**A project report submitted in partial fulfilment for the degree of**

**BSc (Hons) Computer Games Development**

**School of Psychology and Computer Science**

**University of Central Lancashire**

Project Title

Jonathan Mills

**20th April 2022**

# Abstract

It is suggested that the abstract be structured as follows:

* Problem: What you tackled, and why this needed a solution
* Objectives: What you set out to achieve, and how this addressed the problem
* Methodology: How you went about solving the problem
* Achievements: What you managed to achieve, and how far it meets your objectives.

# Attestation

I understand the nature of plagiarism, and I am aware of the University’s policy on this.

I certify that this document reports original work by me during my University project. I also confirm that I adhere to the University’s legal and ethical guidelines for undergraduate projects in Computing.

Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date:

# Acknowledgements

# Table of Contents

[Abstract i](#_Toc99393667)

[Attestation ii](#_Toc99393668)

[Acknowledgements iii](#_Toc99393669)

[Table of Contents iv](#_Toc99393670)

[List of Figures vii](#_Toc99393671)

[List of Tables viii](#_Toc99393672)

[List of Listings ix](#_Toc99393673)

[1 Introduction 1](#_Toc99393674)

[2 Background and Related Work 2](#_Toc99393675)

[2.1 Introduction 2](#_Toc99393676)

[2.2 What is procedural generation and why is it used within games? 2](#_Toc99393677)

[2.2.1 Generation of plants and animals 3](#_Toc99393678)

[2.2.2 Generation of levels and worlds 4](#_Toc99393679)

[2.3 Procedural Generation Techniques 6](#_Toc99393680)

[2.3.1 Introduction 6](#_Toc99393681)

[2.3.2 Outdated techniques used 6](#_Toc99393682)

[2.3.3 Newer techniques used 8](#_Toc99393683)

[2.4 User Evaluation 10](#_Toc99393684)

[2.5 Summary 11](#_Toc99393685)

[3 Project Planning 12](#_Toc99393686)

[3.1 Introduction 12](#_Toc99393687)

[3.2 Methodology 12](#_Toc99393688)

[3.3 Requirements 12](#_Toc99393689)

[3.4 Potential Solutions 12](#_Toc99393690)

[3.5 Tools and Techniques 12](#_Toc99393691)

[3.6 Legal, Social, and Ethical Issues 12](#_Toc99393692)

[3.6.1 First Sub Section 12](#_Toc99393693)

[3.6.2 Second Sub Section 12](#_Toc99393694)

[3.7 Another Section 12](#_Toc99393695)

[3.8 Summary 12](#_Toc99393696)

[4 Design 13](#_Toc99393697)

[4.1 Introduction 13](#_Toc99393698)

[4.2 System Design 13](#_Toc99393699)

[4.3 User Interface Design 13](#_Toc99393700)

[4.4 Summary 13](#_Toc99393701)

[5 Implementation 14](#_Toc99393702)

[5.1 Introduction 14](#_Toc99393703)

[5.2 DirectX and the shader integration 14](#_Toc99393704)

[5.2.1 Geometry shader 14](#_Toc99393705)

[5.2.2 Pixel Shader 14](#_Toc99393706)

[5.3 Building the Terrain 14](#_Toc99393707)

[5.4 Procedural generation algorithms 14](#_Toc99393708)

[5.4.1 Perlin Noise algorithm 14](#_Toc99393709)

[5.4.2 Diamond Square algorithm 16](#_Toc99393710)

[5.4.3 Rigid and Inverse Rigid Noise 18](#_Toc99393711)

[5.4.4 Perlin with Octaves 18](#_Toc99393712)

[5.4.5 Terracing 19](#_Toc99393713)

[5.5 Summary 20](#_Toc99393714)

[6 Test Strategy and Evaluation 21](#_Toc99393715)

[6.1 Introduction 21](#_Toc99393716)

[6.2 Functional Testing 21](#_Toc99393717)

[6.3 Non-Functional Testing 21](#_Toc99393718)

[6.4 User Testing 21](#_Toc99393719)

[6.5 Evaluation 21](#_Toc99393720)

[6.6 Summary 21](#_Toc99393721)

[7 Evaluation, Conclusions and Future Work 22](#_Toc99393722)

[7.1 Project Objectives 22](#_Toc99393723)

[7.2 Self-Evaluation 22](#_Toc99393724)

[7.3 Project Evaluation 22](#_Toc99393725)

[7.4 Applicability of Findings to the Commercial World 22](#_Toc99393726)

[7.5 Conclusions 22](#_Toc99393727)

[7.6 Future Work 22](#_Toc99393728)

[References 23](#_Toc99393729)

[Appendix 1 – Project Proposal 25](#_Toc99393730)

[Project Context 25](#_Toc99393731)

[Specific Objectives 25](#_Toc99393732)

[Potential Ethical or Legal Issues 25](#_Toc99393733)

[Resources 25](#_Toc99393734)

[Potential Commercial Considerations - Estimated costs and benefits 26](#_Toc99393735)

[Proposed Approach 26](#_Toc99393736)

[References 27](#_Toc99393737)

[Appendix 2 – Technical Plan 28](#_Toc99393738)

[Title 28](#_Toc99393739)

[Summary 28](#_Toc99393740)

[Deliverables 28](#_Toc99393741)

[Constraints 28](#_Toc99393742)

[Key Problems 28](#_Toc99393743)

[System and Work Outline 29](#_Toc99393744)

[Project Activities 29](#_Toc99393745)

[Risk Analysis 29](#_Toc99393746)

[Risk 29](#_Toc99393747)

[Severity 29](#_Toc99393748)

[Likelihood 29](#_Toc99393749)

[Action 29](#_Toc99393750)

[Options 29](#_Toc99393751)

[Potential Ethical or Legal Issues 31](#_Toc99393752)

[Commercial Analysis 31](#_Toc99393753)

[Factor name 31](#_Toc99393754)

[Description 31](#_Toc99393755)

[Is this a cost or a benefit 31](#_Toc99393756)

[Estimated Amount 31](#_Toc99393757)

[Estimate of when paid 31](#_Toc99393758)

[Employability Contribution 32](#_Toc99393759)

[References 32](#_Toc99393760)

[Appendix 3 – Title of Appendix 33](#_Toc99393761)

# List of Figures

[Figure 1 - several plants generated with stochastic L-System (Freiknecht, 2020) 3](file:////Users/jontymills/Documents/GitHub/TerrainEngine/DraftReport-MillsJonathan.docx#_Toc99394975)

[Figure 2 - Spore, an example of the creature creation screen (Beale, 2008) 4](file:////Users/jontymills/Documents/GitHub/TerrainEngine/DraftReport-MillsJonathan.docx#_Toc99394976)

[Figure 3 - Map Generated using Cellular automata (van der Linden, Lopes, & Bidarra, 2014) 5](file:////Users/jontymills/Documents/GitHub/TerrainEngine/DraftReport-MillsJonathan.docx#_Toc99394977)

[Figure 4 - Dwarf Fortress an example of procedural content generation of worlds (Grant 2012) 6](file:////Users/jontymills/Documents/GitHub/TerrainEngine/DraftReport-MillsJonathan.docx#_Toc99394978)

[Figure 5 - stages of the midpoint displacement method. (Snook, 2003) 7](file:////Users/jontymills/Documents/GitHub/TerrainEngine/DraftReport-MillsJonathan.docx#_Toc99394979)

[Figure 6 – Example of terrain generation on a 5x5 heightmap (Mijailovic, 2015) 7](file:////Users/jontymills/Documents/GitHub/TerrainEngine/DraftReport-MillsJonathan.docx#_Toc99394980)

[Figure 7 - set up of the Perlin noise function (Snook, 2003) 9](file:////Users/jontymills/Documents/GitHub/TerrainEngine/DraftReport-MillsJonathan.docx#_Toc99394981)

[Figure 8 - Calculation of the height value for a pixel (Snook, 2003) 9](file:////Users/jontymills/Documents/GitHub/TerrainEngine/DraftReport-MillsJonathan.docx#_Toc99394982)

[Figure 9 – comparison between Perlin noise and Simplex noise - (Gustavson, 2005) 9](file:////Users/jontymills/Documents/GitHub/TerrainEngine/DraftReport-MillsJonathan.docx#_Toc99394983)

[Figure 10 - Reef structures designed by an artist (top) and generated procedurally by Perlin noise (bottom) 10](file:////Users/jontymills/Documents/GitHub/TerrainEngine/DraftReport-MillsJonathan.docx#_Toc99394984)

[Figure 11 – Terrain generated with a seed of 125 using the Perlin noise algorithm 15](file:////Users/jontymills/Documents/GitHub/TerrainEngine/DraftReport-MillsJonathan.docx#_Toc99394985)

[Figure 12 - Terrain generated with a seed of 0 using the Perlin noise algorithm 15](file:////Users/jontymills/Documents/GitHub/TerrainEngine/DraftReport-MillsJonathan.docx#_Toc99394986)

[Figure 13 - Terrain generated using the Diamond square algorithm 16](file:////Users/jontymills/Documents/GitHub/TerrainEngine/DraftReport-MillsJonathan.docx#_Toc99394987)

[Figure 14 - Terrain generated using the Rigid Noise algorithm 18](file:////Users/jontymills/Documents/GitHub/TerrainEngine/DraftReport-MillsJonathan.docx#_Toc99394988)

[Figure 15 - Terrain Generated using the Inverse Rigid Noise algorithm 18](file:////Users/jontymills/Documents/GitHub/TerrainEngine/DraftReport-MillsJonathan.docx#_Toc99394989)

[Figure 16 - Terrain generated using the Perlin with Octaves algorithm 19](file:////Users/jontymills/Documents/GitHub/TerrainEngine/DraftReport-MillsJonathan.docx#_Toc99394990)

[Figure 17 - Terrain altered by the terracing function 20](file:////Users/jontymills/Documents/GitHub/TerrainEngine/DraftReport-MillsJonathan.docx#_Toc99394991)

# List of Tables

[Table 1 - Test Results 22](#_Toc99372603)

# List of Listings

[Listing 1 – [CPerlinNoise.cpp] Generation of Perlin noise value for a set of coordinates 16](#_Toc99392496)

[Listing 2 - [DiamondSquare.cpp] Square step of the diamond square algorithm 17](#_Toc99392497)

[Listing 3 – [DiamondSquare.cpp] Diamond step of the diamond square algorithm 17](#_Toc99392498)

[Listing 4 - [TerrainGenerationScene] updating the HeightMap using Perlin noise with octaves 19](#_Toc99392499)

[Listing 5 - [TerrainGenerationScene.cpp] Terracing function 20](#_Toc99392500)

# Introduction

# Background and Related Work

## Introduction

This project aims to make use of procedural generation to create terrain and terrain effects using the DirectX library. The content discussed within this chapter will be around the related topics surrounding this subject and will be split into three sections. The first section will delve into what procedural generation is and why it is used in the creation of video games within the games industry and will discuss the advantages and disadvantages of using this method. The second section will go into depth about the various techniques that can be used to apply procedural generation. It discusses various methods for generating the terrain procedurally and will look at how this was done early on within the industry and compare how these methods stand up to the newer ones introduced that are currently being used. The final section will go into an evaluation on the end user and provide information of what the end user thinks of the application of procedural generation within games and how to keep the user invested while using the project.

## What is procedural generation and why is it used within games?

Procedural generation is a method for creating data algorithmically, without too much input from the creator or developer. Procedural generation is normally used to create content for video games or for animated movies, such as landscapes, animation of 3D objects.

Procedural content generation has become a more frequently used method within video games over the past couple of years, furthermore it has also seen a use in the early history of the industry in such games as *civilisation* (Sid Meier, 1991) and *Rogue* in the 80’s. Video game developers are increasingly using procedural generation for everything within their game; however, the focus has been in creating visually appealing content for the user while using less development resources.

Hendrikx et al (2013) talks about how procedural generation can be used to fight the “new scalability challenges due to the exponential growth over the last decade”. Video games have been increasing in their popularity year over year as reported in the annual report released by the Entertainment Software Association (Entertainment Software Association, 2021) that claims in the United States there are nearly 227 million people playing video games. Due to the increasing popularity of games, the games industry has become very competitive in the development of triple A games, leading them to become larger and more expensive, allowing procedural generation to provide a practical method at reducing these costs. The main cost that is reduced is the time and effort taken to create levels and the artwork associated with these levels. The use of procedural generation allows many similar levels to be created with as few art assets created as necessary. By having an artist create a handful of resources that can be reused by the procedural algorithm allows for lower costs (Green, 2016).

While this method provides several benefits such as speeding up and reducing the costs of video game development there are several downsides to using this method. One of the largest downsides is the requirement of optimal high-end hardware to consistently generate content efficiently. Creating games with procedural generation is done through algorithms that can be intensive and require a lot of computing power (Green, 2016). Due to this, more effort needs to be put into optimising these algorithms so that the game created can be played by anyone and does not require a high-end computer. Another big downside of procedural generation is that the content created can begin to start feeling repetitive. When creating large worlds but only a few basic algorithms are used will inevitably create a lot of repetitive areas being generated (Green, 2016).

### Generation of plants and animals

A picture containing tree, sky, outdoor, plant

Description automatically generatedFreiknecht (2020) discusses how the procedural creation of objects occurring in nature belongs to one of the most explored areas within procedural content generation. Plant recreations within graphical programs have been done extensively for the past several decades, even as early as 1968 where *L-Systems* (Lindenmayer Systems) were introduced to describe the structure of plants (Freiknecht, 2020). L-Systems are created by using a system of production rules that are then applied to a chosen collection of symbols that usually describe how a plant grows. The rules applied to L-Systems can be as extensive as required. Figure 1 shows how L-Systems can be utilised over several iterations to produce a realistic looking plant. Fornander (2013) describes how a third-party plug-in called SpeedTree that implements L-Systems is used for the procedural creation of vegetation, some examples include *Resident Evil Village* (Capcom, 2021) and *Ghost of Tsushima* (Sucker Punch Productions,2020*).*

Figure - several plants generated with stochastic L-System (Freiknecht, 2020)

Graphical user interface

Description automatically generatedFreiknecht (2020) states that research on procedurally generation of creatures is “few and far between” (p. 29). However, a video game called *Spore* (Maxis, 2008), introduced game-play featuring the ability for players to create a microscopic organism that could develop into a highly intelligent creature. Designer Will Wright proposed the approach of using procedural generation to create thousands of assets that would be needed when developing the game, Figure 2 shows this method in effect as the player is able to change the colour and tecture of the creature to anything that they would like and the new assets are created during runtime. Freiknecht (2020) discusses how based on the development of *Spore,* Hecker et al, introduced a system to animate creatures with unknown body types during runtime by getting a set of variables from the creature generation and then translating them into a visual model and then animating it. A more modern game that uses procedural generation is No Man’s Sky (Hello Games, 2016), which uses this technique to generate trillions of planets and the creatures that are found within these planets. This reintroduces the problem of repetitive content when using procedural generation for the bulk of the content within a game, however, it is still an amazing technical feat.

Figure - Spore, an example of the creature creation screen (Beale, 2008)

Procedural generation of animals and creatures in video games has generally been about the placement of the creatures within the game world based on the parameters created when the in-game world was created, such as adding deer anywhere there are large areas of grassland or adding fish to areas of water. Furthermore, the design of animals and creatures have largely still been left to the artists working on the game with the only part of the process that procedural generation can sometimes affect is the generation of textures needed for the animal object.

### Generation of levels and worlds

Procedural generation of content has almost always been used with the generation of dungeon levels within video games with some examples being Diablo (Blizzard Entertainment, 1998) and Don’t Starve (Klei Entertainment, 2013), one of the biggest reasons for the use of this method is the replayability it provides to the end user. The use of this method allows for the reduction in production costs as described earlier in this chapter and it can increase the creativity of the designers, moreover, allowing them to focus on other aspects of the game instead of level design.

A picture containing text

Description automatically generatedOne method of procedurally generating dungeon levels is through Cellular Automata, which is a grid of cells that reference their neighbours, rule sets are then applied to the current state of the cell along with its neighbouring ones. Patterns will begin to form after several generations/iterations through the grid (van der Linden, Lopes, & Bidarra, 2014). This method allows for the easy manipulation of each cell through parameters and constraints, an example would be checking whether all the neighbouring cells are empty and assuming that the current cell is empty as well. Figure 3 shows what the result of Cellular automata generates based on certain parameters and how it resembles what a dungeon level would look like with the grey areas being the level floor and the white area representing the walls of the level.

Figure - Map Generated using Cellular automata (van der Linden, Lopes, & Bidarra, 2014)

The procedural generation used in *dwarf fortress* (Bay 12 Games, 2006) has not been explicitly explained by the creators of the game, however, based on Figure 4, it would be reasonable to assume that cellular automata could have been used when creating the game world as it looks very similar to the map in Figure 3. Furthermore, dwarf fortress has also used procedural generation for the creation of the history and religions of the generated world, with these being completely different each playthrough of the game. This goes to show that procedural generation can be very helpful in the creation of video games as dwarf fortress started out as a two-person indie project and the use of procedural generation allowed them to focus on other aspects of the game required for them to release the game.

Map

Description automatically generated

Figure - Dwarf Fortress an example of procedural content generation of worlds (Grant 2012)

## Procedural Generation Techniques

### Introduction

Procedurally generated content has been implemented in dozens of different games in a variety of ways throughout the past couple of decades, with No Man’s Sky’s’ entire concept, relying on the procedural generation of planets and creatures and dwarf fortressusing procedural generation to generate nearly all aspects of the game. This section aims to discuss the several different techniques that are used within the industry to create this content, how they can be applied to this project and whether they are practical to use. Techniques that will not be used will also be discussed and researched to provide a better understanding of the other techniques that could have been used within the project.

One of the most basic methods of storing terrain data are heightmaps, which are made up of a two-dimensional array with indices indicating the X and Z coordinates and the value indicating the Y value of the terrain. Mijailovic (2015) describes heightmaps as being fast, in terms of data access, and intuitive to use, going on to state that they “became the backbone of many algorithms” (p.7). However, Heightmaps have one big disadvantage of only storing one height value for each coordinate pair, leading to the generation of very simple terrain without any complex structures, such as caves or overhanging cliffs. Voxels – volumetric pixels - have been introduced as a way to counter this and leading to easier generation of complex terrain structures. Each method introduced within this next section, take heightmaps as an input and fill all the values within the heightmap with generated points.

### Outdated techniques used

#### Midpoint displacement

Midpoint displacement was one of the first algorithms introduced for the procedural generation of content in 1982. This method of procedural generation is implemented by splitting the image into four quadrants and adjusting each corner by a random value, each quadrant is then treated as a new image and the process is repeated, while this is happening the range to affect the corners is decreased each time a new image is identified (Snook, 2003). Figure 5 shows each numbered point is set to the average of the two corners they are connected to and how this process is done recursively with every additional quadrant created. This method of procedural generation of terrain is very effective at creating random terrain and can have constraints added to allow the programmer to have more control of the structure created,

A picture containing indoor

Description automatically generatedOne problem that occurs during the use of the midpoint displacement method is that patterns can start emerging and can be very noticeable when looking at the terrain from different angles (Losh, Terrain Generation with Midpoint Displacement, 2016). While natural environments can have some repeating patterns, normally this is done through human involvement, it will be important to take into consideration the end user when making use of the midpoint displacement method, and repetitive terrain can be jarring or annoying for the end user to look at.

Figure - stages of the midpoint displacement method. (Snook, 2003)

within the context of this project, this method of procedural terrain generation would not be effective when creating the terrain as it will become repetitive quickly and the recursive nature of this method can be computationally taxing when generating a large environment. On the other hand, this repetitive pattern is not guaranteed to occur due to the random nature of procedural generation, however, this is not enough to justify using this method within the context of this project.

#### Diamond square

Chart, line chart

Description automatically generatedThe Diamond square method is an improvement on the midpoint displacement method through ensuring that every point uses four sources of data (Losh, Terrain Generation with Diamond Square, 2016). This method is implemented similarly to the midpoint displacement method; however, diamonds are used instead of the square. This method will start out by giving the corners of the image random values and calculating the centre of the image from the average of these values. The algorithm will then iterate through the image and create diamonds to help with the generation of height values. Figure 6 shows an example of this algorithm working when given a 5x5 grid, and shows how the algorithm is also done recursively, similarly to the midpoint displacement algorithm.

Figure – Example of terrain generation on a 5x5 heightmap (Mijailovic, 2015)

Miller (1986) performed an analysis on the diamond square method and found that this method of terrain generation was flawed due to the ‘Tell-tale vertical streaks’ indicating a persistent creasing problem due to the calculations taking place within a rectangular grid. Another big disadvantage of using the diamond square method is that non-repeating infinite terrain cannot be created easily (Mijailovic, 2015), as each heightmap would search for an additional heightmap, that doesn’t exist, next to it for lines near the boundaries to break.

The observation from Miller along with the recursive nature of the algorithm are the reasons why this method will not be implemented within the scope of the project. Despite this, the diamond square method is a major improvement on the midpoint displacement method when generating terrain without patterns appearing.

### Newer techniques used

Commonly used algorithms for the procedural generation of terrain are noise algorithms. These algorithms are commonly used for the generation of height maps that would then be used for terrain generation. These algorithms are great for creating a 2D height map describing how high everything in the environment should be, however, when trying to create areas with overhanging structures such as cliff faces, these 2D versions would not provide enough detail. 3D versions can be created; however, they are used for describing how densely populated areas are within the environment.

#### Perlin Noise

Perlin noise is the most used algorithm for the procedural generation of terrain, several game engines even have plug-ins or built-in implementations of this algorithm, such as Unity since it is the most popular method. Noise is the random unstructured number generator of computer graphics, and the random patterns are often described in terms of frequency (Lagae, et al., A survey of Procedural Noise Functions, 2010). Due to the nature of this method, the amount of terrain that could be generated is nearly endless, while only changing certain parameters and constraints. Two different noise maps can be generated or constructed with this noise algorithm as mentioned earlier, height and density maps. For this project, the height maps are more important as the gradient of the terrain will be generated and the focus will not be on how populated areas are within the terrain.

To generate Perlin noise in 2 dimensions, a pseudo-random 2D normal vector – unit length pointing in a random direction - is placed on each grid point (Snook, 2003), as shown in Figure 7. Snook (2003) goes on to describe how the height value for each point, within a grid square is calculated by getting the vector to the point for each corner of the grid and performing a dot product to get the resultant height value, Figure 8 demonstrates this. This is how Ken Perlin originally introduced this algorithm and it proved to be an effective way to generate random noise and has since been a very popular method to procedurally generate content. This project aims to implement the Perlin Noise algorithm this way to generate the height map for the terrain that will be generated.

Calendar

Description automatically generatedShape, polygon

Description automatically generated

Figure - set up of the Perlin noise function (Snook, 2003)

Figure - Calculation of the height value for a pixel (Snook, 2003)

#### Simplex Noise

A picture containing text, mollusk, cowrie

Description automatically generatedSimplex noise was created in 2001 when Ken Perlin introduced it as an improvement on his noise algorithm to overcome certain limitations (Gustavson, 2005). Simplex noise involves the use of simplex grids - which is the simplest and most compact shape that can fill an entire space – to handle *N* dimensions. Gustavson (2005) tells us that simplex noise was designed to have a lower computational complexity with fewer required multiplications than Perlin noise. Figure 9 shows how Simplex noise compares to Perlin noise when generating noise within a 2D environment, showing how Simplex noise generates an image with more detail demonstrating clearer peaks and troughs while Perlin noise generates an image with less detail it does, however, show more variation. Moreover, the simplex noise generated an image with sharper gradients between the points, as seen with super dark spots surrounded by a white area. This could lead to the terrain being generated looking less realistic to the user.

Figure – comparison between Perlin noise and Simplex noise - (Gustavson, 2005)

While simplex noise is less computationally complex, it was designed to handle a higher number of dimensions, allowing for the quick generation of large structures. Furthermore, the generation of sharper gradients between points within the noise image would be useful when generating large mountains with huge cliff faces which this project does not aim to do. An additional point is that the implementation of Simplex noise in 3 dimensions and higher is currently patented by Ken Perlin (United States Patent No. US6867776B2, 2001), with it currently being expected to expire in early 2022, I do not plan to take this risk when generating terrain in the project.

## User Evaluation

While this project is aimed to create realistic environments, it should be important to consider what would help engage the user. Adding water to generated terrains that produce valleys could be a way to increase the realism of the project. When producing procedurally generated content that would be used within video games, it should be important to consider what would be realistic within the given context, otherwise the player would get bored or uninterested due to the complete randomness of the environment with very little detail. A study found that out of a group of 41 people, 60% preferred Procedurally generated reefs compared to static reefs made by artists (Korn, et al., 2017), figure 10 shows an example of what the participants of the study were shown. This justifies the use of procedurally generated content within video games as the end users prefer the dynamically changing look of environments within video games, however, if the game has a detailed story and focuses heavily on this storyline, the use of procedural generation would be a drawback as it would take away from the story.

A picture containing text, ocean floor

Description automatically generated

Figure - Reef structures designed by an artist (top) and generated procedurally by Perlin noise (bottom)

Another important aspect of realistic environments when done through procedural generation is the use of texturing and visuals of the environment. If the same texture is used throughout the whole environment, especially in complex environments, the end user would become annoyed or uninterested to explore the environment as it would look the same everywhere, they go. End users are getting used to playing games that are visually appealing to them with exciting environments to explore due to the advancements in graphical technology, leading to higher quality and larger textures being produced.

This project will aim to create an environment that would be as realistic as possible without sacrificing the accuracy found within nature, furthermore, trying to create a realistic environment that is also fun to explore for the end user. A balance will have to be struck between these two points depending on what the project would be intended for. If this project were to be used for the generation of game levels or gameplay, artistic creativity would have to take priority compared to the realism of the generation. Although, if this project were to be used for the simulation of realistic environments, the accuracy of the environment would have to take priority over the artistic direction of the generation.

## Summary

With the information that has been researched and discussed within this chapter, I should be able to apply the appropriate methods and techniques within the scope of the project. The information found about the techniques that will not be used has helped considerably and will help towards important decisions throughout the development cycle of the project.

# Project Planning

## Introduction

## Methodology

Content goes here.

## Requirements

Content goes here.

## Potential Solutions

Content goes here.

## Tools and Techniques

Content goes here.

## Legal, Social, and Ethical Issues

### First Sub Section

### Second Sub Section

Content goes here.

## Another Section

## Summary

# Design

## Introduction

This chapter focuses on the design of the Engine itself as well as the user interface.

## System Design

Content goes here.

A screenshot of a computer

Description automatically generated with medium confidence

## User Interface Design

Content goes here.

A screenshot of a computer

Description automatically generated with medium confidence

## Summary

# Implementation

## Introduction

All of the project’s content comes from procedural content/terrain algorithms. This chapter will start with a small discussion on the shaders that helped with creating the terrain and then delve into how these algorithms work and how they fit together to create the final Terrain.

## DirectX and the shader integration

One of the reasons that DirectX was chosen for this project was because of how easy it is to add new shaders for different effects on the terrain. While only two shaders were used for the editing of the terrain, it becomes very handy to have the easy option of adding new shaders to the project if needed to produce new effects.

### Geometry shader

Geometry shaders allow the developer to manipulate the primitives of an object instead of the vertices or pixels. The addition of a Geometry shader to this project allowed the normal for the terrain model to be calculated by the Geometry shader and passed on to the pixel shader without the normals being pre-calculated.

### Pixel Shader

Pixel shaders operate on every pixel in a 2D polygon. The pixel shader used within this project relied on the normals calculated from the geometry shader to then work out the slope of the normal. The calculated slope is then used to shade the terrain based on the slope of the pixel leading to more realistic looking terrain being generated.

## Building the Terrain

In order to build the terrain that would be interacted with by the procedural generation algorithms, a grid mesh had to be made that would be sent over to the GPU for it to make. This grid mesh was done by going through every point of a user defined grid and setting the Y values of the point to the corresponding value in the HeightMap while updating the x and z values to be the next point in the grid at the end of each loop. Building the terrain through this method, allowed the program easy access to the coordinates of the terrain as it gets rebuilt every-time the heightmap gets updated by any of the algorithms.

## Procedural generation algorithms

### Perlin Noise algorithm

The Perlin noise algorithm is one of the staples in the games industry when it comes to generation random content or terrain as it can be nearly endless with the randomness it generates. As seen by the two different terrains generated with different seeds in Figure 12 and 11. The first implementation of this algorithm within this project was done following Ken Perlin’s original implementation in Java, however the terrain generated from this implementation produced very jagged hills. Due to this, the noise value returned was then given the blended value from all of its edges, leading to the produced terrain to become smoother and look more like a bunch of hills.

A picture containing green, plant

Description automatically generatedA picture containing green

Description automatically generated

Figure – Terrain generated with a seed of 125 using the Perlin noise algorithm

Figure - Terrain generated with a seed of 0 using the Perlin noise algorithm

The Perlin noise algorithm generates points using a permutation List (a list containing values from 0 – 255 placed randomly throughout) either premade or randomly sorted using a seed as done in this project. The noise function will generate permutation coordinates from the current heightmap coordinates and creates fade curves for them as well. The corners of the permutation cube are then calculated and the noise value is calculated by blending the 8 edges of the permutation point and using the fade curves.

double CPerlinNoise::noise(double x, double y, double z)

{

// Find the unit cube that contains the point

int X = (int)floor(x) & 255;

int Y = (int)floor(y) & 255;

int Z = (int)floor(z) & 255;

// Find relative x, y,z of point in cube

x -= floor(x);

y -= floor(y);

z -= floor(z);

// Compute fade curves for each of x, y, z

double u = fade(x);

double v = fade(y);

double w = fade(z);

// Hash coordinates of the 8 cube corners

int A = permutationList[X] + Y;

int AA = permutationList[A] + Z;

int AB = permutationList[A + 1] + Z;

int B = permutationList[X + 1] + Y;

int BA = permutationList[B] + Z;

int BB = permutationList[B + 1] + Z;

// Add blended results from 8 corners of cube

double res = LERP(w, LERP(v, LERP(u, grad(permutationList[AA], x, y, z), grad(permutationList[BA], x - 1, y, z)), LERP(u, grad(permutationList[AB], x, y - 1, z), grad(permutationList[BB], x - 1, y - 1, z))), LERP(v, LERP(u, grad(permutationList[AA + 1], x, y, z - 1), grad(permutationList[BA + 1], x - 1, y, z - 1)), LERP(u, grad(permutationList[AB + 1], x, y - 1, z - 1), grad(permutationList[BB + 1], x - 1, y - 1, z - 1))));

return (res + 1.0) / 2.0;

}

Listing – [CPerlinNoise.cpp] Generation of Perlin noise value for a set of coordinates

### Diamond Square algorithm

A picture containing outdoor, nature, rock, distance

Description automatically generatedThe Diamond square algorithm is one of the oldest techniques used for the procedural generation of terrain within games dating as far back as 1986. One of the biggest problems of this technique is that is can produce very pointy mountain tops as shown in Figure 13. A big limitation of using this algorithm is ensuring that the length of the heightmap was equal to 2n-1 otherwise the algorithm would not be able to divide the heightmap equally into quadrants.

Figure - Terrain generated using the Diamond square algorithm

This algorithm splits the terrain into quadrants and in each quadrant performs the square and diamond step before splitting up the quadrants further. During each square step of this algorithm, the four corners of the current quadrant are averaged out and a random offset is added to this value to give the height value of the centre point.

//SQUARE STEP

for (int x = 0; x < m\_Size - 1; x += sideLength)

{

for (int y = 0; y < m\_Size - 1; y += sideLength)

{

//x, y is upper left corner of square

//calculate average of existing corners

double avg = HeightMap[x][y] //top left

+ HeightMap[x + sideLength][y]//top right

+ HeightMap[x][y + sideLength]//lower left

+ HeightMap[x + sideLength][y + sideLength];//lower right

avg /= 4.0;

//add a random value on to the average

HeightMap[x + halfSide][y + halfSide] = abs(avg +

fRand2(-m\_Spread, m\_Spread));

}

}

Listing - [DiamondSquare.cpp] Square step of the diamond square algorithm

During each diamond step of this algorithm, the four centre edge points of the current quadrant are averaged out with a random offset added to give the updated height value of the centre point. Do to the recursive nature of this algorithm the terrain can start to look blocked out.

//DIAMOND STEP

for (int x = 0; x < m\_Size - 1; x += halfSide)

{

//and y is x offset by half a side, but moved by

//the full side length

//NOTE: if the data shouldn't wrap then y < DATA\_SIZE

//to generate the far edge values

for (int y = (x + halfSide) % sideLength; y < m\_Size - 1; y += sideLength)

{

//x, y is center of diamond

//note we must use mod and add DATA\_SIZE

//for subtraction

//so that we can wrap around the array to find the corners

double avg = HeightMap[(x - halfSide + m\_Size - 1)

% (m\_Size - 1)][y] //left of center

+ HeightMap[(x + halfSide) % (m\_Size - 1)][y] //right of center

+ HeightMap[x][(y + halfSide) % (m\_Size - 1)] //below center

+ HeightMap[x][(y - halfSide + m\_Size - 1) % (m\_Size - 1)]; //above center

avg /= 4.0;

//add a random value on to the average

avg = abs(avg + fRand2(-m\_Spread, m\_Spread));

//update value for center of diamond

HeightMap[x][y] = avg;

//wrap values on the edges, remove

//this and adjust loop condition above

//for non-wrapping values.

if (x == 0) HeightMap[m\_Size - 1][y] = avg;

if (y == 0) HeightMap[x][m\_Size - 1] = avg;

}

}

Listing – [DiamondSquare.cpp] Diamond step of the diamond square algorithm

### Rigid and Inverse Rigid Noise

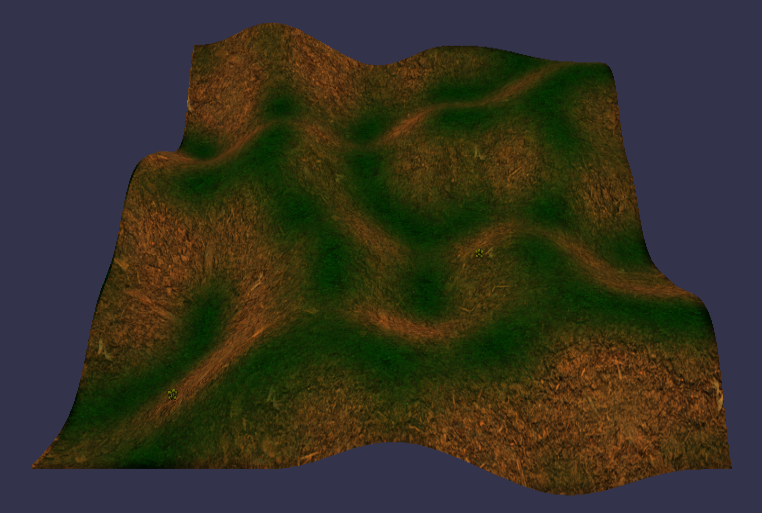
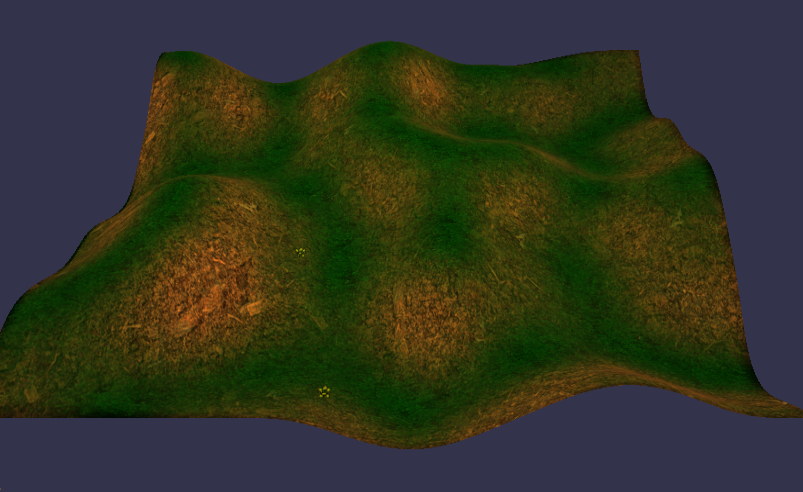
Rigid and Inverse Rigid Noise are very similar to the Perlin Noise algorithm as they are derived from it, as the name suggests, the inverse rigid noise algorithm produces terrain that is the inverse of the rigid noise generated terrain as shown in Figure 14 and Figure 15. Rigid Noise is produced by subtracting the absolute value of the generated Perlin noise value subtracted from one. The negated value is then added to the current height maps value, creating terrain that looks extremely similar to the generated terrain created by the Perlin noise algorithm. Inverse Rigid Noise is done through the same method; however, the value does not get negated when added to the current height value, this leads to the terrain looking sunk down and to be the inverse of the terrain generated through the rigid noise algorithm.

Figure - Terrain generated using the Rigid Noise algorithm

Figure - Terrain Generated using the Inverse Rigid Noise algorithm

### Perlin with Octaves

Perlin noise with Octaves is done by looping through the number of octaves chosen and multiplying the amplitude and frequency of the map each loop, this algorithm is an improved version of the original Perlin noise algorithm as it produces a higher quality piece of terrain. Figure 16 demonstrates this well as it looks similar to the terrain generated by Perlin noise shown earlier in Figure 12 but with an increased level of detail around the cliffs. This method of generating terrain is increasingly becoming more popular within the games industry as it is fairly inexpensive to produce more detail using the same algorithm.

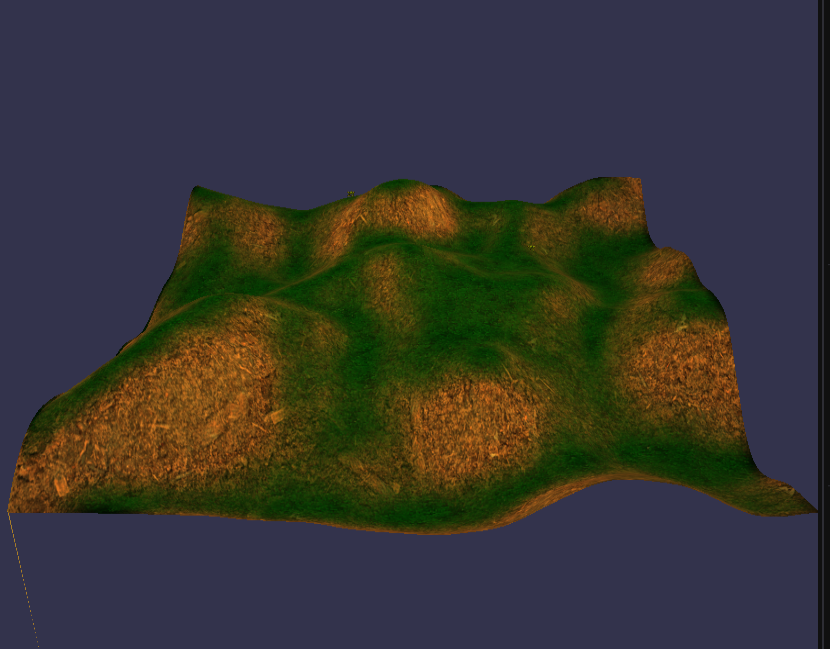


Figure - Terrain generated using the Perlin with Octaves algorithm

for (int i = 0; i < octaves; ++i)

{

//Update the HeightMap with the Perlin noise function using the current Amplitude and Frequency

BuildPerlinHeightMap(Amplitude, frequency, true);

//Update the Amplitude and Frequency variables

Amplitude \*= AmplitudeReduction;

frequency \*= FrequencyMultiplier;

}

Listing - [TerrainGenerationScene] updating the HeightMap using Perlin noise with octaves

The amount that the amplitude and frequency can be multiplied by can be edited by the user along with the frequency and amplitude of the terrain map themselves. This allows for the algorithm to produce different terrain on the fly.

### Terracing

Terracing is an additional feature that was added to this project to produce different effects on the terrain as shown in figure 17. This effect does not create any new terrain by itself but updates the current terrain to produce a terracing effect similar to the landscape found at the Banaue Rice Terraces. This effect was done by rounding the current height value and dividing by a user defined value to create different terracing effects.

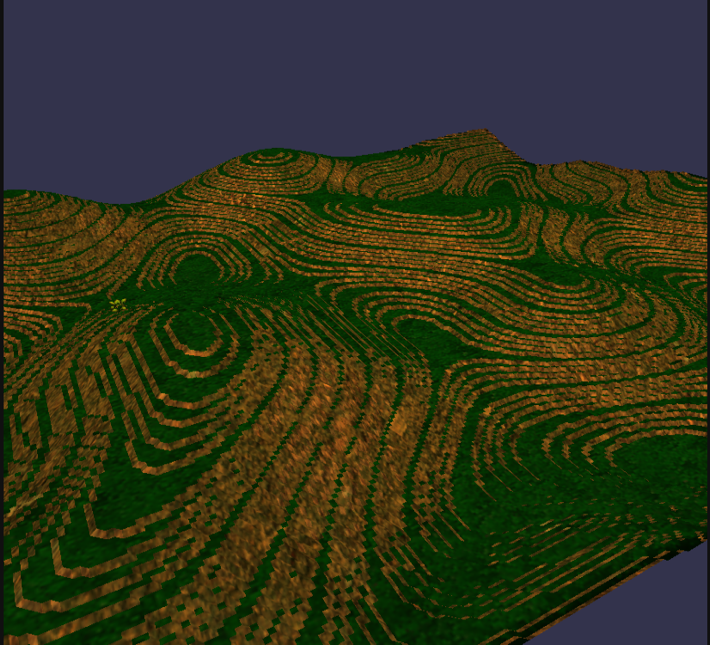


Figure - Terrain altered by the terracing function

for (int x = 0; x <= SizeOfTerrain; ++x) // loop through the x

{

for (int z = 0; z <= SizeOfTerrain; ++z) //loop through the z

{

//Get the current height value

float MapValue = HeightMap[x][z];

//Round this height value

float roundedValue = round(MapValue);

//and then set the height to the rounded value divided by the multiplier

HeightMap[x][z] = roundedValue / terracingMultiplier;

}

}

Listing - [TerrainGenerationScene.cpp] Terracing function

## Summary

Various algorithms and shaders were implemented to enable the procedural generation of terrain within this project. Through this section, some algorithms were replaced and the problems that arose when implementing them were discussed with the improvements that were made to fix these issues being considered and the reason for how they fixed the problems.

# Test Strategy and Evaluation

## Introduction

## Functional Testing

## Non-Functional Testing

Content goes here.

## User Testing

Content goes here.

## Evaluation

Content goes here.

## Summary

# Evaluation, Conclusions and Future Work

## Project Objectives

## Self-Evaluation

## Project Evaluation

## Applicability of Findings to the Commercial World

## Conclusions

## Future Work

# References

Archer, T. (2011). Procedurally Generating Terrain. *Midwest Instruction and Computing Symposium* (pp. 1-15). MICS.

Entertainment Software Association. (2021, July). 2021 Essential Facts About the Video Game Industry . Retrieved from https://www.theesa.com/resource/2021-essential-facts-about-the-video-game-industry/

Erstu, E., Sell, J., & valli, S. (2017, May 27). Perlin Noise Generator.

Fornander, P. (2013). *Game Mechanics Integrated with a Lindenmayer System.* Karlskrona: Blekinge Institute of Technology.

Freiknecht, J. (2020). Procedural Content Generation for Games.

Ginting, A., Sari, K., Fadhilah, C., & Yusra, R. N. (2019). Application of the Perlin Noise Algorithm as a Track Generator in the Endless Runner Genre Game. *Journal of Physics Conference Series*.

Grant, M. (2012). My first Dwarf Fortress World. Retrieved from https://www.flickr.com/photos/mylesdgrant/7689242534

Green, D. (2016). *Procedural Content Generation for C++ Game Development.* Packet Publishing Ltd.

Gregory, J. (2009). *Game Engine Architecture.* Wellesley: A K Peters, Ltd.

Gustavson, S. (2005). *Simplex noise demystified.* Linkoping University.

Hendrikx, M., & Iosup, A. (2013). Procedural Content Generation for Games: A Survey. *ACM Transactions on Multimedia Computing Communications and Applications*.

Hendrikx, M., Meijer, S., Iosup, A., & Van Der Velden, J. (2013). *Procedural Content Generation for Games: A Survey.* ACM Trans. Multimedia Comput.

Korn, O., Blatz, M., Rees, A., Schaal, J., Schwind, V., & Gorlich, D. (2017). Procedural Content Generation for Game Props? A Study on the Effects on User Experience. *Computers in Entertainment*, 1-15.

Lagae, A., Lefebvre, S., Cook, R., DeRose, T., Drettakis, G., Ebert, D. S., . . . Zwicker, M. (2010). A survey of Procedural Noise Functions. *COMPUTER GRAPHICS forum*.

Lagae, A., Lefebvre, S., Cook, R., DeRose, T., Drettakis, G., Ebert, D. S., . . . Zwicker, M. (2010). A survey of Procedural Noise Functions. *COMPUTER GRAPHICS forum*, 1-20.

Losh, S. (2016, June 27). *Terrain Generation with Diamond Square*. Retrieved from Stevelosh.com: https://stevelosh.com/blog/2016/06/diamond-square/

Losh, S. (2016, February 19). *Terrain Generation with Midpoint Displacement*. Retrieved from Steve Losh: https://stevelosh.com/blog/2016/02/midpoint-displacement/

Lucid Content Team. (n.d.). *The pros and cons of Waterfall Methodology*. Retrieved from Lucid Chart: https://www.lucidchart.com/blog/pros-and-cons-of-waterfall-methodology

M. F. Viana, B., & R. dos Santos, S. (2019). A Survey of Procedural Dungeon Generation. *18th Brazilian Symposium on Computer Games and Digital Entertainment*, pp. 29-38.

Mijailovic, V. (2015). *A Graph-Based Approach to Procedural Terrain.* KTH Royal Institute of Technology.

Miller, G. S. (1986). The Definition and Rendering of Terrain Maps. *ACM SIGGRAPH Computer Graphics*, 39-41.

Moss, R. (2016, January 01). *7 uses of procedural generation that all developers should study*. Retrieved from Gamasutra: https://www.gamasutra.com/view/news/262869/7\_uses\_of\_procedural\_generation\_that\_all\_developers\_should\_study.php

Perlin, K. (2001). *United States Patent No. US6867776B2.* Retrieved from https://patents.google.com/patent/US6867776

Snook, G. (2003). *Real-time 3D terrain engines using C++ and DirectX 0 (1st ed.).* Charles River Media.

van der Linden, R., Lopes, R., & Bidarra, R. (2014). Procedural Generation of Dungeons. *IEEE Transactions of Computational Intelligence and AI in Games*, 78-89.

# Appendix 1 – Project Proposal

## Project Context

This project intends to create a game engine with the ability to procedurally generate (procgen) terrain using parameters given by the user. Weather effects and a day night cycle will be attempted to implement into the game engine. The benefits of undertaking this project will be a further understanding of how game engines are created and furthermore, game prototypes are created. Another benefit would be further learning of how graphics are implemented into game engines.

Procgen has been used to many different effects in video games, *Crusader Kings 2* has used it to create rich and detailed family trees dynamically and *Shadow of Mordor* has used procgen to create new and exciting enemies that would remember the player if they were injured by them (Moss, 2016). Procgen has also been used for generating new terrain in many games as this is its main purpose, one of the best examples of this is Sid Meier’s *Civilization* series (Moss, 2016).

A few researchers have gone through the many different techniques of generating noise and the different categories while discussing their benefits and cons (Lagae, et al., A survey of Procedural Noise Functions, 2010). The algorithm I chose to implement is the Perlin noise algorithm, a terrain smoothing technique to overcome a rough field (Ginting, Sari, Fadhilah, & Yusra, 2019), to generate the values required to set the locations of the textures in the scene. The workflow to be used/implemented for generating perlin noise is described in Figure 3.1 in Erstu’s Perlin Noise Generator (Erstu, Sell, & valli, 2017, p. 5).

## Specific Objectives

* Basic Game Engine.
* Procedurally generated terrain based on user entered parameters.
* Different weather effects

## Potential Ethical or Legal Issues

If this project were planned to be sold commercially, one legal issue that would arise is copyright status of this project and whether end users would be able to use the engine to create their own piece of works, the solution to this problem is to decide how I would want the product to be licensed and whether I will allow end users to create commercial products with this project. An ethical issue that might occur with this project is the type of content that can be made with engine by the end user. An example would be content made that is too inappropriate for certain communities. This will not be an issue for this project as the content that can be made from the engine will be restricted to terrain and weather effects.

Another legal issue that can occur is whether I will be able to use the Perlin Noise Algorithm in this project, for example, whether I will need to pay a certain fee for the use of the Perlin Noise Algorithm. This looks like it is most likely not the case and that I will be able to use this algorithm within my project. A final issue that I have found that could arise is the ethical issue of prohibiting users to use my project - if it is to be sold - through not allowing accessibility features.

## Resources

* DirectX 11
* Visual Studio 19
* Programming Language: C++, HLSL
* Github or bitbucket (version control)

## Potential Commercial Considerations - Estimated costs and benefits

The estimated time to complete this project is about 4 months of work, there aren’t many costs for this project as this will be done individually and not to be sold. If this was to be sold commercially, the main cost of this project will be the workforce required to complete this project. The average monthly salary for a programmer in the UK is £4,800, therefore the calculated cost for the working hours of one programmer for this project is £19,200. This project would be planned to be developed to support a company, providing them with tools to be able to develop different game worlds/levels for different scenarios. This project could also be altered to affect different areas/systems in a video game.

## Proposed Approach

The first step would be to try and create a 3D game engine, that will focus on the generation of terrain, this will take the longest of all the tasks required as this is the biggest task to complete. The engine will need to be intuitive for the user to use and will provide the user with parameters to control the terrain generation. I estimate that this task will take about 1-2 months of work. The next task will be to create a perlin noise algorithm, commonly used for texture generation in games (Hendrikx & Iosup, Procedural Content Generation for Games: A Survey, 2013).This will take an estimated 1-2 weeks of work. Another task will be to create the procedural generation algorithm to generate the terrain using the game engine, I estimate that this will take about a month of work to complete. The final task will be to add weather effects and/or a day/night cycle, this task is optional and will be attempted if the three other tasks are completed. I estimate that this additional task will take a month to complete. The main tools that will be used is visual studio community 2019 and DirectX 11.

# References

Erstu, E., Sell, J., & valli, S. (2017, May 27). Perlin Noise Generator.

Ginting, A., Sari, K., Fadhilah, C., & Yusra, R. N. (2019). Application of the Perlin Noise Algorithm as a Track Generator in the Endless Runner Genre Game. *Journal of Physics Conference Series*.

Gregory, J. (2009). *Game Engine Architecture.* Wellesley: A K Peters, Ltd.

Hendrikx, M., & Iosup, A. (2013). Procedural Content Generation for Games: A Survey. *ACM Transactions on Multimedia Computing Communications and Applications*.

Lagae, A., Lefebvre, S., Cook, R., DeRose, T., Drettakis, G., Ebert, D. S., . . . Zwicker, M. (2010). A survey of Procedural Noise Functions. *COMPUTER GRAPHICS forum*, 1-20.

Moss, R. (2016, January 01). *7 uses of procedural generation that all developers should study*. Retrieved from Gamasutra: https://www.gamasutra.com/view/news/262869/7\_uses\_of\_procedural\_generation\_that\_all\_developers\_should\_study.php

# Appendix 2 – Technical Plan

## Title

Procedurally generated terrain using a game engine.

## Summary

This project intends to create a game engine with the ability to procedurally generate (procgen) terrain using parameters given by the user. Weather effects and a day night cycle will be attempted to be implemented into the game engine if everything else has been implemented properly. The benefits of undertaking this project will be a further understanding of how game engines are created and furthermore, prototypes. Another benefit would be further learning of how graphics are implemented into game engines. I will be using to agile methodology while working on this project as it allows for flexibility when obstacles occur during the development process.

## Deliverables

I will be submitting a project solution with all the source files. Another artefact that will submitted is the demonstration video, demonstrating the key features of my project. The final artefact to be submitted is the project executable to allow the marker/users to run the project themselves to understand how it works.

## Constraints

A major constraint on this project is the time management aspect of the project timeline in relation to the fixed deadlines, as well as having several months to complete the project. A way to overcome this constraint is to set up a schedule to follow and to stick to it as much as possible. A method that will be used to set up and follow this schedule is a Kanban board, the one that I will use is Trello®, as this has many features that will enable me to stick to the schedule that I have set up.

## Key Problems

One problem that needs to be overcome is the designing of an intuitive user interface to allow the user to easily understand what each control does to the program. This problem will be overcome by the implementation of the imGUI library as this has been specifically designed for user interfaces and has a free software license.

Another problem is the efficient implementation of the Perlin Noise algorithm within my project. This is the main problem to overcome within my project as this algorithm is the key to my project succeeding. However, I do not think that this will be a hard problem to overcome.

The third problem to this project is the creation of a 3D graphical environment, luckily, I was taught last year how to use DirectX11 as the graphics library to do this and have based this project off of the ‘skinning’ lab project from Year 2. This lab will be used for now until I fully understand how to setup a DirectX11 environment to the specifications that will be required.

## System and Work Outline

There are several key requirements within this project: the creation of a very basic game engine type program that will be able to procedurally generate terrain based off of the user input, an intuitive user interface, and the algorithm that will be used to generate this terrain. I will be using visual studio 2019 as my development environment with DirectX, imGUI, TinyXML and potentially assimp as external libraries to help me develop this project. I have checked the licenses for these libraries and have included the necessary licenses within my project.

I will begin development of this project by building the outline of the game engine and including the basic needs of the program. After this is done, I will be implementing the Perlin noise algorithm in my project to generate coordinates for the terrain that will be created. Once this is finished the finishing touches of the program will be done by designing and implementing the User Interface to allow the user to perform the function of the project. If I have enough time after this, I will be implementing the Midpoint Displacement algorithm as an additional option for the user, to understand how the different algorithms generate terrain. Doing it in this order will allow to me to deal with any problems/errors that occur, quickly since I will be doing everything in parts, and this will also allow me to adapt to any further changes required in my project.

## **A picture containing timeline Description automatically generated**Project Activities

## Risk Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Risk | Severity | Likelihood | Action |
| Lose of access to resources required | Extreme | Very Low | Restart Project with another idea. |

## Options

Lifecycles:

One of the first Lifecycle options that I considered was the waterfall method because it is the easiest to understand and work with. This methodology has several advantages such as having a clear structure to follow, and the end goal is determined early on within the project. However, I have chosen not to use this methodology due to the fact that it is not adaptable to new problems that can arise during the lifecycle of the project and that it does not permit testing until the project has been completed.

The second lifecycle option that I considered was the agile methodology. This was considered because of how popular it is among software developers in the industry. This popularity is warranted due to the several advantages this lifecycle provides. One of the big advantages, is how flexible it is to changes/problems that occur during the development lifecycle as the methodology works by breaking the lifecycle into 4-week segments/scrums and at the end of every scrum a review is done to see if anything new needs to be worked on. One disadvantage of this methodology is that there is no clear end goal to work towards as these 4-week scrums could go on indefinitely. The reason I chose to use this methodology is due to the flexibility as discussed earlier.

Development Tools:

There were two options I considered using for providing the graphics libraries to my engine, OpenGL and DirectX 11. I will go through there pros and cons before deciding which one to go with.

OpenGl was considered because it is easily portable to other platforms such as web browsers. This would make it easy to provide my engine’s services to more customers. However, there are several disadvantages to using OpenGl within my project, one the of the big disadvantages is the lack of proper documentation that is readable without confusing the reader. This will not be workable for me as I would be learning OpenGL from scratch and this would delay the project timeline significantly. Another disadvantage is that the implementations of OpenGL vary between graphic card manufacturers, such as Nvidia and AMD.

DirectX 11 was considered because of my experience using it during Year 2, I found directX 11 to be powerfull and understandable during this time. DirectX 11 is very similar to OpenGL except that the implementation doesn’t change between graphics and the documentation being easier to understand. I have chosen to use DirectX 11 in my project.

Algorithms:

There were two algorithms that were being considered for this project, Perlin Noise and Midpoint Displacement. The midpoint displacement algorithm is efficient for the generation of height maps used in terrain generation, However I have chosen not to use this algorithm as my main algorithm, instead opting to add it as an option to the user, allowing them to see the difference between the two algorithms. Losh (2016) goes into detail on how this particular algorithm works, however he is using Wisp to demonstrate the code used, therefore I will be using his work to understand the theory behind the algorithm.

The Perlin Noise algorithm has been chosen as the main algorithm used, because there are more sources available online to help me understand how it works and it generally generates smoother terrain.

## Potential Ethical or Legal Issues

If this project were planned to be sold commercially, one legal issue that would arise is copyright status of this project and whether end users would be able to use the engine to create their own piece of works, the solution to this problem is to decide how I would want the product to be licensed and whether I will allow end users to create commercial products with this project. An ethical issue that might occur with this project is the type of content that can be made with engine by the end user. An example would be content made that is too inappropriate for certain communities. This will not be an issue for this project as the content that can be made from the engine will be restricted to terrain and weather effects.

Another legal issue that can occur is whether I will be able to use the Perlin Noise Algorithm in this project, for example, whether I will need to pay a certain fee for the use of the Perlin Noise Algorithm. This looks like it is not the case and that I will be able to use this algorithm within my project without having to pay anything and will not be required to produce a license. A final issue that I have found that could arise is the ethical issue of prohibiting users to use my project - if it is to be sold - through not allowing accessibility features.

## Commercial Analysis

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Factor name | Description | Is this a cost or a benefit | Estimated Amount | Estimate of when paid |
| Salary | The average salary for a software developer in England | Benefit | £4,800 | Monthly until the end of the project |
| Price | The price of the product if sold commercially | Benefit | £300 | After project, if customer can be found. |

The estimated time to complete this project is about 4 months of work. There are not many costs for this project as this will be done individually and with the intention of the project to not be sold. If this was to be sold commercially, the main cost of this project will be the workforce required to complete this project. The average monthly salary for a programmer in the UK is £4,800, therefore the calculated cost for the working hours of one programmer for this project is £19,200. This project would be planned to be developed to support a company, providing them with the tools to be able to develop different game worlds/levels for different scenarios. This project could also be altered to affect different areas/systems in a video game. i.e. Weather systems, loot found in chests. if this project was done commercially, I would package it as its own product and propose its use to several game companies, providing it as a package that could be implemented within their other game engines.

## Employability Contribution

This project will improve my technical skills since I am working on a back-end supportive engine that will enable computer games to have the ability to create new terrain on the spot with different parameters. I have needed to research the Perlin Noise algorithm in order to create this project, giving me the knowledge to understand how the algorithm works. I can now use this knowledge to more easily understand how other algorithms are designed and meant to function. By going through with this project, my understanding of programming techniques has improved significantly as I worked through the problems that arose. I would be able to bring these techniques to where I will end up working.

## References

Archer, T. (2011). Procedurally Generating Terrain. *Midwest Instruction and Computing Symposium* (pp. 1-15). MICS.

Entertainment Software Association. (2021, July). 2021 Essential Facts About the Video Game Industry . Retrieved from https://www.theesa.com/resource/2021-essential-facts-about-the-video-game-industry/

Erstu, E., Sell, J., & valli, S. (2017, May 27). Perlin Noise Generator.

Fornander, P. (2013). *Game Mechanics Integrated with a Lindenmayer System.* Karlskrona: Blekinge Institute of Technology.

Freiknecht, J. (2020). Procedural Content Generation for Games.

Ginting, A., Sari, K., Fadhilah, C., & Yusra, R. N. (2019). Application of the Perlin Noise Algorithm as a Track Generator in the Endless Runner Genre Game. *Journal of Physics Conference Series*.

Grant, M. (2012). My first Dwarf Fortress World. Retrieved from https://www.flickr.com/photos/mylesdgrant/7689242534

Green, D. (2016). *Procedural Content Generation for C++ Game Development.* Packet Publishing Ltd.

Gregory, J. (2009). *Game Engine Architecture.* Wellesley: A K Peters, Ltd.

Gustavson, S. (2005). *Simplex noise demystified.* Linkoping University.

Hendrikx, M., & Iosup, A. (2013). Procedural Content Generation for Games: A Survey. *ACM Transactions on Multimedia Computing Communications and Applications*.

Hendrikx, M., Meijer, S., Iosup, A., & Van Der Velden, J. (2013). *Procedural Content Generation for Games: A Survey.* ACM Trans. Multimedia Comput.

Korn, O., Blatz, M., Rees, A., Schaal, J., Schwind, V., & Gorlich, D. (2017). Procedural Content Generation for Game Props? A Study on the Effects on User Experience. *Computers in Entertainment*, 1-15.

Lagae, A., Lefebvre, S., Cook, R., DeRose, T., Drettakis, G., Ebert, D. S., . . . Zwicker, M. (2010). A survey of Procedural Noise Functions. *COMPUTER GRAPHICS forum*.

Lagae, A., Lefebvre, S., Cook, R., DeRose, T., Drettakis, G., Ebert, D. S., . . . Zwicker, M. (2010). A survey of Procedural Noise Functions. *COMPUTER GRAPHICS forum*, 1-20.

Losh, S. (2016, June 27). *Terrain Generation with Diamond Square*. Retrieved from Stevelosh.com: https://stevelosh.com/blog/2016/06/diamond-square/

Losh, S. (2016, February 19). *Terrain Generation with Midpoint Displacement*. Retrieved from Steve Losh: https://stevelosh.com/blog/2016/02/midpoint-displacement/

Lucid Content Team. (n.d.). *The pros and cons of Waterfall Methodology*. Retrieved from Lucid Chart: https://www.lucidchart.com/blog/pros-and-cons-of-waterfall-methodology

M. F. Viana, B., & R. dos Santos, S. (2019). A Survey of Procedural Dungeon Generation. *18th Brazilian Symposium on Computer Games and Digital Entertainment*, pp. 29-38.

Mijailovic, V. (2015). *A Graph-Based Approach to Procedural Terrain.* KTH Royal Institute of Technology.

Miller, G. S. (1986). The Definition and Rendering of Terrain Maps. *ACM SIGGRAPH Computer Graphics*, 39-41.

Moss, R. (2016, January 01). *7 uses of procedural generation that all developers should study*. Retrieved from Gamasutra: https://www.gamasutra.com/view/news/262869/7\_uses\_of\_procedural\_generation\_that\_all\_developers\_should\_study.php

Perlin, K. (2001). *United States Patent No. US6867776B2.* Retrieved from https://patents.google.com/patent/US6867776

Snook, G. (2003). *Real-time 3D terrain engines using C++ and DirectX 0 (1st ed.).* Charles River Media.

van der Linden, R., Lopes, R., & Bidarra, R. (2014). Procedural Generation of Dungeons. *IEEE Transactions of Computational Intelligence and AI in Games*, 78-89.

# Appendix 3 – Title of Appendix