project Aim and Objectives

The primary goal of this dissertation is:

Implement and create a computationally efficient Procedurally Generated Terrain using Unity3D, emphasising graphical quality and performance efficiency.

To achieve this, the project has been structured around three core objectives:

To conduct an in-depth examination of existing procedural generation methodologies and tools, focusing on Perlin Noise, Procedural Content Generation Algorithms, and other contemporary techniques, identifying the most effective strategies for terrain generation.

To design and execute a prototype within Unity3D that demonstrates the practical application of these algorithms, emphasising adaptability, graphical quality, and infinite terrain exploration.

To integrate an adaptive level of detail and evaluate the performance of the terrain generation process, ensuring optimal balance between high-quality graphics and system efficiency.

Through this study, I aim to produce a procedural terrain generation framework that enhances virtual environments' visual and interactive quality and remains accessible and efficient for content creators and users.

1.3 Dissertation Outline

Introduction

An introduction to the dissertation detailing the problem domain of creating vast, interactive terrains in digital media and explaining the project's aim and objectives:

Purpose

- Address the need for computationally sustainable terrain generation.
- Focus on detailed and dynamic environments.
- Expand upon foundational algorithms like Perlin's noise.
- Explore the use of machine learning for adaptive terrains.

Project Aim and Objectives

- Enhance procedural generation within Unity3D.
- Achieve robustness and efficiency for gaming industry applications.
- Concentrate on improving graphical quality and performance.

Dissertation Outline

- Detail the dissertation's structure.
- Trace the progression from initial concept to procedural terrain generation system creation in Unity3D.

Research

Investigation into the field of procedural terrain generation, including a comprehensive review of literature and correspondence with industry experts:

Research Strategy

- Conduct a literature review on existing methodologies.
- Focus on procedural content generation systems.
- Assess their role in digital environments.

Background Research

- Study the evolution of algorithms, notably Perlin Noise.
- Explore advanced techniques for terrain generation, such as Generative Adversarial Networks.
- Explore an application that uses Procedural Generation techniques in Terrain Generation.

Methodology

An elaboration of the methodologies employed in this research:

Elicitation

- Identify core requirements for procedural terrain generation.
- Recognize challenges in terrain generation processes.

Engine Comparison

- Evaluate Unity3D and Unreal Engine 5 for terrain generation.

- Compare graphical fidelity and performance metrics.
- Analyze developer experience and workflow efficiency.
- Assess optimisation capabilities for different hardware.

Development

- Develop a prototype showcasing diverse terrain generation.
- Focus on improving texture quality.

Testing &Results

System Analysis

- Test generated terrains for environmental fidelity and diversity.

Black Box Testing

- Conduct functionality tests to verify output against desired criteria without examining the internal algorithm structure.

Environment Testing

- Assess terrain response to user interaction and modifications.

System Testing

- Evaluate computational usage across different devices.

Evaluation

Critical assessment computational usage of the procedural generation tool:

User Testing

- Gathering results from tests run where different devices interacted with the terrain to evaluate computational resources
- Gather insights for improvements.

Conclusion

Summarising the dissertation and reflecting on the research outcomes:

Fulfilment of Objectives

- Review whether the research met its initial aims and how effectively the objectives were achieved.

Personal Development

- Reflecting on the skills and knowledge acquired through this research process.

Future Work

- Identifying potential areas for future research and development to continue advancing the field.

2 Research

This section summarises the substantial research during this dissertation, focusing on procedural terrain generation based on an extensive literature review. It includes a research strategy that involves a review of revolutionary works, investigations, and sources on modern methodologies combined with an analysis of an application that uses procedural generation systems in their different stages, discusses the evolution of Perlin noise and the technology that powers it in digital terrains since its first implementation, and looked into the current use of trending and advanced technologies such as GANs in terrain generation. Analysis of a case study of the procedural terrain generated in one of the world's most popular video games, Minecraft, to understand how well procedural generation tools are used in digital media.

2.1 Research Strategy

Academic papers on procedural terrain generation were acquired and thoroughly analysed to understand the fundamentals of procedural generation techniques and explore opportunities for further advancements. The research was conducted through two principal approaches: an in-depth review of seminal papers and practical testing with software tools that implement these techniques. Utilising Google Scholar, a vital academic search engine, four pivotal papers were identified that directly link to the concept of procedural generation. Among these, one particularly influential paper was thoroughly examined for its methodologies and findings. Additionally, the use of a specific tool designed to simulate procedural generation allowed for a practical understanding of procedural generation techniques used in the world of digital media.

This dual-faceted research strategy of combining theoretical exploration with practical application proved to be highly effective as academic papers provided a solid foundation of theoretical knowledge and highlighted various advancements and challenges within the field. The papers were instrumental in offering a deeper understanding of procedural techniques and identifying areas ripe for further exploration. Meanwhile, it was understood that software tools offered tangible insights and a deeper practical understanding of procedural generation. This hands-on engagement with a video game renowned for its use of procedural generation technologies aided with an evaluative perspective on the current applications and user experiences associated with such tools. Through this gameplay experience, it was possible to draw parallels between theoretical assertions and their practical implementations, allowing an understanding of the strengths and limitations of existing procedural generation tools in video games.

It was deduced that by leveraging both academic research and practical experimentation, this approach bridges the gap between theoretical knowledge and practical application, understanding the effectiveness of techniques in the field. It is anticipated that through testing this tool, key features will be identified for the procedural generation tool that is being implemented.

2.2 Background Research

2.2.1 Ken Perlin's Seminal Contribution: An Image Synthesizer

The foundational element of this research initiative is a comprehensive analysis of Ken Perlin's seminal work, "An Image Synthesizer" [1]. Within this influential paper, Perlin noise is introduced as an innovative procedural algorithm critical for generating textures that exhibit a pseudo-random visual complexity like that found in natural settings. This algorithm has been instrumental in setting a

computational benchmark within graphics programming, significantly influencing the spectrum of procedural generation techniques. Perlin noise is mainly known for its computational efficiency and its ability to produce gradients of noise that are spatially coherent yet lack apparent periodicity and visual artefacts. This makes it a quintessential tool in the repertoire of computational graphics and simulation, with applications that span from real-time gaming rendering to the stochastic simulation of natural textures. Its mathematical robustness and algorithmic versatility have solidified Perlin's contributions as a cornerstone in the study of computer graphics and procedural modelling.

2.2.2 Advancements in Machine Learning: Wulff-Jensen et al. on DCGANs

The study by Wulff-Jensen et al. signifies a contemporary advancement in procedural terrain generation [2]. Their incisive exploration into deep convolutional generative adversarial networks (DCGANs) showcases the profound collaborations achievable by integrating machine learning paradigms with procedural generation methodologies. The implementation of DCGANs underscores a considerable leap in the computational synthesis of terrain, providing a framework that leverages deep learning architectures to autonomously learn and replicate the statistical distributions inherent in natural landscapes. This research illuminates the capabilities of DCGANs to produce terrain formations that are diverse in their topological features and exhibit a heightened level of realism, potentially transforming the procedural terrain generation pipeline into a more adaptive and data-driven process. The findings of Wulff-Jensen et al. contribute to the computational graphics discourse by demonstrating how advanced neural network models can be trained to understand and recreate complex environmental patterns, thereby pushing the frontier of automated digital environment creation.

2.2.3 User-Controlled Terrain Synthesis: Guérin et al. and cGANs

Guérin et al.'s research represents a quantum leap in procedural design, focusing on applying Conditional Generative Adversarial Networks (cGANs) to terrain synthesis [3]. Their approach introduces an evolutionary step in the design methodology, enabling an intuitive interface that empowers creators with unprecedented control and streamlines the design pipeline. By integrating the conditional aspect into the generative adversarial network framework, this study harnesses the power of machine learning to adhere to specific user-defined constraints, yielding a procedural generation process that is both flexible and precise. This method allows for the generation of complex terrain that is not only algorithmically sound but also tailored to the aesthetic and functional requirements of the end-user. The work of Guérin et al. is emblematic of the shift towards a more interactive and user-centric paradigm in procedural generation, where the complex underlying computational processes are made accessible and controllable through higher-level design abstractions.

2.2.4 Evaluation Methodologies in PCG: Summerville's Framework

Summerville’s [4] contribution is groundbreaking as it pioneers a critical framework for the empirical evaluation of PCG systems. This approach secures a functional set of analysable metrics that form the PCG system’s described limits and execution boundaries. Such a toolkit underlines the ability to meaningfully evaluate PCG systems about the algorithm, which has been performed on a machine-learning level. In this aspect, Summerville clearly and systematically elaborates on the functional basis on which the set limits, providing researchers and developers with a benchmark-focused toolkit to evaluate their output while maintaining the content’s pre-centralized metrics. This is new as it does not yet exist for a serialisation uniform of evaluation, shedding light on performance systems, which have primarily hovered under swathes of approximation.

The selection of synthesised pivotal studies connects a direct line of development from Perlin's pioneering work to integrating machine learning technologies in PCG. It is the purposeful representation of the assembled research results meant to highlight the revolutionary importance of PCG systems for the digital creative sector and their lasting contribution to the ongoing development of the game-making process.

2.2.5 Evaluation of Software: Minecraft (2011)

One enlightening case study in this regard is Minecraft [5]. This game explores a novel procedure of generating algorithms paramount to the user's goal of creating endless explorable terrain. Minecraft can make such an expansive and dynamic world by efficiently using Perlin and Simplex noise functions to give each action a unique terrain feature computed from a noise function. Minecraft, for example, is instructive about how noise functions can create an algorithmic world, which ensures a diversity of landscapes, each with precise characteristics such as rolling hills or epic deserts, all from small numeric tablets.

The noise algorithms behind Minecraft's terrain complexity and scalability underpin the iconic blocky aesthetic and allow for theoretically unlimited exploration by the player. Similarly, the project aims to generate terrains that recede into the horizon and feature the level of detail necessary to encourage exploration and interaction. The project can draw on Minecraft's procedural approach to noise to learn how similar functions can be dialled in and stacked to characterise diverse ecosystems and topographies dynamically generated in real-time as the player navigates the virtual world.

Engaging with Minecraft's overwhelming expanse during gameplay prompted thoughts about my project's potential and limitations in using similar procedural generation techniques. Specifically, the game's effective use of noise functions to direct the distribution and shape of the environment provides an excellent example of the project's goals. The use case suggests that algorithms-generated landscapes can still feel authentic and multilayered, which are core qualities to target during emulation.

The gameplay experience, with its scenery stretching into the horizon, translates directly into workable insights regarding the game's design strategy. The way Minecraft presents a natural evolution of its terrain, constantly causing players to feel a sense of discovery and awe, serves as the perfect manifestation of the type of experience the creators would like their procedural generation to deliver. Whether through the further study of Minecraft's method or a reflective inspiration, the project should strive to apply an organised structure to its procedural generation process, ensuring each valley peak or plateau holds the naturalistic aura necessary for an engaging and immersive experience. Minecraft's procedural generation practices significantly contributed to my project's design framework. It deepened my commitment to creating a terrain exquisitely broad yet finely etched with naturalistic formation principles.

The concept of chunk generation used in Minecraft, which divides the game's world into distinct, manageable pieces, directly implies the project's resource efficiency, and the overall system performance is pivotal. Minecraft's system, which processes or renders solely the chunk of the terrain surrounding the player, is a blueprint for the project's procedural terrain generator's optimisation; using these concepts, the project can remain performant without putting an undue burden on computational resources.

Using noise maps in Minecraft, the method to determine biomes develops terrain characteristics. This method particularly applies to one of the project's objectives: developing contextually adaptive terrains. Minecraft's procedural system forms different environmental factors, such as biomes, which significantly contribute to the appearance of the terrain. This methodology could be extrapolated to develop a vibrant environment that adapts to in-game ecological factors.

Furthermore, the block textures generated by Minecraft highlight the possibility of using procedural techniques to enable real-time texture generation in the project. The game introduces a change of texture depending on different biomes with minimal manual adjustments, so this methodology has much potential.

Strengths of Minecraft's procedural generation system, relevant to the critical evaluation, include:

- Procedural generation's efficiency; the system can adopt vast and unique terrain with minimal storage requirements.
- Algorithmic randomness, enabled by a reproducible seed, allows the player to have unique "runs" while maintaining a sense of overall consistency.
- Performance optimisations, such as chunk generation; Minecraft's case indicates that the procedural systems of complex content generation can be managed effectively.

The lessons learned from Minecraft's procedural terrain and texture generation are directly relevant to the proposed project. They illustrate the potential of procedural techniques to create expansive, detailed terrains and emphasise the need to balance the algorithmic emphasis with a focus on player engagement. Finally, the case study inspires the proposed project's exploration of procedural generation algorithms such as cGANs and DCGANs, which were not employed in Minecraft but represent advanced options for achieving further realism and variety in terrain representation.

Taking on the procedural model from Minecraft will guarantee the robustness and efficiency of the terrain generation method in the project. This will enhance the fun for the player, as Minecraft has maintained a large following since its inception. This case study represents a crucial aspect of the literature review for the project, as it not only proves the capability of procedural generation but also challenges and explores the possibilities for researching new applications and methodologies.

2.2.6 Conclusion of Background Research

This review of seminal research elucidates a coherent trajectory of technological evolution in the procedural content generation (PCG) field, tracing a path from Perlin's foundational algorithm to the incorporation of sophisticated machine learning technologies in PCG. The collective analysis rigorously highlights the transformative influence of PCG systems within the digital creative industry, underpinning significant advancements in game development and related areas. This work illustrates the strategic integration of classic procedural algorithms with contemporary machine learning models, revealing a progressive shift towards more intelligent and adaptive content generation mechanisms. The research convergence presented here is a testament to the enduring impact and critical importance of PCG systems, which continue redefining the possibilities of digital creativity and interactive entertainment.

3 Methodology

This section outlines the strategies and foundations for choosing and implementing procedural generation methods performed using the Unity3D framework. The procedure follows strong research

ethics and involves advanced algorithmic decisions that align with ethical guidelines applicable to the software development process.

3.1 Overview

The main tasks in this section are:

- Research into the project area (Background Research)
- Planning the project
- Project Requirements
- Engine Comparison
- Software design and development

3.2 Planning

The Incremental Model approach was chosen. The incremental model is a software development model like a waterfall model. It is a structured construction methodology that allows for variation in the development of various periods. It is beneficial for tasks that need iterative refining and gradual changes. Developing a procedural generation tool, especially for endless terrain systems, guarantees that development is structured and data-driven modifications are scheduled to permit systemic adjustments, demonstrating a full grasp of the power of procedural algorithms used.

The background research helped to lay the way for an iterative approach, with the bulk of it completed at the beginning of the second semester. In contrast, the remaining work was completed during the lifespan of implementation. It began on February 14th, 2024, with the beginning of the projects, and ended on April 8th, 2024. Therefore, there was ample time to test and refine each module to ensure all components met the quality and project requirements outlined.

3.3 Project Requirements

3.3.1 Functional Requirements

3.3.1.1 FR1: Basic Terrain Generation

- **FR1.1**: Generate a noise map using Perlin noise
- **FR1.2**: Create a grid that outlines the terrain structure.

3.3.1.2 FR2: Basic Visualization

- **FR2.1**: Assign colours to the noise map based on heightmap elevation to differentiate terrain features.
- **FR2.2**: Convert the noise map into a 3D mesh representing the terrain.

3.3.1.3 FR3: Dynamic Level of Detail and Endless Terrain

- **FR3.1**: Implement a Level of Detail (LOD) system to adjust dynamically based on players' view distance.
- **FR3.2**: Develop logic for generating terrain chunks dynamically for endless terrain.

3.3.1.4 FR4: Performance Optimization

- **FR4.1**: Utilise multithreading for background terrain generation to ensure smooth gameplay.
- **FR4.2**: Refine the LOD system to ensure seamless transitions between detail levels.

3.3.1.5 FR5: Seamless Transitions and Advanced Features

- **FR5.1**: Implement seam handling to align terrain chunks.
- **FR5.2**: Apply a falloff map to smooth terrain edges.

3.3.1.6 FR6: Enhanced Interactivity

- **FR6.1**: Calculate normals to improve terrain lighting and shading.
- **FR6.2**: Integrate collision detection for player interaction with the terrain.

3.3.1.7 FR7: Visual Enhancements

- **FR7.1**: Apply flat shading for a stylised appearance.
- **FR7.2**: Incorporate texture shading for added realism and detail.

3.3.1.8 FR8: Data Management and Optimization

- **FR8.1**: Develop mechanisms for efficient saving and loading of terrain data.
- **FR8.2**: Refine shading and colour application for improved visual quality.

3.3.1.9 FR9: Fixes, Optimization, and Refactoring

- **FR9.1**: Identify and fix issues to enhance performance.
- **FR9.2**: Refactor code for modularity and maintainability.

3.3.1.10 FR10: Final Touches

- **FR10.1**: Address any remaining gaps or inconsistencies in terrain generation.
- **FR10.2**: Finalize the terrain system to ensure it is robust and visually polished.

3.3.2 Non-Functional Requirements

3.3.2.1 NFR1: Performance

- **NFR1.1**: Ensure smooth gameplay with minimal lag during terrain generation.
- **NFR1.2**: Utilise multithreading and optimised LOD for high performance on various hardware.

3.3.2.2 NFR2: Scalability

- **NFR2.1**: Efficiently handle varying map sizes.
- **NFR2.2**: Accommodate infinite terrain generation without significant performance loss.

3.3.2.3 NFR3: Usability

- **NFR3.1**: Provide an intuitive interface for terrain customisation and control.
- **NFR3.2**: Offer flexible tools for quick experimentation and feature integration.

3.3.2.4 NFR4: Reliability

- **NFR4.1**: Generate consistent and reproducible terrain layouts.

3.3.2.5 NFR5: Compatibility

- **NFR5.1**: Support various platforms compatible with Unity3D.
- **NFR5.2**: Integrate seamlessly with other game components.

3.3.2.6 NFR6: Maintainability

- **NFR6.1**: Ensure the codebase is modular and well-documented for future modifications.
- **NFR6.2**: Refactor to enhance the modularity and clarity of the code.

3.3.2.7 NFR7: Aesthetics

- **NFR7.1**: Achieve high visual quality with natural-looking features and smooth transitions.
- **NFR7.2**: Use visual enhancements like texture shading for appealing terrain.

3.4 Engine Comparison

An analysis of Unity3D and Unreal Engine 5 for terrain generation revealed that both engines possess a high level of sophistication and efficiency in terrain sculpting and rendering, and each has some pros. For instance, when it comes to Unity3D, the function Mathf.PerlinNoise should be mentioned explicitly as a computationally efficient and algorithmically robust method to customise terrains with varied and detailed landscape details. Since I have experience in C#, using this function in the Unity engine would be easier to implement due to my understanding of the syntax of C#, and it would speed up the development process significantly. Unreal Engine's graphical rendering quality is impressive, as the Unreal Engine 5 renderer can create high-resolution landscapes with intricate features and complex lighting. This renderer is superior in visualising complex materials and external conditions, which ensures a high level of immersive quality that is critically important for a project where visuals are prevalent.

On the other hand, performance metrics are essential in judging the two engines, especially in the rendering function of complex terrains. Unity's performance is enabled by resource optimisation, which ensures that the rendering and other functions are stable and possible across various devices. Therefore, Unity's execution efficiency is impressive as it is less demanding on resources than Unreal Engine 5. With Unreal Engine being used in high-end computer graphics production, the system performance is often strenuous. Most of the scenes produced by Unreal Engine 5 demand high-performance computers, which means the engine is limited in the performance of low-end platforms. The balance between graphical performance and hardware requirements is a significant factor in the choice of the engine. Hence, resource utilisation should be a critical area to consider when considering engine choice, as it is a balanced scenario when the graphical quality is pitched against performance, a core aim for the toolkit being implemented.

Another angle reveals that Unity's simplistic interface, which is powered by native support for C# source code, makes for an excellent development environment that is perfect for fast iterations. Such features are partly because Unreal Engine uses a powerful development-optimized C++ programming language that might have a more extensive learning curve, slowing down the development process. Still, Unity's unified package of professional tools can take an initial idea for a prototype to a desired final product.

Ultimately, assessing optimisation capacities for various hardware configurations is essential because the created terrain must be accessible to a broad audience. As mentioned earlier, Unity3D's terrain generation system performs exceptionally well, which is consistent with the toolkit's intent to secure stable and qualitative performance across the entire set of devices, from high-end PCs to more modest CPUs and GPUs. In other words, unity seeks to offer developers the opportunity to work with the hardware they have and can afford.

Based on these considerations, the decision to specifically use Unity3D for this terrain generation project was made considering these relevant factors: Unity3D's terrain generation strengths, graphical quality, performance and optimisation metrics, prior familiarity with using C#, and hardware platform optimisation. Unity3D's compatibility with C# was also a key contributor, as my familiarity with C# was already substantial before this project began, offering a high strategic benefit in terms of workflow and time to proficiency. Furthermore, optimising the Unity engine for numerous hardware platforms is essential to ensure that the terrain will perform across multiple systems from high-performance to bare minimum entry-level.

Unity3D itself is the preferred engine for the reasons mentioned, as it meets all of the project-specific requirements, aligns with my presently available skill set, and, most importantly, provides the "best fit" for these purposes, given the end goal of creating high quality and accessible digital terrains, ultimately envisioning a generative game level.

3.5 Software Development and Design

3.5.1 Terrain Generation and Visualisation

3.5.1.1 Terrain Generation

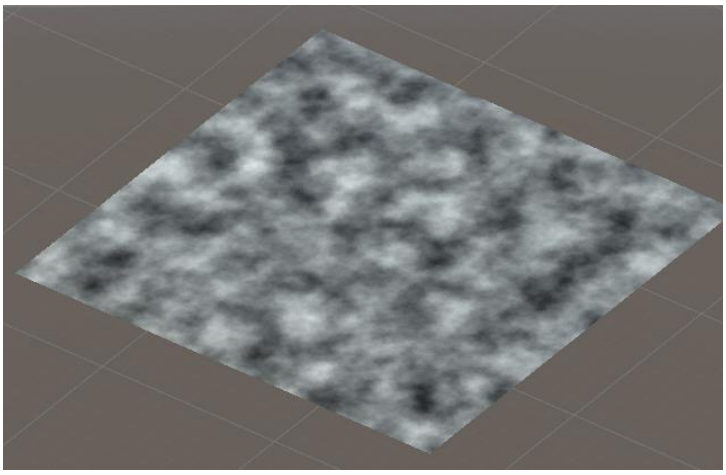

Figure 1: Generated Perlin Noise

The figure above demonstrates how perfectly the Noise class is illustrated within the toolkit, which uses an established method, Perlins noise, and PCG tools to aid with procedural terrain generation. Using the Perlin noise algorithm, the script makes it possible to generate a random simulation of noise to generate height maps to produce a terrain that can be modified to look more or less random. It is an example of an exceptional approach, as the terrain generated should resemble a natural landscape.

```
for (int y = 0; y < mapHeight; y++)
{
    for (int x = 0; x < mapWidth; x++)
    {

        amplitude = 1;
        frequency = 1;
        float noiseHeight = 0;

        for (int i = 0; i < settings.octaves; i++)
        {
            float sampleX = (x - halfWidth + octaveOffsets[i].x) / settings.scale * frequency;
            float sampleY = (y - halfHeight + octaveOffsets[i].y) / settings.scale * frequency;

            float perlinValue = Mathf.PerlinNoise(sampleX, sampleY) * 2 - 1;
            noiseHeight += perlinValue * amplitude;

            amplitude *= settings.persistance;
            frequency *= settings.lacunarity;
        }

        if (noiseHeight > maxLocalNoiseHeight)
        {
            maxLocalNoiseHeight = noiseHeight;
        }
        if (noiseHeight < minLocalNoiseHeight)
        {
            minLocalNoiseHeight = noiseHeight;
        }
        noiseMap[x, y] = noiseHeight;

        if (settings.normalizeMode == NormalizeMode.Global)
        {
            float normalizedHeight = (noiseMap[x, y] + 1) / (maxPossibleHeight / 0.9f);
            noiseMap[x, y] = Mathf.Clamp(normalizedHeight, 0, int.MaxValue);
        }
    }
}
```

Figure 2 GenerateNoiseMap class

The primary role of the class GenerateNoiseMap is to implement perlins noise. It was discovered that unity implements purlins noise through the function Mathf.PelrinNoise(x,y) will produce 2D noise maps used in procedural terrain creation. Using octaves and normalisation, the method carefully emulates real-world differences in terrain elevation. The technique adds several noise layers with amplitude and frequency, offering more depth and creating additional texture. This way, the strategy strikes a delicate balance between computational load and graphical fidelity.

The class also utilises random offsets applied to the sample points to diversify the generated terrain, which undercuts the predictable pattern of appearance. Amplitude and frequency parameters can be altered to provide flexibility over the variety and intensity of the terrain, allowing for the generation of a highly unique and realistic procedural environment. As a result, they are creating the desired terrain aesthetics and landscapes that align with the envisioned gameplay requirement.

```
System.Random prng = new System.Random(settings.seed);
```

Figure 3 PCG algorithm in Noise.cs

The script uses the PCG (Pseudo-Random Number Generator) algorithm, which allows for reproducing randomness in the terrain. This is a well-defined and innovative solution, as it allows for consistent terrain replication in procedural generation. The use of this algorithm can result in the production of terrains that can be rigorously tested and refined.

The PCG algorithm achieves this by employing a fixed random seed to produce the same outcome without fail. Consequently, the algorithm generates reproducible offsets that guarantee that the same seed creates the same terrain and allows for systematic evaluation. Additionally, the offsets make it possible to identify suitable terrains for large games by generating infinite terrains with constant similarities and differences.

3.5.1.2 Terrain Visualisation

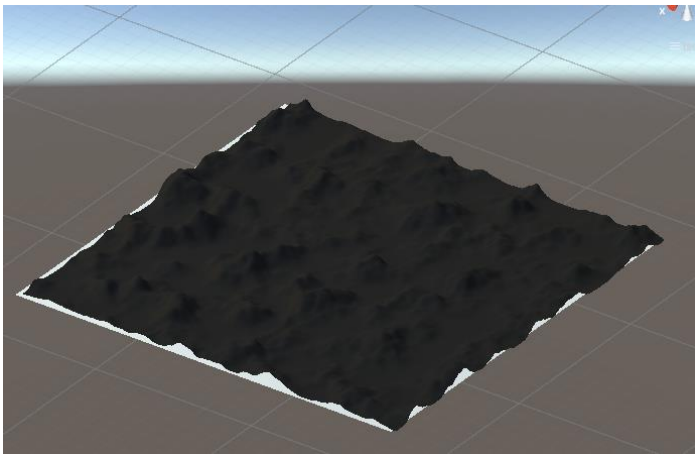

Figure 4 Generated Terrain Mesh

The HeightMapGenerator class is an example of the thoughtful use of procedural generation. They allow for the proper implementation of the terrain height map algorithm. With the help of Perlin noise, the natural elevation differences on the terrain can be captured. The critical aspect of the class is its function, GenerateHeightMap, which creates height maps using the appropriate method, as seen in the figure above. They are making a structured 2D array with height values represented on the terrain.

```
public static HeightMap GenerateHeightMap(int width, int height, HeightMapSettings settings, Vector2 sampleCentre)
{
    float[,] values = Noise.GenerateNoiseMap(width, height, settings.noiseSettings, sampleCentre);

    AnimationCurve heightCurve_threadsafe = new AnimationCurve(settings.heightCurve.keys);

    float minValue = float.MaxValue;
    float maxValue = float.MinValue;

    for (int i = 0; i < width; i++)
    {
        for (int j = 0; j < height; j++)
        {
            values[i, j] *= heightCurve_threadsafe.Evaluate(values[i, j]) * settings.heightMultiplier;

            if (values[i, j] > maxValue)
            {
                maxValue = values[i, j];
            }
            if (values[i, j] < minValue)
            {
                minValue = values[i, j];
            }
        }
    }

    return new HeightMap(values, minValue, maxValue);
}
```

Figure 5 GenerateHeightMap class

The class keeps careful track of minimum and maximum height, which allows for a precise and detailed elevation profile that fully captures each area's range of heights. Ultimately, the output is a HeightMap object that stores the points attributed with the range information, presenting a comprehensive height terrain and demonstrating a method to produce unique and realistic terrain views.

```
public struct HeightMap
{
    public readonly float[,] values;
    public readonly float minValue;
    public readonly float maxValue;

    public HeightMap(float[,] values, float minValue, float maxValue)
    {
        this.values = values;
        this.minValue = minValue;
        this.maxValue = maxValue;
    }
}
```

Figure 6 HightMap Struct

The HeightMap struct is a well-defined and implemented component which is crucial throughout the procedural terrain development. It efficiently stores the end values and limits for the height, ensuring its dynamic use with previously determined maximum and minimum levels. The structure appropriately encompasses the height map information, simplifying its exposure in the overall generation approach, which indicates a perception of data encapsulation in procedural development.

The script employs noise-based height data to generate height maps for terrains, applying an innovative approach to terrain generation. The script offers a unique way to ensure that the rendered terrain automatically adapts to the range of heights based on the height data, which creates a natural and visually appealing terrain without spending additional time tuning the terrain's heights, demonstrating a thorough understanding of procedural generation techniques, allowing the stored terrain height data to control the visual quality and realism of the procedurally generated terrains showcasing how structured data can be harnessed to achieve visually impressive results.

3.5.2 Level of Detail & Endless Terrain Generation

3.5.2.1 Level of Detail (LOD)

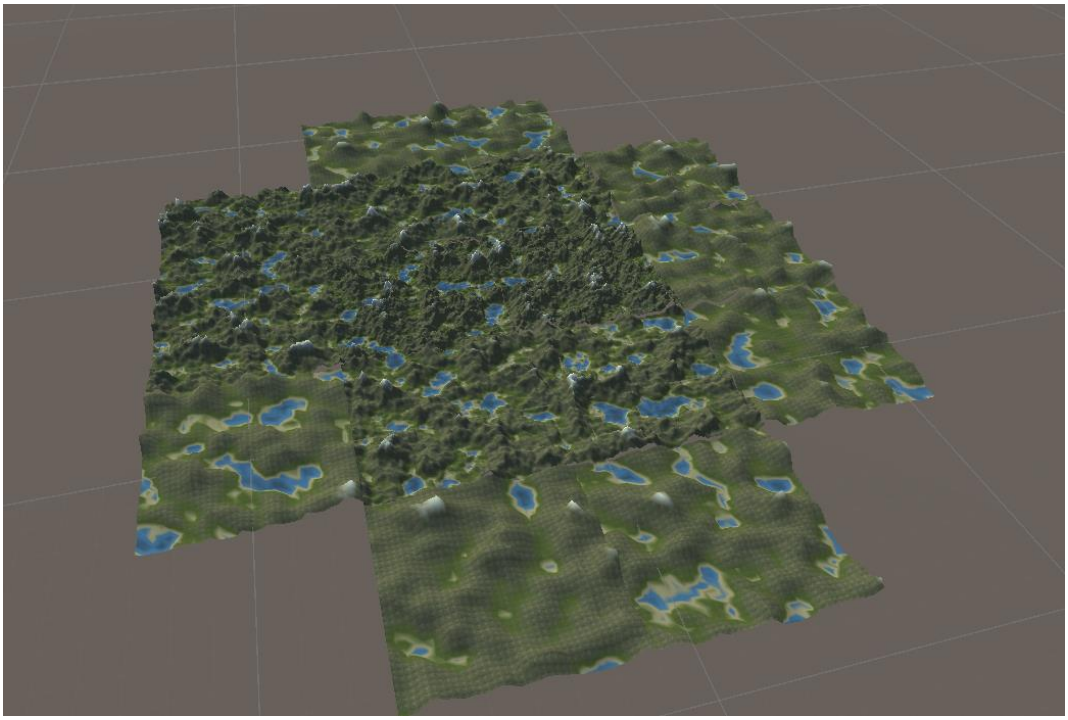

Figure 7 Level of Detail being portrayed on terrain

This figure shows that the closer the terrain is to the viewer, the more detail there is on the terrain chunks close to the player, with apparent textures, elevation, and water bodies visible. Mesh resolution is set according to the viewer's proximity, resulting in high-resolution meshes replicating detailed terrain models in the foreground and simplifying as lower-resolution meshes if the terrain is more distant.

By varying LODs in this way, the rendering process is fully optimised rationally and thoroughly, applying innovative solutions to exploit the advantages of LOD methods while being aware of their restrictions. The grid arrangement of the terrain demonstrates that mesh resolutions of separate parts range dynamically due to the proper allocation of computational resources. This approach allows for decreasing the rendering resources needed for the most distant objects, therefore permitting the system to increase additional computational resources on the objects that require high-detail rendering, leading to a real-time rendering process.

```
public void UpdateTerrainChunk()
{
    if (heightMapReceived)
    {
        float viewerDstFromNearestEdge = Mathf.Sqrt(bounds.SqrDistance(viewerPosition));

        bool wasVisible = IsVisible();
        bool visible = viewerDstFromNearestEdge <= maxViewDst;

        if (visible)
        {
            int lodIndex = 0;

            for (int i = 0; i < detailLevels.Length - 1; i++)
            {
                if (viewerDstFromNearestEdge > detailLevels[i].visibleDstThreshold)
                {
                    lodIndex = i + 1;
                }
                else
                {
                    break;
                }
            }

            if (lodIndex != previousLODIndex)
            {
                LODMesh lodMesh = lodMeshes[lodIndex];
                if (lodMesh.hasMesh)
                {
                    previousLODIndex = lodIndex;
                    meshFilter.mesh = lodMesh.mesh;
                }
                else if (!lodMesh.hasRequestedMesh)
                {
                    lodMesh.RequestMesh(heightMap, meshSettings);
                }
            }


        }
```

Figure 8 RequestMesh Method

The RequestMesh method is extracted from the LODMesh class. This method is responsible for producing the mesh, appropriately considering the current LOD of the chunk. This provides clear evidence that using the suitable techniques creates new answers for terrain rendering complications. It ensures that every chunk is rendered while considering the ideal amount of detail to produce the best visual output while optimising performance.

The design of this method demonstrates an excellent grasp of the strengths and limitations of different mesh resolutions in procedural generation. This system responds to a player's proximity, generating high-resolution meshes for relatively close land and low-resolution ones for distant terrain. This balance between visual quality and stable performance is an example of adaptive mesh generation. It allows for the seamless management of expansive terrains without undermining the

graphical fidelity, is visible to the player, and uses fewer computational resources as it generates different chunks with different mesh resolutions.

LOD Management is handled by interactively manipulating the Level of Detail for each segment of the terrain. The LODInfo array includes LOD levels and chooses the correct mesh based on the distance to the camera while considering the quality of the user's system. This demonstrates a creative solution to a problem that contributes to quality and performance.

The UpdateTerrainChunk method uses the LODInfo array to determine the most appropriate LOD mesh for each terrain chunk. The ability to choose the best LOD dynamically allows for a perfect balance between the necessary level of visual detail and performance optimisation. The tool dynamically adapts the terrain's complexity to make the traversing experience engaging and immersive.

3.5.2.2 Endless Terrain Generation

```
public void Load()
{
    ThreadedDataRequester.RequestData(() => HeightMapGenerator.GenerateHeightMap(meshSettings.numVertsPerLine,
        meshSettings.numVertsPerLine,
        heightMapSettings,
        sampleCentre),
        OnHeightMapReceived);
}



void OnHeightMapReceived(object heightMapObject)
{
    this.heightMap = (HeightMap)heightMapObject;
    heightMapReceived = true;

    UpdateTerrainChunk();
}

Vector2 viewerPosition
{
    get
    {
        return new Vector2(viewer.position.x, viewer.position.z);
    }
}


public void UpdateTerrainChunk()
{
    if (heightMapReceived)
    {
        float viewerDstFromNearestEdge = Mathf.Sqrt(bounds.SqrDistance(viewerPosition));
```

Figure 9 Endless Terrain Script

This function is responsible for loading and updating terrain chunks. It is demonstrated by using relevant methods and tools for the procedural generation of an endless terrain by implementing a fast and elaborate process of rendering neverending landscapes.

The Load method ensures that each terrain chunk is initialised with the necessary data and generates its mesh outline so that it renders correctly.

The following method, UpdateTerrainChunk, significantly expands this arrangement. This method keeps track of the player's movement and adjusts the active chunks accordingly so that only the visible parts are ready for real-time rendering. This optimises the resources spent on rendering, making it a sustainable minimax solution.

.

```
public void SetVisible(bool visible)
{
    meshObject.SetActive(visible);
}
```

Figure 10 SetVisible method

The method, SetVisible, is created specifically to control chunk visibility by using the camera-determined distance to turn a terrain chunk on and off. The SetVisible() method efficiently manages resources by dynamically updating the visibility status of each chunk depending on the camera's current position. Only displaying chunks of mesh within the range of a player's view are activated, thus minimising the rendering load and enhancing performance with minimal computing expense.

```
public class TerrainGenerator : MonoBehaviour
{

    const float viewerMoveThresholdForChunkUpdate = 25f;
    const float sqrViewerMoveThresholdForChunkUpdate = viewerMoveThresholdForChunkUpdate * viewerMoveThresholdForChunkUpdate;
```

Figure 11 TerrainGenerator Class

Tracking player movement by monitoring the player's position and updating the dataset used, ensuring dynamic terrain adjustments. The sqrviewerMoveThresholdForChunkUpdate provides the minimum distance the player must move before the terrain updates, accurately determining when a new terrain chunk should be generated, or an existing terrain chunk should be modified. This is all based on players' proximity, and it calculates how close the player character is to the edge of the rendered area. It generates new areas, giving it a never-ending experience when traversing this terrain. This ensures that only necessary terrain sections are updated, meaning the computational resources are spent only updating the necessary terrain chunks.

3.5.3 Graphical Improvement

3.5.3.1 Texture Shader

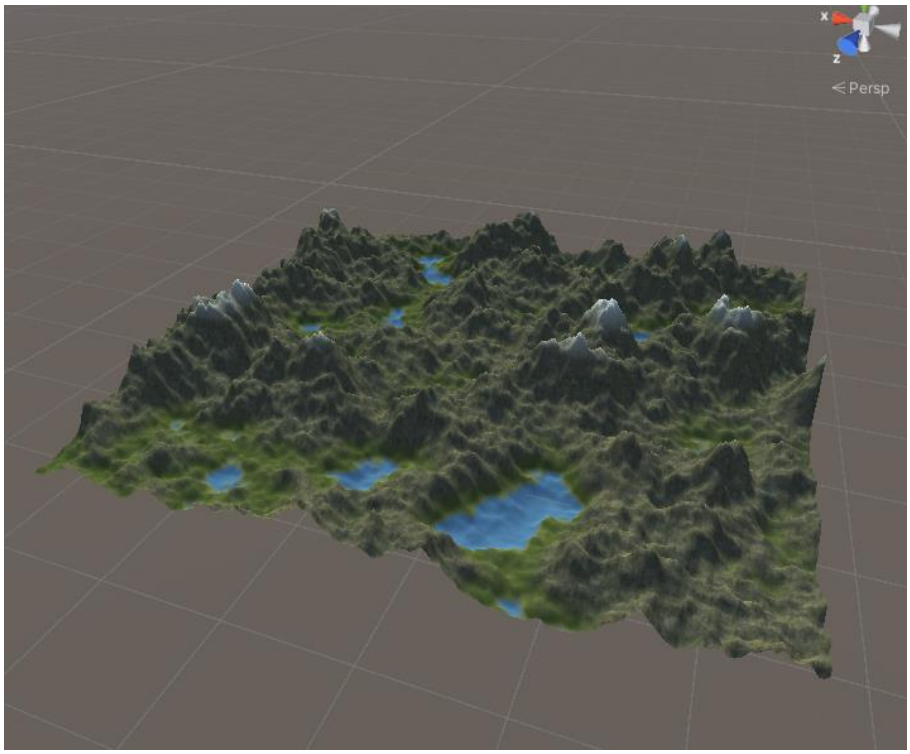

Figure 12 Standard Surface Shade on Terrain

The figure illustrates a version of the terrain during the project's lifespan, using advanced 2D texturing techniques to render textures influenced by different regions displaying varied vegetation and surface textures. These include surface shaders that use Shader Properties, Texture Arrays, and Layer Properties, thereby allowing the use of complex texturing techniques. Unlike the previous version, in which 2D vector colours painted the biomes in the heightmap-generation stage, the improved textures have significantly increased visual quality due to the proper use of the appropriate standard surface shades.

```
Shader "Custom/Terrain" {
    Properties {
        testTexture("Texture", 2D) = "white"{}
        testScale("Scale", Float) = 1
```

Figure 13 Standard Surface Shader Properties

The Shader Properties allow for using properties such as test texture, which refers to a 2D texture. A texture that maps to the scaling factor enables the shader to implement a new method of modifying its visual textures through Unity's material inspector. Using shader properties provides an advanced tool to adjust terrain texture properties accurately. The newly implemented feature changes the texture scale and many more visual parameters directly via the material inspector. This example shows a clear understanding of the Unity framework's strengths and weaknesses. Therefore, it offers solutions to tweak visual results, such as varying texture scales, by adjusting the shader properties

directly under the Unity Editor. This example shows how the shader programming may offer a flexible and controlled set of tools when using properties with a balanced approach.

```
const static int maxLayerCount = 8;
const static float epsilon = 1E-4;

int layerCount;
float3 baseColours[maxLayerCount];
float baseStartHeights[maxLayerCount];
float baseBlends[maxLayerCount];
float baseColourStrength[maxLayerCount];
float baseTextureScales[maxLayerCount];
```

Figure 14 Texture Arrays in Standard Surface Shader

Texture arrays and layer properties are crucial for managing terrain textures with precision. Texture arrays store multiple textures representing different terrain layers, while the layer properties, such as start heights and blending strengths, define how these textures blend seamlessly across the terrain. This approach ensures an autonomous blending of textures, resulting in realistic visuals.
This is shown by each layer's textures and properties being transmitted to the shader through material properties, highlighting the application of appropriate methods. Each terrain layer is accurately represented, resulting in consistent landscapes. The blend of textures, driven by these properties, reveals an understanding of terrain texture layering's strengths and limitations. This allows for smooth transitions and realistic terrain visuals, optimising graphical fidelity by applying blending strategies thoughtfully to simulate realistic terrains.

```
public void ApplyToMaterial(Material material)
{

    material.SetInt("layerCount", layers.Length);
    material.SetColorArray("baseColours", layers.Select(x => x.tint).ToArray());
    material.SetFloatArray("baseStartHeights", layers.Select(x => x.startHeight).ToArray());
    material.SetFloatArray("baseBlends", layers.Select(x => x.blendStrength).ToArray());
    material.SetFloatArray("baseColourStrength", layers.Select(x => x.tintStrength).ToArray());
    material.SetFloatArray("baseTextureScales", layers.Select(x => x.textureScale).ToArray());
    Texture2DArray texturesArray = GenerateTextureArray(layers.Select(x => x.texture).ToArray());
    material.SetTexture("baseTextures", texturesArray);

    UpdateMeshHeights(material, savedMinHeight, savedMaxHeight);
```

Figure 15 Application of shader to material

The Material Application function effectively applies appropriate methods and tools in terrain rendering, translating texture data into visual representation. Configuring properties like blending, scaling, and colour adjustments simulate the intricate visual aspects of the terrain for an immersive and realistic appearance during rendering.

```
float3 triplanar(float3 worldPos, float scale, float3 blendAxes, int textureIndex) {
    float3 scaledWorldPos = worldPos / scale;
    float3 xProjection = UNITY_SAMPLE_TEX2DARRAY(baseTextures, float3(scaledWorldPos.y, scaledWorldPos.z, textureIndex)) * blendAxes.x;
    float3 yProjection = UNITY_SAMPLE_TEX2DARRAY(baseTextures, float3(scaledWorldPos.x, scaledWorldPos.z, textureIndex)) * blendAxes.y;
    float3 zProjection = UNITY_SAMPLE_TEX2DARRAY(baseTextures, float3(scaledWorldPos.x, scaledWorldPos.y, textureIndex)) * blendAxes.z;
    return xProjection + yProjection + zProjection;
}
```

Figure 16 Triplanar mapping function

The Triplanar Mapping function is essential in applying advanced projection techniques that blend textures seamlessly across surfaces of varying orientations, ensuring natural-looking terrain. It applies projections along the x, y, and z axes, blending textures based on the surface's position and scale. This ensures correct texture application regardless of the surface's orientation, preventing stretching on steep slopes or uneven terrains.
By implementing this method, the function effectively maintains visual consistency across diverse topographies. The strategic use of tri-planar projections shows a thorough understanding of traditional texturing techniques, emphasising the need for innovative methods in procedural terrain generation. The result is visually stunning terrain, where textures appear accurately and realistically across all angles, significantly enhancing the environment's overall aesthetic appeal.

3.5.3.2 URP Shader Graph

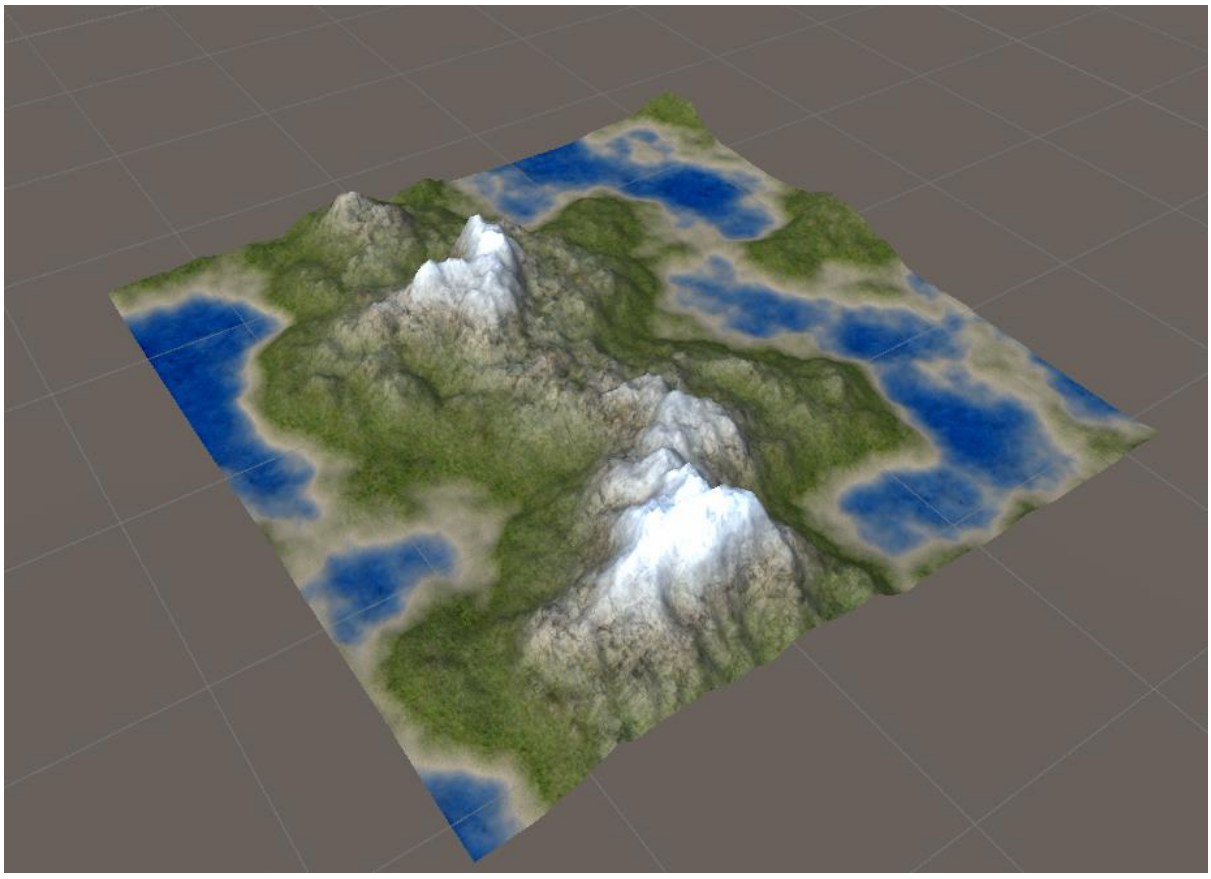

Figure 17 URP Shader used for Terrain Texturing

The noticeable improvement in graphics showcases the effective use of tools such as Unity's Universal Render Pipeline (URP) for generating realistic terrain. Coupled with shader graphs, the URP creates an innovative framework for texturing landscapes. This approach allows for detailed textures like water, sand, grass, rock, and snow to blend seamlessly across the terrain. The shader graph efficiently handles these transitions, dynamically adjusting to the terrain's features for realistic blending. URP demonstrates a deep understanding of terrain rendering, enabling efficient, high-quality texturing that meets various performance needs. The customisable nature of this approach

gives the flexibility to refine the terrain's appearance iteratively, aligning it with aesthetic goals while staying within performance limits.

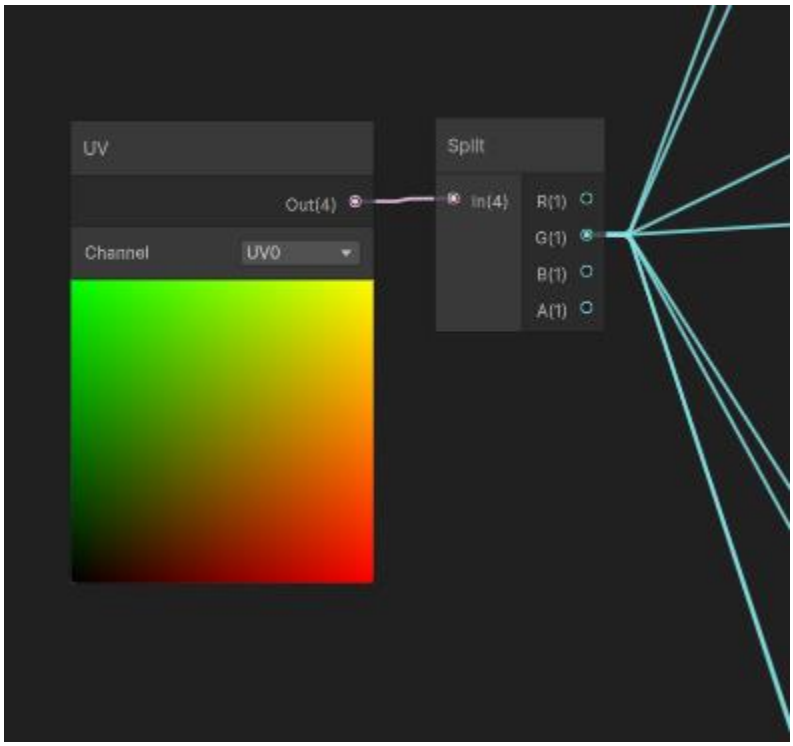

Figure 18 UV Node in Shader Graph

The UV Node enhances the visual quality of procedurally generated terrain by leveraging its adaptive texturing to blend textures naturally, creating realistic transitions based on input values that reflect desired visual conditions. Its implementation of the UV Node showcases the effective use of tools and methods precisely defined within the system. The UV Node's ability to provide adaptive and realistic transitions enhances the procedural generation process, demonstrating the application of advanced tools and methods to achieve effective results.

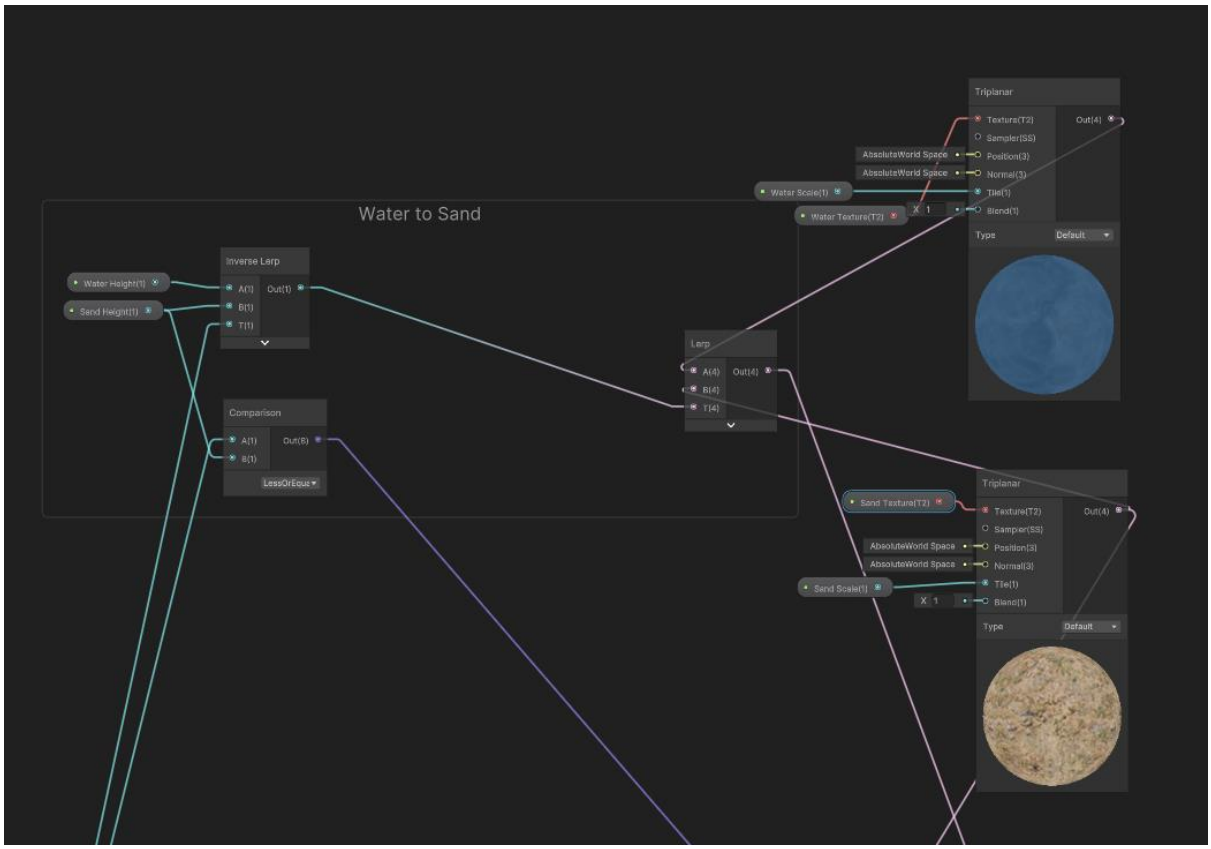

Figure 19 Transition Nodes in Shader graph

The Transition Nodes implement the proper methods and tools to manage the transition between different pairs of textures, such as from water to sand, sand to grass, grass to rock, and rock to snow. As a feature developed to mimic the behaviour of natural landscapes, this feature demonstrates an understanding of terrain rendering innovations.

Integrating the transition nodes helps the system ensure that the fade occurs gradually, which mirrors the gradual shifts in natural landscapes. Another essential part of the Transition Nodes is adaptive texturing, which helps the system adjust transitions according to terrain elevation and slope, similar to natural terrain. The system can create natural blends by accommodating different terrains. The feature is also dynamic, allowing for customisation of each transition. This aspect ensures that the terrain behaves independently, thus noting the strength of smooth transitions and the limitation of traditional texturing.

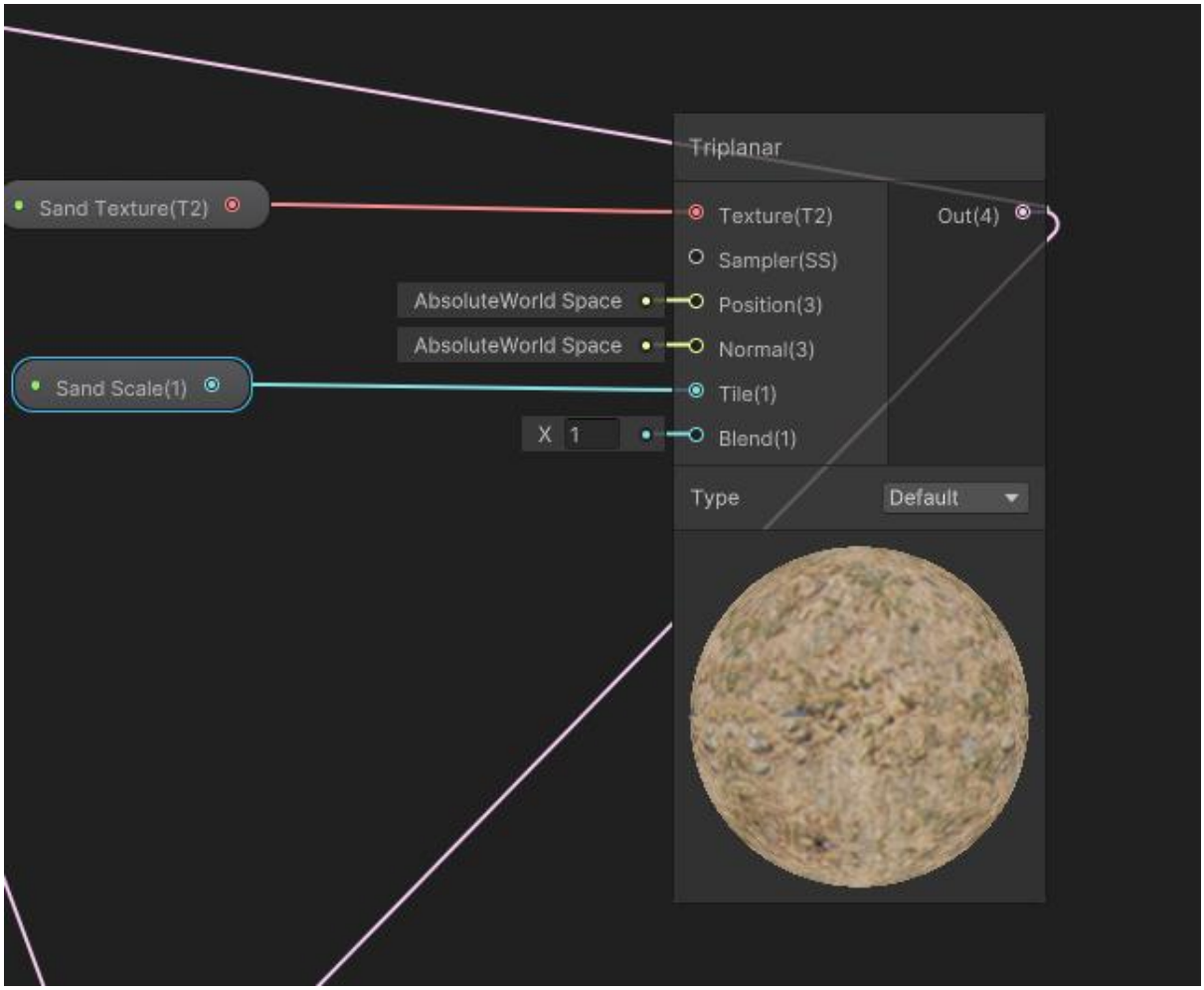

Figure 20 Gradient and Comparison Nodes

The Gradient and Comparison Nodes are pivotal tools that employ advanced, adaptive methods for managing texture transitions based on specific terrain conditions. These nodes are innovatively crafted to produce realistic transitions by skillfully blending textures across various terrain features. They use adaptive texturing to dynamically adjust texture blending based on the terrain's height, slope, and other attributes. The customisable design of these nodes reflects a deep understanding of the strengths and limitations of the texturing process, enabling users to modify blending conditions effortlessly. This adaptability makes them essential for creating visually diverse landscapes, effectively tackling the challenges of realism in procedural generation.

Texture Nodes are equally well-crafted, holding the textures used to render the terrain. Their significant contribution to the realism of terrain transitions by providing textures that seamlessly blend across different terrain types demonstrates the application of appropriate methods. The strategic combination of these nodes results in terrains with smooth and adaptive transitions, maintaining a visually realistic appearance. The integration of these shader tools signifies a comprehensive increase in the graphical quality of procedural terrain generation, yielding a well-defined, innovative solution that prioritises visual quality and realism.

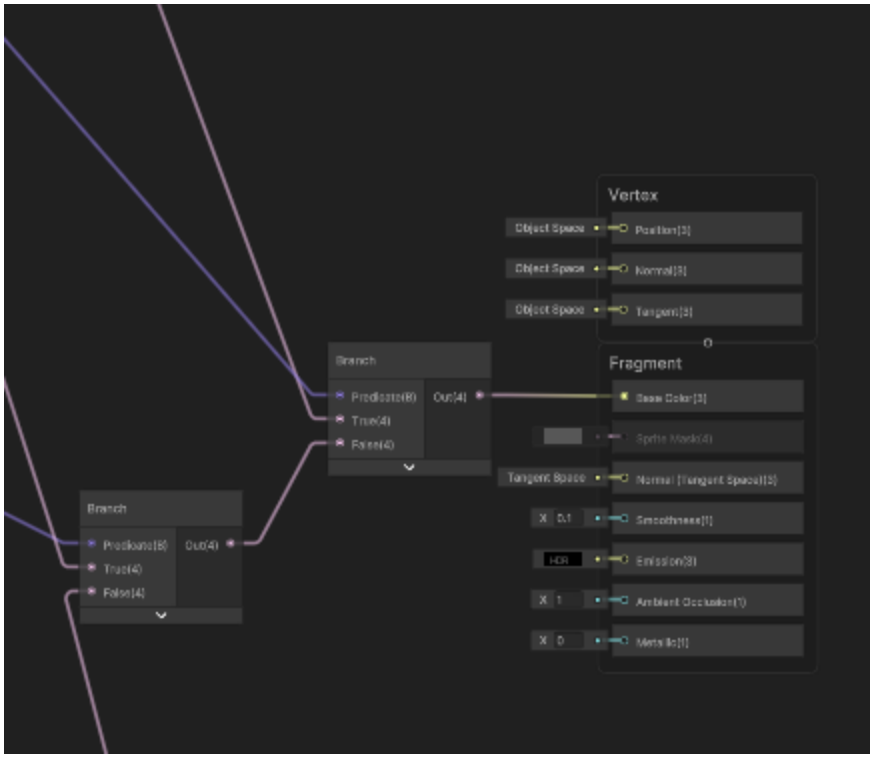

Figure 21 Vertex and Fragment Nodes

The Vertex and Fragment Nodes play a crucial role by demonstrating suitable methods and tools to achieve innovative solutions with a clear understanding of their strengths and limitations. As an essential part of the rendering process, these nodes ensure that the vertex and fragment shaders accurately represent the terrain, handling transitions to ensure that textures and effects are appropriately applied. The adaptive texturing feature showcases their innovative approach by dynamically adjusting texture output based on complex calculations and blending. These nodes provide significant flexibility, enabling the user to fine-tune the shader's output. This flexibility highlights an understanding of shader programming, teaching how to create and facilitate accurate and visually striking terrain representation across various environments.

4 Testing & Results

4.1 System Analysis

4.1.1 Environment testing

The justification for the choice of Windows and macOS systems to test the procedural terrain generation process with Unity was well supported by the fact that more than available Linux devices were needed for the experiment. This made it possible to assess the results of the work as objectively and comprehensively as possible and take into account its actual effectiveness in the context of the typical equipment and configurations of the target audience. Given that the Unity developers focused on the benefits of the proposed platforms, the system was fully optimised for the available configurations, and therefore, in this work, it would also be possible to assess the capabilities and limitations of the system, which made the offered platform as realistic as possible for the further performance of optimal changes in the work. The existence of development tools and hardware for Windows and macOS also contributed to the simplified performance of the mathematics of the configuration of the CPU, GPU, and memory, respectively. Such substantive support made it possible to gather more compelling evidence of the effectiveness of terrain generation and the disadvantages that the selected system may have when working in the available conditions.

	Cluster Computer	**Lenovo Legion 5**	**MacBook Pro (2015)**	**MacBook Pro (2021)**
Processor	Intel® Core™ i5-6600 CPU @ 3.30GHz	AMD Ryzen 7 4800H	intel core i5 @ 2.7GHz	Apple M1 Pro
Memory	16GB	16GB	8GB	16GB
GPU	NVIDIA GeForce GTX 1050	NVIDIA GeForce RTX 2060	Intel Iris Graphics 6100	Integrated CPU graphics
Operating System	Windows 10	Windows 10	macOS Monterey	macOS Sonoma

Figure 22 Table of Devices Used

The method to calculate CPU, GPU, and memory usage relies on statistics gathered through black-box testing of performance to obtain figures for system usage.

4.1.2 CPU Usage

To determine the CPU usage, the primary and render thread times were comprehensively analysed, after which the CPU time was found for one frame in total. Based on these data, the frame interval was calculated as the reciprocal of the frame rate and, therefore, the available time per frame. This mathematical expression allowed us to visualise the data in a detailed manner, which could later be expressed in the following formula to determine the CPU usage in percentage.

The formula:

$$\text{CPU Usage} = \frac{\text{Total CPU Time per Frame}}{\text{Frame Interval}} \times 100$$

This evaluation revealed that optimal CPU utilisation for running multiple threads is indicated by CPU usage exceeding 100%. It means that the CPU uses all the available processing power effectively to perform tasks across multiple threads. However, this evaluation has its drawbacks as it only demonstrates how well the CPU is utilised. Other components and bottlenecks, such as GPU usage and memory, need to be taken into consideration, which could also affect the overall performance of the system. Nevertheless, this method can still be utilised as a valuable indicator for evaluating CPU performance in complex systems.

4.1.3 GPU Usage

The amount of GPU used was estimated using FPS achieved during black-box testing. When we reach a higher FPS, we can infer that the GPU is well utilised since a low FPS indicates the high usage of the GPU as a diverse computational workload is imposed. However, this method needs to be more accurate, given that FPS is influenced by other factors such as CPU usage, memory bandwidth, the nature of rendering tasks used, and more. Despite not being accurate, FPS gives a rough estimate of the percentage of the GPU capacity and how much is being utilised to achieve a sustained FPS. This estimate provides an overview of the system performance, whereas defining tools are required for an exact analysis.

4.1.4 Memory Usage

The calculation of memory usage involves evaluating the screen memory usage during gameplay and comparing it to the computer's total memory.

$$\text{Memory Usage} = \left(\frac{\text{Screen Memory Usage}}{\text{Total System Memory}}\right) \times 100$$

Such an analysis provides an in-depth overview of the outcomes, representing the basis for a valid and profound assessment. It delineates the extent to which optimisation changes made during development impact memory management. The low percentage of memory usage indicates that resources are used effectively, suggesting that optimisations were successfully implemented to enhance memory efficiency. Additionally, this calculation underscores the limitations of the evaluation method, which focuses primarily on measuring optimisation efficiency rather than delivering a comprehensive assessment of overall performance quality and extent.

4.2 Black-Box Testing of Performance

Testing was performed to examine how various systems cope with the new version of the procedurally generated terrain software. The results are well-described as a result of the comprehensive assessment, providing clues on how well computer systems can cope with the complex graphical tasks within the software. The terrain was launched on computers following the restart, and the software was allowed to function for five minutes to make sure it ran steadily. During this period, the Endless procedurally generated terrain was assessed to collect CPU, GPU, and memory usage rates for all the systems involved.

All tests were run at a standard 1080p resolution on all systems to provide a reliable ground for comparison. For the two lowest-performing computers, the two high-end PCs that could render at 4K were used to achieve a more accurate determination of the computers' ability to handle graphically

challenging assignments. This ensured that the tests provided an adequate overview of each computer's performance under different levels of graphics intensity. The results also presented essential evidence that could be utilised in the evaluation and optimisation of the system's performance by only outlining the performance features and drawbacks that could be addressed

4.3 Results

4.3.1 Environmental Testing

The latest Prototype version v1.0 did not have any issues, working on all 4 systems with ease. However, it was observed that the time taken to resolve packages had a long duration, which is mostly due to the time needed to import literally all libraries and install all packages used.

4.3.2 System Testing

4.3.2.1 University of Newcastle, Urban Science Building, Cluster Computer

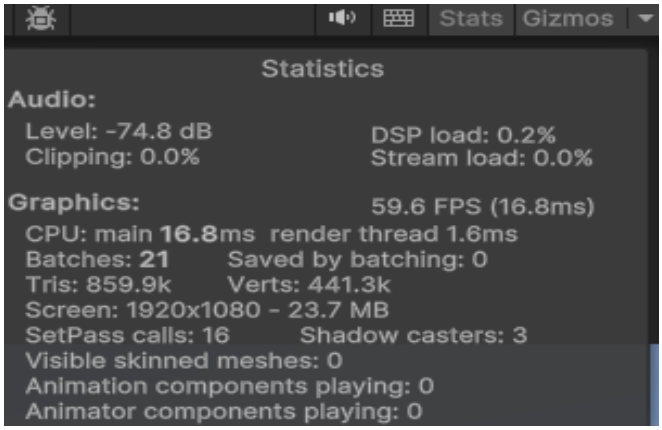

Figure 23 Benchmarking of Cluster Computer

CPU Usage Calculation

$$\textbf{Main Thread Time: } 16.8\ \textbf{ms}$$

$$\textbf{Render Thread Time: } 1.6\ \textbf{ms}$$

$$\textbf{Frame Interval: } \frac{1000\ \textbf{ms}}{59.6\ \textbf{FPS}} \approx 16.8\ \textbf{ms}$$

$$\textbf{Total CPU Time per Frame} = \textbf{Main Thread Time} + \textbf{Render Thread Time}$$

$$= 16.8\ \textbf{ms} + 1.6\ \textbf{ms} = 18.4\ \textbf{ms}$$

$$\textbf{CPU Usage} = \frac{\textbf{Total CPU Time per Frame}}{\textbf{Frame Interval}} \times 100$$

$$\approx \frac{18.4\textbf{ms}}{16.8\textbf{ms}} \times 100 \approx 109.52\%$$

GPU Usage Estimation:

Given that the GeForce GTX 1050 is a mid-range GPU and the frame rate is near the target of 60 FPS, the GPU is estimated to be working at moderate to high load.

Estimation Formula:

This is an estimation rather than a calculation: GPU Usage=Approximately 65%

Memory Usage Calculation: $\text{Memory Usage} = \left(\frac{\text{Screen Memory Usage}}{\text{Total System Memory}}\right) \times 100$

$$= \left(\frac{23.7\ \text{MB}}{8192\ \text{MB}}\right) \times 100 \approx 0.29\%$$

These calculations indicate that the CPU is likely fully utilised with multiple threads, the GPU is nearing its performance limits, and the system has plenty of available memory capacity.

The CPU usage analysis reveals that the terrain generation system operates efficiently, utilising around 109.52% of the CPU resources per frame, which includes the main thread and render thread times. This suggests a heavy but manageable load on the CPU.

The GPU, a mid-range GTX 1050, is estimated to be functioning at moderate to high load levels, nearly reaching its performance limits while maintaining a steady frame rate close to the target of 60 FPS.

Memory usage remains low at approximately 0.29% of the total system memory, indicating that the system's RAM has ample capacity to handle additional tasks. This comprehensive evaluation illustrates the software's ability to utilise system resources effectively while acknowledging the importance of ethical considerations in its analysis.

4.3.2.2 Lenovo Legion 5 (1080p)

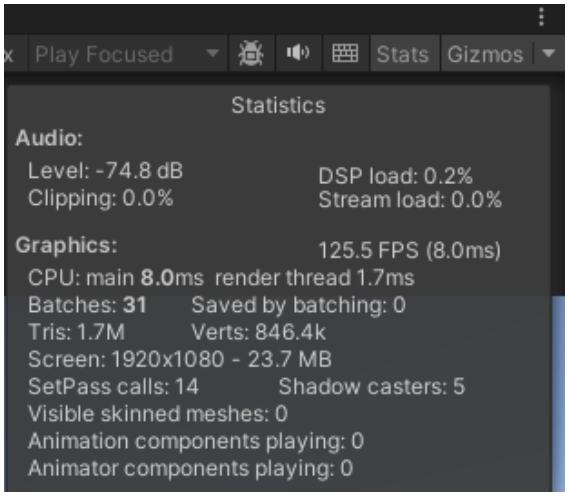

Figure 24 Benchmarking of Lenovo Leigon 5 in 1080p

CPU Usage Calculation

Main Thread Time: 8.0 ms

Render Thread Time: 1.7 ms

Frame interval: $\frac{1000\text{ms}}{125.5\text{FPS}} = 7.97$

$$\textbf{Total CPU Time per Frame} = \textbf{Main Thread Time} + \textbf{Render Thread Time}$$

$$= 8.0\text{ ms} + 1.7\text{ ms} = 9.7\text{ ms}$$

$$\textbf{CPU Usage} = \frac{\textbf{Total CPU Time per Frame}}{\textbf{Frame Interval}} \times \textbf{100} = \frac{9.7\text{ ms}}{7.97\text{ ms}} \times 100 \approx 121.7\%$$

A CPU usage of over 100% indicates that your CPU handles multiple threads efficiently, showing that it can keep up with more than one thread within the frame interval.

GPU Usage Estimation

Estimation Approach: Given the high frame rate and the capabilities of the NVIDIA GeForce RTX 2060, along with the workload implied by the screen resolution and FPS:

GPU Usage: Assuming a higher frame rate indicates less stress under current conditions, estimate the GPU usage based on typical gaming performance benchmarks for similar setups.

Estimated GPU Usage: 35% capacity, considering the GPU is performing well without reaching its limits but maintaining a high FPS.

Memory Usage Calculation

Total System Memory: 16GB

Screen Memory Usage: 23.7 MB

$$\text{Memory Usage} = \frac{\text{ScreenMemoryUsage}}{\text{TotalSystemMemory}} \times 100$$

$$= \frac{23.7\text{MB}}{16{,}384\text{MB}} \times 100 \approx 0.14\%$$

Memory Usage: At 0.58%, reflecting minimal use of total system memory for the specific operations captured.

The analysis of the Lenovo Legion 5 at 1080p resolution highlights the system's performance in terrain generation. The CPU's main thread and render thread times total 9.7 ms per frame, with usage of approximately 121.7%, indicating that the system handles multiple threads efficiently, even exceeding 100% usage. This demonstrates the CPU's ability to manage more than one thread within the frame interval. The GPU, an NVIDIA GeForce RTX 2060, is estimated to be operating at 35% capacity, performing well without reaching its performance limits and maintaining a high frame rate. The GPU usage estimation is based on a high frame rate, which implies less stress under current conditions.

In terms of memory usage, the system shows minimal consumption, with only 0.14% of the total system memory being used. This translates to about 23.7 MB of the available 16 GB, reflecting that the system's RAM has plenty of headroom for additional operations. This thorough evaluation underscores the system's efficient resource management, which is capable of handling demanding graphical tasks while maintaining a robust performance level.

4.3.2.3 Lenovo Legion 5 (4K)

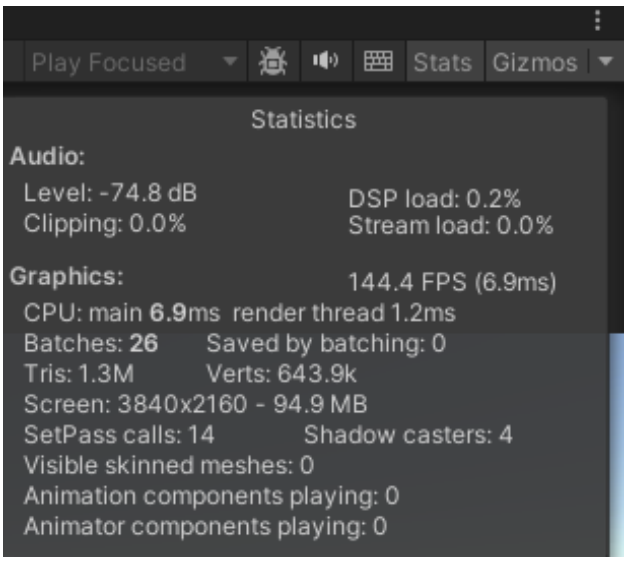

Figure 25 Benchmarking of Lenovo Legion 5 in 4K

CPU Usage Calculation

Main Thread Time: 6.9 ms

Render Thread Time: 1.2 ms

$$\text{Frame Interval} = \frac{1000\ \text{ms}}{144.4\text{FPS}} = 6.9\text{ms}$$

$$\text{CPU Time per Frame} = \text{Main Thread Time} + \text{Render Thread Time}$$

$$= 6.0\ \text{ms} + 1.2\ \text{ms} = 7.2\ \text{ms}$$

$$\text{CPU Usage} = \frac{\text{Total CPU Time per Frame}}{\text{Frame Interval}}\text{X}100$$

$$= \frac{7{,}2\ \text{ms}}{6.9\ \text{ms}}\text{X}100 \approx 104.3\%$$

At approximately 104.3%, the CPU exceeds 100% capacity, likely handling multiple threads concurrently and managing its workload efficiently.

GPU Usage Estimation

Given the efficiency of the NVIDIA GeForce RTX 2060 and its ability to handle gaming and other graphics-intensive applications efficiently, combined with the high FPS achieved, we estimate the GPU usage:

GPU Usage: Assuming that a lower FPS would mean a higher GPU load, maintaining a high FPS of 144.4 likely suggests that the GPU is not being maximally taxed. Estimating this as a moderate load, the GPU is operating at about 50% capacity.

Memory Usage Calculation

Screen Memory Usage: 23.7 MB

Total System Memory $=$ 16GB

$$\text{Memory Usage} = \frac{\text{Memory Usage}}{\text{Total System Memory}} \times 100$$

$$= \frac{94.9\ \text{MB}}{16{,}384\ \text{MB}} \times 100 \approx 0.59$$

At 0.14%, indicating very minimal use of the available memory.

The Lenovo Legion 5, running at 4K resolution, demonstrates significant CPU and GPU utilisation for terrain generation. The combined CPU time per frame, which includes the main thread and render thread times, totals 7.2 ms, indicating approximately 104.3% CPU usage. This high percentage reflects the CPU's ability to handle multiple threads concurrently, managing its workload efficiently despite exceeding 100% capacity.

The GPU, an NVIDIA GeForce RTX 2060, is estimated to be operating at around 50% capacity based on the achieved high frame rate of 144.4 FPS. This implies that the GPU is effectively handling the graphical workload without being overburdened, maintaining a balance between high performance and resource usage.

Memory usage is efficiently managed, with only 0.59% of the available 16 GB system memory being utilised, corresponding to around 94.9 MB. This minimal memory usage indicates that the system has ample RAM capacity to accommodate additional tasks, reinforcing the overall efficiency and robustness of the system in managing demanding graphical workloads at high resolutions.

4.3.2.4 MacBook Pro 2015

Figure 26 Benchmarking of MacBook 2015

CPU Usage Calculation

Main Thread Time: 16.7 ms

Render Thread Time: 2.2ms

Frame Rate: 59.9 FPS

$$\text{frame interval} = \frac{1000\ \text{ms}}{59.9\ \text{FPS}} \approx 16.7\text{ms}$$

$$\text{CPU Time per Frame} = \text{Main Thread Time} + \text{Render Thread Time}$$

$$= 16.7\ \text{ms} + 2.2\ \text{ms} = 18.9\ \text{ms}$$

$$\text{CPU Usage} = \frac{\text{CPU Time per Frame}}{\text{Frame Interval}} \times 100$$

$$= \frac{18.9\ \text{ms}}{16.7\ \text{ms}} \times 100 \approx \mathbf{113.2}\%$$

GPU Usage Estimation

Given:

FPS: 59.9

Approximately 50-60%, based on similar configurations and typical usage levels.

Memory Usage Calculation

Given:

Screen memory usage: 33.8 MB

8GB of total system memory is typical for models from that period.

$$\text{Memory Usage} = \frac{\text{Screen Memory Usage}}{\text{Total System Memory}} \times 100$$

$$= \frac{33.8\ \text{MB}}{8192\ \text{MB}} \times 100 \approx \mathbf{0.412}\%$$

The MacBook Pro 2015 provides exciting insights into terrain generation performance. The CPU time per frame, calculated by combining the central thread time and render thread time, is 18.9 ms, leading to a CPU usage of approximately 113.2%. This figure indicates that the CPU is handling more than one thread concurrently, pushing it beyond its typical capacity.

In terms of GPU performance, with a frame rate of 59.9 FPS, the GPU usage is estimated to be between 50% and 60%, which aligns with similar configurations and standard usage levels. This suggests that the GPU is managing the graphical workload effectively without being overly taxed.

Memory usage is relatively low, with about 33.8 MB used from the available 8 GB of system memory. This translates to around 0.41% of the total RAM, indicating that the system has sufficient memory to handle additional tasks. Overall, these results reflect the MacBook Pro's ability to manage terrain generation workloads efficiently, despite its age and hardware limitations.

4.3.2.5 MacBook Pro 2021 (1080p)

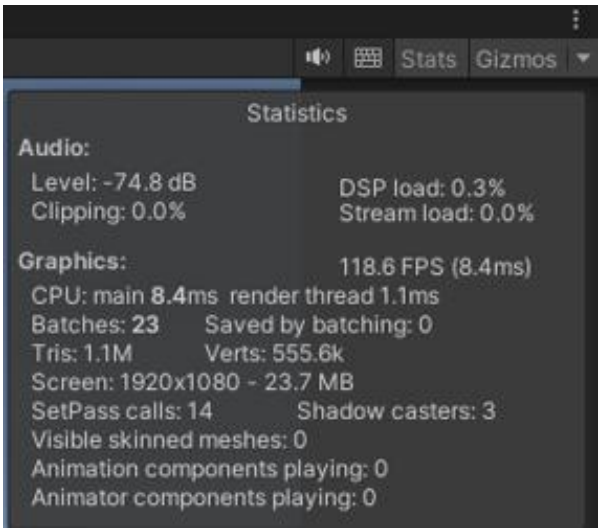

Figure 27 Benchmarking of MacBook 2021 in 1080p

CPU Usage Calculation

Main Thread Time: 8.4 ms

Render Thread Time: 1.1ms

Frame Rate: 118.6 FPS

$$\text{Frame Interval} = \frac{1000\ \text{ms}}{118.6\ \text{FPS}} \approx 8.4\text{ms}$$

$$\text{CPU Time per Frame} = \text{Main Thread Time} + \text{Render Thread Time}$$

$$= 8.4\ \text{ms} + 1.1\ \text{ms} = 9.5\ \text{ms}$$

$$\text{CPU Usage} = \frac{\text{CPU Time per Frame}}{\text{Frame Interval}} \times 100$$

$$= \frac{9.5\ \text{ms}}{8.4\ \text{ms}} \times 100 \approx \mathbf{113.1}\%$$

GPU Usage Estimation

Given:

FPS: 118.6

Approximately 75%, based on similar configurations and typical usage levels.

Memory Usage Calculation

Given:

ScreenMemoryUsage: 23.7 MB

$\text{TotalSystemMemory} = 16\ \text{GB}.$

$$\text{Memory Usage} = \frac{\text{Screen Memory Usage}}{\text{Total System Memory}} \times 100$$

$$= \frac{23.7\ \text{MB}}{16384\ \text{MB}} \times 100 \approx \mathbf{0.144}\%$$

The MacBook Pro 2021, operating at 1080p, demonstrates impressive performance metrics for terrain generation. The CPU's combined main thread and render thread times result in a total of 9.5 ms per frame, translating to approximately 113.1% CPU usage. This indicates efficient multitasking, with the CPU managing multiple threads simultaneously and pushing beyond its standard capacity.

In terms of GPU usage, the achieved frame rate of 118.6 FPS suggests an estimated GPU utilisation of around 75%, which aligns with similar system configurations and usage levels. This reflects that the GPU is handling the graphical workload effectively while maintaining a high frame rate.

Memory usage remains minimal, with about 23.7 MB used out of the 16 GB available, which amounts to approximately 0.14% of the total system memory. This low memory usage highlights the system's efficient handling of resources, allowing ample headroom for additional tasks. Overall, the MacBook Pro 2021 demonstrates robust performance in managing complex graphical workloads at high frame rates, emphasising its efficiency in terrain generation.

4.3.2.6 MacBook Pro 2021 (4k)

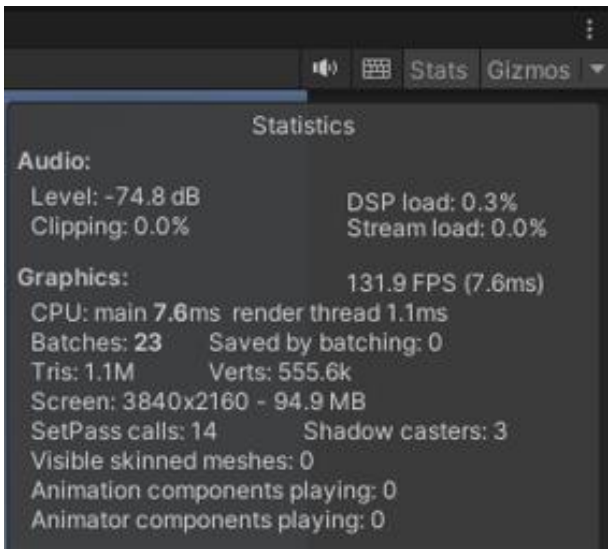

Figure 28 Benchmarking of MacBook 2021 in 4K

CPU Usage Calculation

Main Thread Time: 7.6 ms

Render Thread Time: 1.1ms

Frame Rate: 131.9 FPS

$$\text{Frame Interval} = \frac{1000\ \text{ms}}{131.9\ \text{FPS}} \approx 7.6\text{ms}$$

$$\text{CPU Time per Frame} = \text{Main Thread Time} + \text{Render Thread Time}$$

$$= 7.6\text{s} + 1.1\ \text{ms} = 8.7\ \text{ms}$$

$$\text{CPU Usage} = \frac{\text{CPU Time per Frame}}{\text{Frame Interval}} \times 100$$

$$= \frac{8.7\ \text{ms}}{7.6\ \text{ms}} \times 100 \approx \mathbf{114.5}\%$$

GPU Usage Estimation

Given:

FPS: 131.9

Approximately 20%, based on similar configurations and typical usage levels.

Memory Usage Calculation

Given:

$\text{ScreenMemoryUsage: } 94.9 \text{ MB}$

$\text{TotalSystemMemory} = 16 \text{ GB}.$

$$\text{Memory Usage} = \frac{\text{Screen Memory Usage}}{\text{Total System Memory}} \times 100$$

$$= \frac{94.9 \text{ MB}}{16384 \text{ MB}} \times 100 \approx \mathbf{0.58}\%$$

The MacBook Pro 2021, running at 4K resolution, demonstrates robust performance in terrain generation. The total CPU time per frame, which combines the main thread and render thread times, is 8.7 ms, resulting in approximately 114.5% CPU usage. This high percentage indicates the CPU's efficient management of multiple threads, pushing beyond its typical capacity.

In terms of GPU performance, the achieved frame rate of 131.9 FPS suggests an estimated GPU utilisation of about 20%, indicating that the GPU is managing the graphical workload effectively and has significant headroom for more demanding tasks.

Memory usage is relatively modest, with around 94.9 MB of the available 16 GB system memory being utilised. This represents about 0.58% of the total RAM, showing that the system efficiently handles memory usage, providing plenty of capacity for additional operations. Overall, the MacBook Pro 2021 effectively manages complex graphical workloads at high frame rates and resolutions, showcasing its strong performance in terrain generation tasks.

4.4 Conclusion of Benchmarking

All tests in the first version ran entirely without issues. Every device was enabled to run more than 60 FPS at 1080p resolution. This means that the tool is optimised perfectly, and each configuration runs properly without any issues with performance and frame-rate consistency. The GPU and CPU power in each of the tested systems varied significantly. However, each of the completed series provided high frame rates. It demonstrates that the tool has been meticulously designed to handle procedural terrain generation independently of other processes, all while maintaining good graphics and system behaviour. The only statement that can be expressed is that the results are posiitve, as benchmarks recorded states that all usage and processing speeds where not at full capacity and worked with the limitations and benefits of each device.

5 Evaluation

Name	USB Computer	Lenovo 1080p	Lenovo 4k	MacBook Pro (2015) 1080p	MacBook Pro (2021) 1080p	MacBook Pro (2021) 4k
CPU Usage	109.52%	121.70%	104.30%	113.20%	113.10%	114.50%
GPU Usage	65%	35%	75%	55%	75%	20%
Memory Usage	0.29%	0.14%	0.59%	0.46%	0.14%	0.58%

Figure 29 Results of Computational Resources Used in Testing

5.1 Analysis of CPU and GPU Usage

A detailed analysis of CPU and GPU use enables us to see how procedural terrain generation is employed across different devices. Similarly, comparing the devices in terms of screen resolution helps to understand and demonstrate the capabilities and boundaries of their computational capacities.

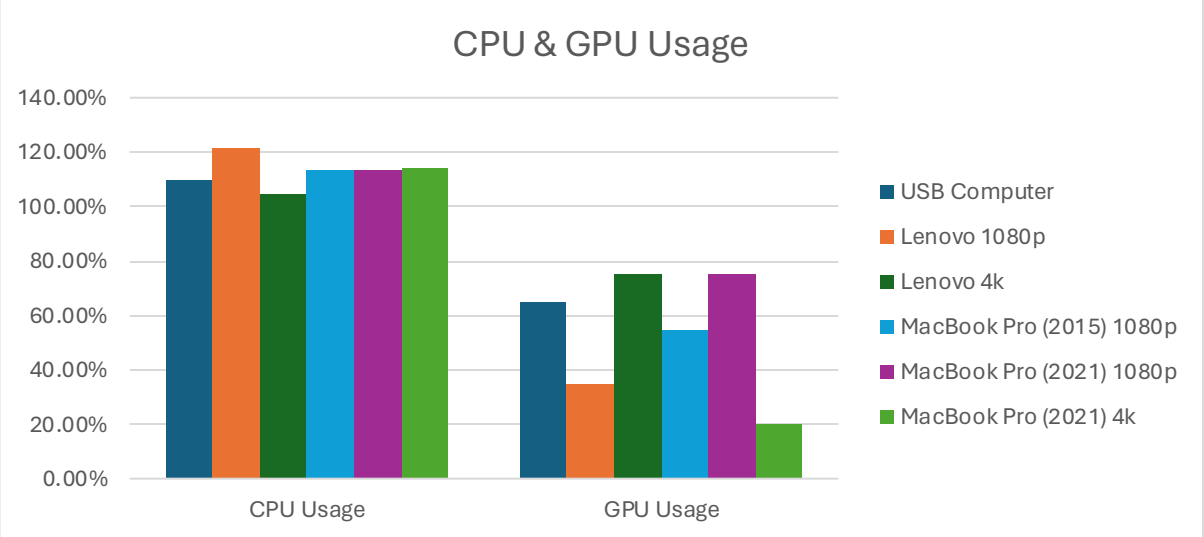

Figure 30 Bar Chart Comparing CPU and GPU usage of Devices

Among the devices running at 1080p resolution, The Lenovo Legion 5 and MacBook Pro 2021 show high CPU usage, 121.7% and 113.1%, respectively. The high percentage remains indicative of the effective use of multi-threading since it means that the Central Processing Unit is used fully. Indeed, the terrain generation tool is effective in utilising the total capacity of all the available cores. The MacBook Pro 2015 also shows 113.2% CPU usage, which makes it comparable to the MacBook Pro 2021. Hence, it is evident that even older hardware can handle the intensive workload.

Regarding GPU usage, the MacBook Pro 2021 and Lenovo Legion 5 each reached 75% of the GPU performance at this resolution, demonstrating their ability to work effectively with demanding graphics loads, even while providing enough resources to add more features. The 2015 MacBook Pro's performance, using this resource, was much less: with the exact resolution, the system reached only 55% of the GPU capacity when it used fewer resource-intensive graphics. Thus, assuming the load patterns, then at this resolution, the Lenovo Legion 5 used more CPU, but in general, being the most potent computer among the compared, provided equal ratio use on all resources, and here the Legion 5 is the best as more room is left for productivity intensive tasks. The least optimised was the 2015 MacBook Pro, as it couldn't use all of its resources effectively, even at different usage ratios.

Evaluating the devices running at 4K resolution, the CPU of the Lenovo Legion 5 worked at 104.3%, and the MacBook Pro 2021 achieved 114.5%. Both percentages are high and prove the effective use of multi-threading, suggesting that both devices could handle complex workloads despite the high resolution. However, further optimisation might be needed to manage the increased demands of 4K rendering efficiently.

In terms of GPU usage, the Lenovo Legion 5 utilised 75% of its GPU, indicating effective management of high-resolution graphical workloads. However, the MacBook Pro 2021 exhibited only 20% GPU usage at 4K, which may suggest that the system needed to be fully optimised for the benchmark scenario or that it offloaded more of the workload to the CPU.

The Lenovo Legion 5 stands out as the best-performing 4K device, effectively balancing CPU and GPU usage to handle high-resolution terrain generation. In contrast, the MacBook Pro 2021 could have been more efficient due to its low GPU usage, potentially leading to suboptimal performance despite its high CPU capacity.

5.2 Analysis of Memory Usage

Considering that the efficiency with which a computational application runs is essential, memory usage is another critical feature to consider. This feature is especially crucial in graphically intensive scenarios such as procedural terrain generation. More specifically, the goal of this unit is to determine how devices with different resolutions and other influential hardware features utilise memory while running a Unity project that generates never-ending terrain. Therefore, a comparison between 1080p and 4k devices was employed to determine the limitations and strong points present within their hardware configurations.

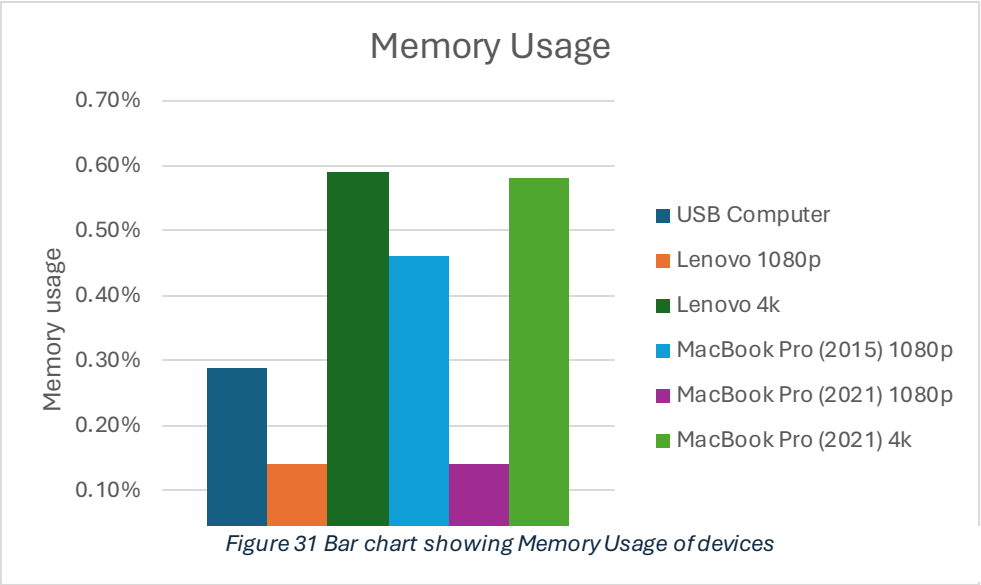

Figure 31 Bar chart showing Memory Usage of devices

Among the devices operating at 1080p resolution, the Lenovo and MacBook Pro (2021) displayed the lowest memory usage, both recording at 0.14%. This indicates a high efficiency in memory management, likely due to more recent hardware optimisations that include better memory allocation strategies and potentially more advanced RAM technologies. The MacBook Pro (2015), despite its age, managed a 0.46% memory usage, suggesting that older models, while less efficient, still handle the memory demands of procedural generation without significant resource overuse.

When evaluating the 4k resolution configurations, the increase in memory usage is noticeable, with the Lenovo recording the highest at 0.59% and the MacBook Pro (2021) closely following at 0.58%. The elevated memory usage in 4k is attributable to the higher data requirements needed to manage and render the increased detail and complexity of the terrain at this resolution. Despite the slight difference in memory usage, the MacBook Pro (2021) shows a slight edge in memory efficiency, which could be linked to its newer generation of hardware that includes integrated memory systems that optimise high-resolution data handling.

To summarise the comparative analysis, it is apparent that new devices perform better in memory usage tests than relatively older ones. The performance gap between the standard resolutions of the 1080 and the 4k resolutions narrows down to negligible differences as the resolution increases. The finding suggests that the hardware factor is a determinant of both the margin of the mean of the performance results and control of the memory intensive requirement.

I was able to infer the peculiarities of handling the memory requirements of procedural terrain generation in Unity by various devices based on their resolution differences, which served to illustrate the profound influence of hardware improvement. Such an analysis is essential for performance optimisation and efficient memory use of applications that perform complex graphics processing.

6 Conclusion

6.1 Fulfilment of Objectives

The project aim was split into three core objectives:

- “To conduct an in-depth examination of existing procedural generation methodologies and tools, focusing on Perlin Noise, Procedural Content Generation Algorithms…”

I am pleased to state that this objective was accomplished through the vast research that was described in the second section of my study. As noted above, this journey began with a comprehensive review of Ken Perlin's work on noise algorithms that form the basis of many procedural generation methods. Such groundwork, in turn, allowed me to gain an intricate understanding of the mechanics of Perlin noise. As a result, I was able to implement this variation into the terrain generation tool, which facilitated the creation of intricate and natural-looking terrain patterns and promoted the tool’s capacity to generate realistic landscapes.

Furthermore, the work discussed the software system of Minecraft, which is famous for using noise algorithms to create vast worlds that span infinitely without any repetition. This particular scenario expanded my mind on procedural generation by accelerating the addition of an endless terrain in my project. This is because after analysing, I learned how Minecraft employs procedural methods in coming up with an array of landscapes every time; then, I improved on that and came up with a technique that not only generates infinite terrain but also doesn’t affect the project’s performance and application.

The combination of these research activities made both my theoretical knowledge and practical application of procedural generation in game development much stronger. Therefore, the established insights from seminal works and successfully implemented practices in the industry significantly supported the development of a solid and dynamically operating terrain generation tool.

- “To design and execute a prototype within Unity3D that demonstrates the practical application of these algorithms, emphasising adaptability, graphical quality, and infinite terrain exploration.”

The main objective was the development of a procedural terrain generation tool and was successfully met through the methodologies presented and demonstrated. Specifically, the Unity tool I have developed stands as a critical technological component for creating vast and infinite terrains necessary for open-world exploration games. The Aim was achieved through a set of rigid development steps, structured testing processes, and step-by-step revision, all of which were thoroughly described in the Methodology and Evualation sections of this dissertation.

This project was conducted with a structured approach that encompassed thorough planning, extensive algorithmic design, and the incorporation of procedural generation tenets that are uniquely designed to elevate the scalability and realism of digital landscapes. Uses of this method enabled not only the development of procedural terrains that can change dynamically as the user navigates through the environment but also the maximal visual realism without significant associated computational expense.

Moreoverthe practical use and outcome were presented , by highlighting the tool’s operation in the demonstartion. In this way, the demonstration was proof of the absolute responsiveness of the tool to the expansive areas, which would be beneficial in any open-world explorer game. The terrain generator coped well with the broader spectrum of ecological and geological elements, as presented in the demonstration. Thus, it seems to be a comprehensive tool for creating natural diversification in games based on a realistic environment.

To sum up, the emergence of the Unity-based terrain generation tool indicates a critical breakthrough in the advancement of procedural technology. The possibilities of generating an

infinite variety of landmasses not only satisfy the present project requirements but also serve as a starting point for the future creation of vast games. Hence, the tool can be defined as a working document for further research and development in the field of large-scale virtual environment production.

- "To integrate an adaptive level of detail and evaluate the performance of the terrain generation process, ensuring optimal balance between high-quality graphics and system efficiency."

This research achieved the objective of improving the tool for generating the terrain based on the described demonstration of the software's most recent version and extended inquiry on the matter in the Methodology. As for the former, regarding the level of detail implementation, Figure 7 presents the core object, which was emphasised as part of the project's development. The feature significantly influences the visual quality and performance optimisation by allowing the terrain to be rendered with increased detail when situated near the viewer and vice versa, managing the process economically.

The integration of the LOD feature into the tool represents a significant improvement in its functionality since it provides the ability to adjust to challenging environments dynamically. Such adaptability is essential in maintaining performance while ensuring that the quality of the user experience is not compromised, especially for applications that require significant resources, including substantial game environments. The development and incorporation of the feature were based on testing and refinement until it could transition between levels of detail without distorting the visual function.

Furthermore, the effective integration of the LOD feature suggests that this development methodology was efficient per se. The methodology was thoroughly structured, prepared, and implemented and featured both the research and actual output, as explained in the Methodology section. As a result, this created space not only to comprehend all technical obstacles of procedural terrain generation but also to develop fresh methods and solutions to address them.

To conclude, the approach structure that was adopted in the development and integration of the LOD feature in the terrain generation tool has significantly boosted its functionality. The results point not only to the practical nature of the applied development methodologies but also place the tool in the current technological landscape as a cutting-edge tool that can produce high-performance, high-detail virtual environments that will be required for the future of digital media applications.

6.2 Personal Development

6.2.1 Unity and C# Development

This project was primarily centred around working with C# as a programming language and the Unity environment in general. My involvement and level of understanding of the project immensely helped me enhance my technical skills with the relevant software. A large variety of scripts to carry out complex functions or integrate complicated algorithms were in C# , that cpntrolled how my terrain dynamics and rendering work. I spent an extensive amount of time optimising, debugging, and solving other issues, which gave me the knowledge I need to have to work with more such coding in the future.

The broad spectrum of development capabilities offered by Unity was an indispensable learning tool; it gave me first-hand experience in all aspects of the development process, from terrain tools to

shader development. This experience was essential in improving my ability to quickly create and modify game environments, as this skill is a cornerstone of become a game developer. The hands-on experience with C# within Unity also helped me implement the theoretical and pratcial knowledge I acquired in various academic modules throughout the last year. For example understanding Unity3d's user interface., and I could apply this knowledge to manage the procedural generation of landscapes and environmental elements.

Moreover, this engagement with Unity allowed me to experiment and excel in different aspects of game development, from physics engines to AI interaction systems, expanding the field of my capabilities. Furthermore, every difficulty that I encountered during the course of the project provided me with a chance to implement the knowledge I obtained from theory and furthered it in practice, creating a spiral of improvement of my understanding and abilities. Overall, the project has not only cemented my technical understanding and practical coding proficiencies but has developed the ability to tackle creative problem solutions, which will allow me to face new challenges in game development in the future

6.2.2 Appericatiоn of Procedural Generation Techniques Used in Industry

In conducting my algorithm-driven and more hands-on exploration of the landscape generation process, I reached a level of increased respect for the varied types of terrains I used to see in video games, especially in the case of those games that use procedural generation for their maps. For every aspect I analysed or created the concept around, including terrain types or the differences between them, I learned to recognise the art and technical skill involved in game design. Again, when looking at complicated software needed to generate landscapes, I often expressed a further understanding of how much detail can be put into every millimetre of surface and how the process would be based on a profound level of mathematics and, at the same time, intricate artistry.

In short, the experience has not only expanded my technical capabilities; it has also developed my understanding of the developer and the gamer. Working on the project, I saw myself applying procedural algorithms like Perlin noise and PCG algorithms, which underlie the creation of organic and captivating virtual worlds.

The convivial perspective and my capability to critique and design both reflect on my own work were also notably improved. A clear understanding of the premise's principles and difficulties of process-generating empowered me to be more critical of game environments. That is, I have achieved exceptional awareness regarding the underlying technological skill and artistic decisions while designating each segment of the game's world. Such an achievement has undoubtedly influenced my design thinking and methodology, encouraging me to invigorate and incorporate the learned knowledge to produce more arresting game experiences. Thus, the practical implementation of process-generating is one of my most meaningful achievements regarding personal and professional development, connecting practical experience with theory and improving my understanding of the sensory domain within which game production takes place.

6.2.3 Problem-Solving Skills

The development of a procedural terrain generation tool presented a severe set of complex challenges that I solved during the development, shaping my approach to such problems. Most prevalent, of course, was the need to switch from a typical surface shader to a URP shader graph to generate textures in a more efficient manner according to the height map regions. It was a massive deal for both visual fidelity and terrain performance, but it also introduced a range of technical limitations, most significantly in the domain of texture mapping.

The primary issue faced initially was the difficulty of correctly applying textures to height regions using the URP shader graph. As mentioned before, the node-based system was much different from the standard shader setup that I was used to. The latter system was more straightforward in terms of connecting textured layers to desired terrain elevations. In contrast, the former required a better understanding of how the nodes interact and how data is passed within the shader graph.

To solve the issue, I utilised several problem-solving methods. First, I isolated the problem identifying the shader graph's processing of the height data as a critical variable of the texture application. Second, I researched multiple node configurations and their impact on the terrain rendering. In this process, I also went through the comprehensive URP documentation and various community forums to obtain more information and potential solutions. Third, I adjusted the node configuration through the trial and error method to correctly integrate the height data in the shader and allow the texture to smoothly transition from one elevation zone to another on the terrain.

Therefore this task has become a real test not only of my technical abilities but also of my need to be inventive in order to use the principles of a standard shader in a way that is more flexible and powerful for this system. Hence to find a solution to the problem, I had to combine hours research through videos and forums with an intricate trial and error processes which ultimately helped me solve the issue and enhance my skills in the spheres of shader programming and the optimisation of the graphicd.

The lessons will show a methodical and comprehensive approach to problem-solving in game development. As the experience overseeing how to combat a workflow challenge from theoretical knowledge, practical gameplay. The terrain improvement through adding URP shader graphs revealed that overcoming technical challenges required strategic rationalisation, which resulted in the noticeable improvement of textures that are set according to starting heightmaps

6.3 Future Work

6.3.1 Enhanced Vegetation Modeling through Advanced PCG Algorithms

the subsequent plan of action is to utilise advanced Procedural Content Generation algorithms customised for vegetation. The goal is to create an even more realistic and diverse environment for the terrains generated in the process. If the tool obtains algorithms that can imitate the spread, population, and kinds of plant life indigenous to numerous environments, it can create more realistic ecosystems for immersive gaming experiences. This development will significantly increase not only the visual realism but also engulf more interactions in the environment as it develops a more comprehensive range of flora types that are reactive and evolving concerning the game's ecological specifications.

6.3.2 Implementing cGANs for Enhanced Computational Efficiency and Terrain Fidelity

The current framework currently utalises Perlins noise and PCG algorithms and has effectively produced dynamic, diverse landscapes essential for immersive gaming environments. Its capabilities can be further advances by integrating conditional Generative Adversarial Networks (cGANs). This strategic upgrade is designed to dramatically improve both the computational efficiency and the fidelity of the terrains generated. The use of cGANs will facilitate the real-time creation of intricately detailed terrains that can be customized to precise environmental parameters such as ecosystems or climatic conditions, offering an unprecedented level of detail and realism.

The integration of cGANs will not only enhance the visual quality and diversity of the terrains but will also accelerate the generation and rendering processes. This improvement is crucial for real-time applications and can significantly reduce development cycles in game production. Additionally, the adaptable nature of cGANs supports continuous enhancements and customization, allowing the toolkit to evolve based on user feedback and the changing needs of game design. This adaptability ensures that the toolkit can dynamically adjust and improve, maintaining its cutting-edge status in terrain generation technology. Research in cGANS conducted in section 2 of this dissertation allow me to evaluate a substantial improvement over the existing methods used. Implementation of cGANs allows for redefinement of standards for realism and efficiency in game development, offering game a powerful tool to craft more engaging and visually captivating gaming experiences.

6.3.3 Diversification through Biome-Specific Terrain Generation

The procedural terrain generation toolkit provides a solid foundation for further expansion into creating a wide array of biomes, each defined by its unique features and obstacles. By producing biome-specific modules within the terrain generator, the software tool could be used to bring into reality a range of terrains, from desert biomes featuring vast, merciless deserts with rolling sand dunes to rainforest biomes offering lush underbrush and multi-leveled canopy to extremely cold biomes that include thorough snow accumulation subject to melt. Such a range would not only increase the beauty and discovery potential of the environments but also open up new challenges and possibilities for game development in any style and genre

References

[1]Perlin, K. (1985). An image synthesiser. *SIGGRAPH Comput. Graph.*, 19(3), 287-296.

[2]Wulff-Jensen, A., Sørensen, N. R., & Risi, S. (2017). Procedural Generation of Terrain with Conditional Generative Adversarial Networks. *Proceedings of the Genetic and Evolutionary Computation Conference*.

[3]Guérin, E., Digne, J., Galin, E., & Peytavie, A. (2017). Interactive example-based terrain authoring with conditional generative adversarial networks. *ACM Transactions on Graphics (TOG)*, 36(6), 1-13.

[4]Summerville, A. (2018). Procedural Content Generation via Machine Learning (PCGML)

[5] Mojang. 2011. Minecraft (Version 1.0). [Video Game].