|  |
| --- |
| An Evaluation of the Feasibility of Using Procedural Generation in the Creation of Unique Levels for Mobile Games  Lee Elliott  BSc (Hons) Computer Game Applications Development, 2020 |

|  |
| --- |
| School of Design and Informatics  Abertay University |

**Table of Contents**

[Table of Figures iii](#_Toc38563498)

[Table of Tables iv](#_Toc38563499)

[Acknowledgements v](#_Toc38563500)

[1.0. Abstract i](#_Toc38563501)

[1.1. Context i](#_Toc38563502)

[1.2. Aims i](#_Toc38563503)

[1.3. Method i](#_Toc38563504)

[1.4. Results i](#_Toc38563505)

[1.5. Conclusions i](#_Toc38563506)

[Abbreviations, Symbols and Notation ii](#_Toc38563507)

[Chapter 1 – Introduction 1](#_Toc38563508)

[1.1. Project Relevance 1](#_Toc38563509)

[1.2. Project Focus 3](#_Toc38563510)

[1.3. Research Question 3](#_Toc38563511)

[1.4. Aim 3](#_Toc38563512)

[1.5. Objectives 3](#_Toc38563513)

[Chapter 2 – Background 1](#_Toc38563514)

[2.1. Procedural Generation 1](#_Toc38563515)

[2.2. Procedural Texturing 3](#_Toc38563516)

[2.3. Implementation on a Mobile Platform 4](#_Toc38563517)

[2.4 Survey Design 4](#_Toc38563518)

[2.5. User Testing 4](#_Toc38563519)

[Chapter 3 – Method 1](#_Toc38563520)

[3.1. Development 1](#_Toc38563521)

[3.1.1. Perlin Noise 2](#_Toc38563522)

[3.1.2. Random Walk 2](#_Toc38563523)

[3.1.3. Object Placement 3](#_Toc38563524)

[3.1.3.1. Boundary Marking 3](#_Toc38563525)

[3.1.3.2. Small Object Generation 3](#_Toc38563526)

[3.1.3.3. Large Object Generation 4](#_Toc38563527)

[3.1.3.4. Enemy Object Generation 4](#_Toc38563528)

[3.1.4. Combination and Water Level 5](#_Toc38563529)

[3.2. Performance Testing 5](#_Toc38563530)

[3.2.1. Quantitative 5](#_Toc38563531)

[3.2.2. Qualitative 5](#_Toc38563532)

[Chapter 4 – Results 6](#_Toc38563533)

[4.1. Quantitative 6](#_Toc38563534)

[4.2. Qualitative 10](#_Toc38563535)

[4.2.1. Method 10](#_Toc38563536)

[4.2.2. Survey Results 11](#_Toc38563537)

[4.2.2.1. Likert Scale Questions 11](#_Toc38563538)

[4.2.2.2. Additional Comments 15](#_Toc38563539)

[4.2.3. Evaluation of Qualitative Study 17](#_Toc38563540)

[Chapter 5 – Conclusions 19](#_Toc38563541)

[Chapter 6 – Future Work 22](#_Toc38563542)

[List of References 25](#_Toc38563543)

[Bibliography 27](#_Toc38563544)

[Appendices 30](#_Toc38563545)

[Appendix 1 - User Testing Survey 30](#_Toc38563546)

# Table of Figures

|  |  |  |
| --- | --- | --- |
| Figure | Name | Page Number |
| Figure 1 | Diablo III Nephalem Rift Floor Scene Variation | 1 |
| Figure 2 | Diablo III Nephalem Rift Floor Map Variation | 2 |
| Figure 3 | Graphical Comparison of Performance Data | 8 |
| Figure 4 | Comparison of Calculated Mean, Mode and Median Values for Windows and Android Implementations | 9 |
| Figure 5 | Graph of Responses to Likert Scale Questions | 12 |
| Figure 6 | Graph of Calculated Tester Response Mean Value | 13 |
| Figure 7 | Comparison of Calculated Mean, Mode and Median Values. | 14 |

# Table of Tables

|  |  |  |
| --- | --- | --- |
| Table | Name | Page Number |
| Table 1 | Windows Procedural Generation Performance Data | 6 |
| Table 2 | Android Procedural Generation Performance Data | 7 |
| Table 3 | Calculated Percentage Difference in Performance Times | 7 |
| Table 4 | Responses to Likert Scale Questions | 11 |

# Acknowledgements

Special thanks to my supervisor, Andrei Boiko, for all of the advice and guidance provided during the course of this research. Also to my family, for all of the support along the way, this would not have been possible without you all.

# 

# 1.0. Abstract

# 1.1. Context

In the infancy of video games, procedural generation was common practice because it saved developers the time and money spent designing levels and also due to the hardware constraints at that time. As technology has improved, the need for this has waned but there has been a resurgence lately, for the creation of an almost infinite number of random or pseudo-random outputs.

# 1.2. Aims

To evaluate the feasibility of using a procedural generation framework for the creation of seemingly random 3D levels for games, including the terrain and objects.

# 1.3. Method

Currently used methods of procedural generation were examined to find the most viable option for this project. These methods included but, were not limited to, Cellular Automata, Generative Grammars and Genetic Algorithms. A prototype was ultimately created using a combination of Perlin Noise and a random walk function, which generates terrain and level objects at runtime.

# 1.4. Results

The results produced during the quantitative testing performed upon this algorithm depict an average computational time of 91ms, with spikes of up to 300ms. As this generation process would be performed during the level loading phase of gameplay, a generation time of less than half a second indicates that this method would be viable. The qualitative study that was conducted for this application uncovered several areas in need of improvement, although the study also determined that the application performed favourably overall.

# 1.5. Conclusions

Overall, this project successfully ascertains that it is feasible to implement a procedural generation framework for the purpose of creating entire levels, including the terrain and population, for mobile games. The prototype created does not include full functionality, but it is successful at generating unique world space at run time for 3D mobile games.

# Abbreviations, Symbols and Notation

PG Procedural Generation

PCG Procedural Content Generation

# Chapter 1 – Introduction

## 1.1. Project Relevance

The use of procedural generation is popular with developers that are attempting to randomise aspects of a game, for which it is widely used within rogue like games. One of the earliest examples of procedural content generation in this genre dates back 1978, “Beneath Apple Manor”**(Scorpia 2015)** became the first to employ procedural content generation to create entire levels. More recently, using **PCG** to create unique 3D world space is used to great effect within Minecraft. The use of cubes to construct the world allows **PCG** to be used much more easily but prevents the creation of a realistic world.

Many modern games include an endless mode, which often has procedurally generated elements within a prebuilt environment. A prime example of this was included in an update to Diablo 3, where the player can enter “Nephalem Rifts” **(Johnson 2013)**. This game mode features randomised tile sets, layouts, lighting, weather, and enemies. The method of randomisation used in this game mode is effective in its attempt to create seemingly unique world space for each floor, each time a rift is started.

Figure – Diablo III Nephalem Rift Floor Scene Variation. An image displaying the vast variation within a single Nephalem Rift in Diablo 3. An image was captured from each of the nine floors of the same Rift to show the difference in the generated scenes (Regier 2012).

A picture containing photo, sitting, different, room

Description automatically generatedFigure – Diablo III Nephalem Rift Floor Map Variation. An image displaying the map of each of the nine floors generated for a single Nephalem Rift in the game Diablo 3, which shows the variation of the world space layout generated for each of these floors (Regier 2012)

As can be seen from the images in figures 1 and 2 above, the Nephalem Rifts featured within Diablo III consist of several floors, each of which are as unique as is possible. Each floor features a randomised layout as shown in the maps featured in figure 2. Each floor also features randomised tile sets, weather and lighting effects, which are shown in figure 1. The comparison image above does not adequately display that each floor contains randomised enemies, as the rift was beaten on floor two and this caused the enemies to no longer generate on the remaining floors.

The downside of many of the attempts made at creating an endless game mode is that the method used involves the use of a finite number of prebuilt areas, with the contents being generated at run time, which leads to playing in the same few world spaces repeatedly. This method of randomised levels is especially prevalent within mobile games. Procedurally generated world spaces are a potential solution to create infinite, unique level terrains, without bloating the game with prebuilt models. The mobile games industry in particular would benefit from the use of **PCG**, as storage is generally much more of an issue than with any other device types.

## 1.2. Project Focus

Procedural generation is already widely used for creating varied levels within 2D games, although it is much more common for levels to be pre-built models, with the content within the level being the only generated aspect. In instances of 3D random level generation, it is often produced by generating premade “rooms” in pseudo-random layouts. This project was focused on whether it is possible to procedurally generate entire levels for mobile games.

## 1.3. Research Question

How can procedural generation be used to create a wholly unique world space for use within 3D mobile games?

## 1.4. Aim

To evaluate the feasibility of using a procedural generation framework for the creation of an infinite number of unique 3D levels for games, including the terrain, population and textures.

## 1.5. Objectives

• To research and evaluate current procedural generation techniques

• To select, with reasoning, suitable techniques of procedural generation for each, terrain, population and objectives.

• To construct a working prototype to illustrate the key features

• To collate performance data relating to speed and aesthetic appeal of the prototype

• To assess the performance and viability of the prototype

• To evaluate the prototype and suggest improvements

# Chapter 2 – Background

In preparation for this project, an extensive literary review was carried out to determine suitable algorithms and frameworks that could be used in the development of a suitable procedural generation framework.

## 2.1. Procedural Generation

The article “Dive into the ‘educational’ world of PLATO”**(Moss 2016)**  was discovered. This article is an overview of early computer games, which focusses on the PLATO system. The article outlines the relatively undocumented era of computer games starting in the 1960s, of which very little was known due to the games being lost to time. In this article, it is stated that the earliest known precursor to the rogue-like genre was a game called pedit5. The information contained in this article indicated that it was possible to use procedural content generation techniques to create games to be run on very limited machines, much more limited than modern mobile phones.

In the conference paper “From Artistry to Automation: A Structured Methodology for Procedural Content Creation”**(Roden and Parberry 2004)**, the authors detail the various uses of procedural generation within games, the drawbacks of using handcrafted assets and how **PCG** can be utilised. Aside from the beneficial effects of using **PCG** upon the development time, there is the consideration of asset storage. Using **PCG** to create levels at run-time would reduce the number of pre-built levels, each of which requires a considerable amount of storage. A game with hundreds, or thousands of these would be much too large to be stored on a mobile device, which typically have between 32 and 128GB of storage, barring storage expansion. Although the latest mobile devices are being produced with up to 1TB of in-built storage, an ordinary plane model of dimensions 1000 vertices by 1000 vertices stored as an OBJ would require 117MB of storage space, without including normal data. While many of the concepts contained within Roden and Parberry’s paper are similar to those being proposed, the projects are ultimately very different.

Another online based textbook, “Procedural Content Generation in Games”**(Shaker, Togelius and Nelson 2016)** was discovered to provide a brief overview of many different methods of procedural content generation, without delving too deeply into explanations of the implementation of them. This source served to provide a list of avenues to research and explore. Also of note within this textbook, is an entire chapter dedicated to explaining the importance of evaluating a procedural content generation algorithm and many of the considerations that may need to be taken into account when doing so. This includes sections pertaining to both qualitative and quantitative testing and offers advice for optimizing the quality of the feedback.

In the journal article “A Proposal for a Procedural Terrain Modelling Framework”**(Smelik et al. 2008)**, also details the benefits of using **PCG** in content creation and focuses on the creation of a procedural generation framework. The proposed project differs from theirs in several ways, the most drastic being that the user is required to provide a terrain map, which is used to construct a 3D scene. The aim of this project was to create entire levels without the need for external input, instead using pseudo-random, or seeded, generation to produce the terrain for the level.

Another journal article “Procedural Generation of Dungeons”**(Linden, Lopes and Bidarra 2014)** discusses the many techniques used in the generation of entire levels for dungeon style games. This provides an in-depth knowledge of some of the techniques that may be pertinent to this project. The journal focusses heavily on the use of artificial intelligence in creating dungeon maps, but several of the concepts also seemed to be applicable to this project.

An online version of the textbook “The Book of Shaders”**(Gonzalez and Lowe n.d.)** was discovered to provide an in-depth examination of noise algorithms in roughly chronological order. This book was used to attain a good level of knowledge of the noise generation techniques that could potentially be used to complete this project. In particular, the description of the original Perlin Noise algorithm proved very useful in production of a value noise algorithm during the initial period of prototype development. This algorithm was implemented due to simplicity and possible speed but the information in this source indicates that Fractal Brownian Motion could produce a more aesthetically pleasing result, although this method may incur more computational work, resulting in performance considerations.

## 2.2. Procedural Texturing

The academic paper “GPU Based Algorithms for Terrain Texturing”**(Nicholson and Naicker 2008)** discusses several methods that can be used to apply appropriate textures to a procedurally generated terrain. Two of the methods covered in this paper are height based mapping and gradient based mapping. Alternative methods to the traditional tiling are also discussed in this paper, explaining that these methods avoid the obvious and unavoidable pattern that emerges when using traditional texture tiling. From the information in this paper, it becomes clear that a combination of several techniques should be used to produce the best results, with the base method being height based, with a gradient consideration used as a modifier. The paper also indicates that if stretching becomes apparent, then additional methods need to be incorporated to combat it.

Another article “Multi-textured Terrain in OpenGL”**(Jeremiah 2011)** was also discovered, which explains one possible method of implementing height-based texturing in OpenGL. This article serves as evidence that this procedure is possible using the OpenGL framework and provides a clear insight on one method of implementation.

## 2.3. Implementation on a Mobile Platform

Despite an extensive search, using various terminologies, there were no results similar to what is being proposed. There were multiple articles detailing attempts to design a procedural generation method for creating levels for the game “Angry Birds”, a 2D game **(Jiang et al. 2016; Stephenson and Renz 2016)**. The extent of these level generations extended only as far as generating the target structure contained within the level, which is significantly different to the project proposed in this document. From the research conducted in preparation for this project, it was concluded that there was a definite lack of research into the use of **PCG** to create pseudo-random levels at run time on mobile devices. The research also seemed to indicate that the implementation of a **PCG** framework for the creation of world space for mobile games was viable.

## 2.4 Survey Design

The academic paper ”Optimal Number of Response Categories in Rating Scales: Reliability, Validity, Discriminating Power, and Respondent Preferences”**(Preston and Colman 2000, p. 1–15)** discusses in depth the correlation between the number of points in a Likert style scale and the accuracy of the results. This paper suggests that the reliability of the results increases with the number of points until it starts to decrease again at around nine response options.

The academic paper “Multipoint Scales: Mean and Median Differences and Observed Significance Levels”**(Lewis 1993, p. 383–392)** indicates similar results and suggests that there is a greater clarity with the use of a seven point scale than with only a five point.

The general consensus across several academic papers on this subject is that a higher number of options in a Likert Scale can begin to cause confusion or disinterest in the participant. There is also a consensus that lower number of options decrease the accuracy of the results. The optimal number of options appears to lie somewhere between 5 and 9.

## 2.5. User Testing

The article “How Many Test Users in a Usability Study?” **(Nielsen 2012)** states that for a usability study, there is no benefit to be gained by enlisting more than five testers but also claims that for quantitative testing, twenty is the number of testers required for a reliable result.

# Chapter 3 – Method

## 3.1. Development

An initial prototype was created for the Windows platform using C++ within Visual Studio, making use of the OpenGL library. The prototype generates a terrain model, coordinates for object placement and procedurally textures the environment. First, the framework generates a height map by making use of a Perlin style noise algorithm. Next, a playable area is added, using a random walk algorithm. Using the data from this generated height map, the framework determines object locations, before applying the height data in the map to the creation of the terrain model. The data held within the map is also used to generate coordinates for the placement of representations of the objects in their generated locations.

The algorithms created for this initial prototype were then used in the creation of the native application prototype. This was again created using Visual Studio, by implementing the Native Application setup with OpenGL. This was intended to utilise the most recent, stable version of OpenGL es available, which, at present, is version 3.2. Due to unforeseen circumstances, this was not possible and version 1.0 was the only one available for this project.

Rudimentary control methods were then added to the application. The user is able to tap the screen in order to generate a world space, providing the user with as much time as they wish to analyse the result of the generation.

It was decided that a Perlin Noise based procedural generation algorithm should be implemented within this prototype, as this was one of the earliest methods used for **PCG** and was therefore most likely to be successful. It was also determined that the success of this method on the mobile platform would indicate that the newer methods could be attempted. In order to implement a Perlin based **PCG** framework, several algorithms were combined. Perlin noise was used to provide height data for the terrain, a path area was created using a random walk algorithm and object locations were generated based on the two “maps” created by these two algorithms.

### 3.1.1. Perlin Noise

Although not a true Perlin Noise, the algorithm was created to take in an array of floating-point values and modify them accordingly, using an array of permutation floats as a base value. This algorithm differs from true Perlin, as the permutation array used here is equal in size to the array of values used to map the terrain and is refilled every time the algorithm is run. The implemented algorithm also varies from a true Perlin Noise algorithm, as the permutation array is filled with pseudo-random values in the range 0 to 255, whereas in true Perlin, an array of 256 elements is filled with the values 0 to 255 in a pseudo random order.

### 3.1.2. Random Walk

This algorithm was created to take in an array of integer values to be modified, creates an array of a smaller scale of 64 by 64 to work upon and initializes every element to the value 1. The starting point of the “walk” is pre-set to the coordinates (31, 1) and the finishing conditions of the algorithm are that at least one third of the array has been marked with a zero and that the path must end adjacent to the “edge of map”. Until both of these conditions are met, the algorithm is designed to select one of the four adjacent grid-squares using a generated pseudo-random number and, if not already marked, mark it with a zero. Once the finishing conditions have been met, the algorithm applies these values to the large-scale array.

### 3.1.3. Object Placement

The object placement algorithm contains several distinct sub-algorithms for the generation of different types of object. This algorithm takes in the integer array filled by the random walk function to ensure that objects are placed in an appropriate location.

#### 3.1.3.1. Boundary Marking

The algorithm iterates through the array, first checking if the current value is equal to 1. If this check is passed, another check is performed, to see if any adjacent values are equal to zero. Finally, the current array element is checked to ensure that it is not an “edge” element, a check that prevents false positives around the edge of the map array.

It was observed that the boundary marking function produced a vast number of marked positions, the number of which could be limited to improve usability. An additional function was added to facilitate with this. This additional function takes in an integer that is used to set the space left between marked array elements. It then iterates through the array to check for marked elements. It then checks for concurrent marked horizontal elements, un-marking them unless they are either the last in the line or the space left has met the requirements. This process is repeated for concurrent vertical elements.

This algorithm was not enabled during testing, as it was deemed too difficult to represent with any clarity in the limited functionality available. This decision was also influenced by the consideration of this features effect on the time taken to render the scene.

#### 3.1.3.2. Small Object Generation

This portion of the algorithm takes in an integer counter variable to determine the number of small objects to be generated. It then loops until the number of generated objects matches the counter. The position coordinates are produced using pseudo-random integer generation, which are then checked to ensure that they fall within the playable area of the array before being marked.

#### 3.1.3.3. Large Object Generation

Designed for the purpose of generating large, decorative scenery objects within the world space. This was originally designed to take only the integer map array and number of objects required. First, the length and width of the objects are generated using pseudo random integer generation within a pre-set range. The algorithm then produces position coordinates using pseudo-random integer generation, which are then checked to ensure that the entirety of the object is out with the playable area.

This was later improved, to also account for the gradient of the generated terrain. This required the floating-point value array produced by the Perlin algorithm to be included in the parameters. In addition to the check to ensure that the entire object is out with the playable area, the new viability check ensures that the gradient of the terrain does not exceed 30 degrees at any point within the space that the object would occupy.

#### 3.1.3.4. Enemy Object Generation

This section of the algorithm takes in an integer counter variable to determine the number of small objects to be generated. It then loops until the number of generated objects matches the counter. The position coordinates are produced using pseudo-random integer generation, which are then checked to ensure that they fall within the playable area of the array and do not intersect a small object location, before being marked.

### 3.1.4. Combination and Water Level

During the combination of the Perlin and Walk algorithms, the average height of the terrain is calculated from the Perlin array, and a pre-set percentage of this height is applied to the playing area. The height is then reduced again, which is then applied to the plane that represents the water in the generated level.

## 3.2. Performance Testing

Performance testing was performed in two stages, the first was to determine the computation time of the algorithm and the second to ensure that the result of the algorithm was viable.

### 3.2.1. Quantitative

The computational time, in milliseconds, of the generation algorithm was collected during two hundred and fifty executions of the generation. This was also performed on the Windows version prototype. The average load time for each were calculated and compared. To make the comparison, the mean, the median and the range were all calculated from the collected performance data.

### 3.2.2. Qualitative

To evaluate the aesthetic quality of the generated terrain and ensure that a reasonable level of quality has been attained, a small group was enlisted to perform user testing and to give their opinion on the performance of the application. Each of these users was provided with the same questionnaire, comprising of mainly Likert scale questions, with a space provided for the user to also provide any additional comments. Due to research conducted into the production of a Likert Scale survey, a seven-point Likert Scale was used in the survey.

# Chapter 4 – Results

## 4.1. Quantitative

Performance times were recorded over a course of 250 generations using the Windows Implementation of the algorithm. The resulting times were then used to calculate the median, mean and range, to provide a clear indication of the performance of the algorithm at an early stage of the development process. As can be seen from this data, the computational time of the algorithm when implemented on Windows was, on average, 177 milliseconds, and even the spikes were under 450 milliseconds. Also noted from the collected data, the lowest time value recorded features three times in the data set, whereas the highest value appears only once.

Table 1 - Windows Procedural Generation Performance Data. Shows the recorded procedural generation performance times for the Windows implementation, with calculated median, mean, high, low and range.



Android Implementation performance times were also recorded over a course of 250 generations. These times were again used to calculate the median, mean and range. For this performance test, the generation was run using an emulated android phone with 3Gb of memory, running Android version 8.1. These settings for the emulated device were selected as they are typical for an upper midrange device of the last generation of Android devices.

Table 2 - Android Procedural Generation Performance Data. Shows the recorded procedural generation performance times for the Android implementation, with calculated median, mean, high, low and range.



The data from this round of testing provides an average computational time of the algorithm of 91 milliseconds, with spikes that were under 300 milliseconds. These times were an unexpected, drastic improvement on those recorded from the Windows implementation.

As a result of the comparison of these two data samples, it became apparent that the relationship between the datasets should be calculated in an easily quantifiable way. This was achieved by calculating the percentage difference between the Windows data and the Android data, which resulted in the following fields being added to the existing table, which shows that the average generation time is almost fifty percent faster than when performed in the windows environment. Another observation that can be made from this data is that the range of the data values is less than three quarters of those collected from the Windows implementation.

Table 3 – Calculated Percentage Difference in Performance Times. Shows the calculated percentage difference in calculated median, mean, high, low and range of the Windows and Android data sets

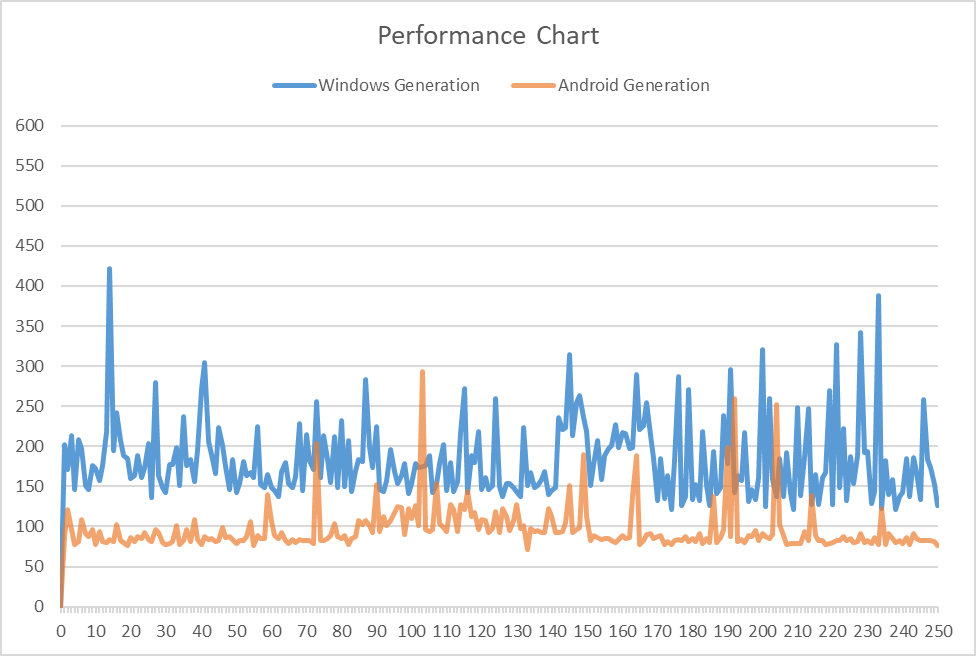
The graph shown in Figure 3 clearly shows that the mobile implementation of this procedural generation algorithm is viable. The graph also shows that the Android implementation is significantly faster and that the windows implementation suffers from a much wider variation, which indicates a much more reliable performance from the Android implementation, with fewer instances of spikes in the computational time taken.

Figure 3 – Graphical Comparison of Performance Data. A side by side comparison of the plotted performance data with blue representing the Windows value and orange representing the value for Android.

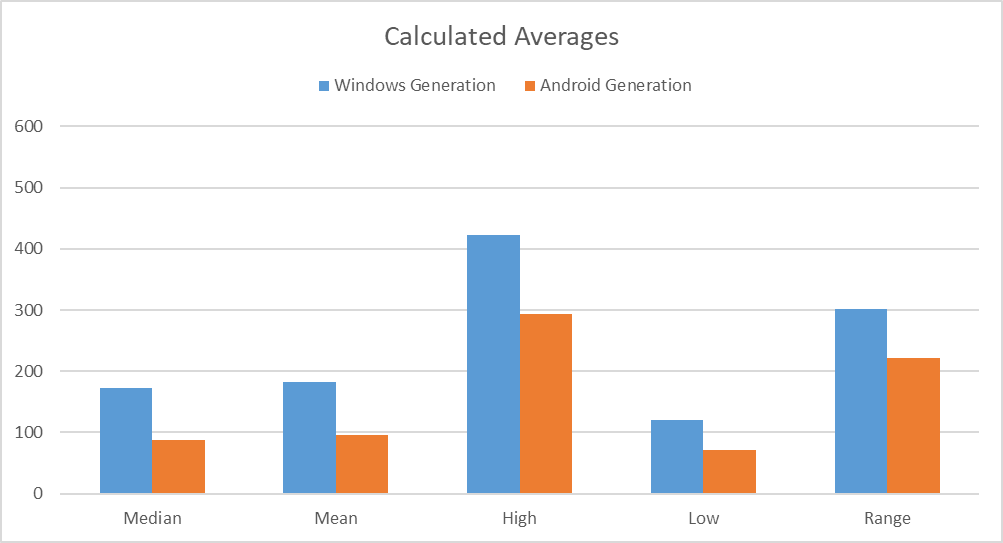
Another chart was created to display only the relationships between the calculated averages. This again provides a visual representation of the difference between the performance of both implementations. As can be clearly seen from this graph, both the mean and median values calculated for the Android implementation are significantly lower than even the lowest recorded time for the Windows implementation.

Figure 4 – Comparison of Calculated Mean, Mode and Median Values for Windows and Android Implementations. A side by side comparison of the calculated values, with blue representing the Windows value and orange representing the value for Android.

## 4.2. Qualitative

### 4.2.1. Method

It was originally decided that a group of no more than twenty would be employed for the task of testing this application due to the information learned while researching the optimal number for such a test, which indicated that any more than twenty would not be time effective. The final number of participants in the survey was twelve. One of the main contributing factors that caused a reduction in the number of participants in the qualitative study was the outbreak of a pandemic. This caused a nation-wide lockdown, which necessitated all participation in the survey to be contactless.

For this study, testers were selected with a range of backgrounds. It was deemed crucial to ensure that there were testers both with and without experience in each of the following criteria - playing games, playing mobile games and software development, as a group of testers with this range of experience would provide more insight into the aesthetic quality of the application. To meet these criteria, testers were selected from several Abertay University third- and fourth-year software development courses, including Computer Games Applications Development and Computer Games Technology. Other testers were selected, with no software development experience, from both within and out with Abertay University.

Although extremely difficult to source individuals with no experience of playing games, it was however possible to employ the services of individuals with very little experience. It is a similar situation when trying to source testers based on their experience with mobile games. If the study had been performed as originally planned, two additional questions would have been included within the questionnaire. These questions would have asked the participants to disclose their frequency of playing games and, more specifically, mobile games. With the reduced number of test participants, these questions would have been less likely to have produced results depicting any correlation. For this reason, the questions were omitted.

Each of the selected testers was provided with an identical survey featuring seven-point Likert Scale questions; ten questions in total. Each of these questions also provided an additional input box for optional, relevant comments. An additional comment space was also provided after these questions to ensure that the testers had ample opportunity to make their opinions known.

### 4.2.2. Survey Results

#### 4.2.2.1. Likert Scale Questions

The data gathered from the Likert Scale questions was analysed using several methods. First, an overall score was computed from each of the testers by calculating the mean value of the responses. This provided only the most basic of insight into the aesthetic proficiency of the application. Next, the mean, mode and median were calculated for each question across the entire survey. This served to provide a much more detailed analysis of how the application faired in each category. As an additional comparison, the mean and median of the calculated tester scores was calculated to provide a broad overview of the application’s performance in this user testing. Finally, graphs were plotted based on the received data, the overall scores and the computed averages to ease comparison and evaluation.

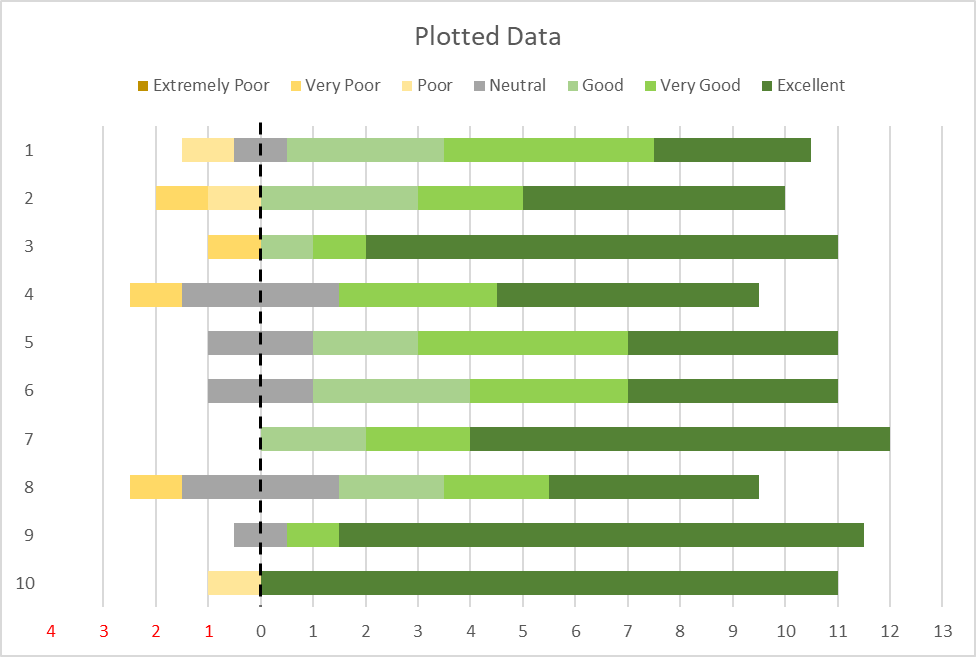
Table 4 - Responses to Likert Scale Questions. Shows the numerical values of the responses to the Likert Scale questions with calculated mean for each tester and average values calculated for each question.



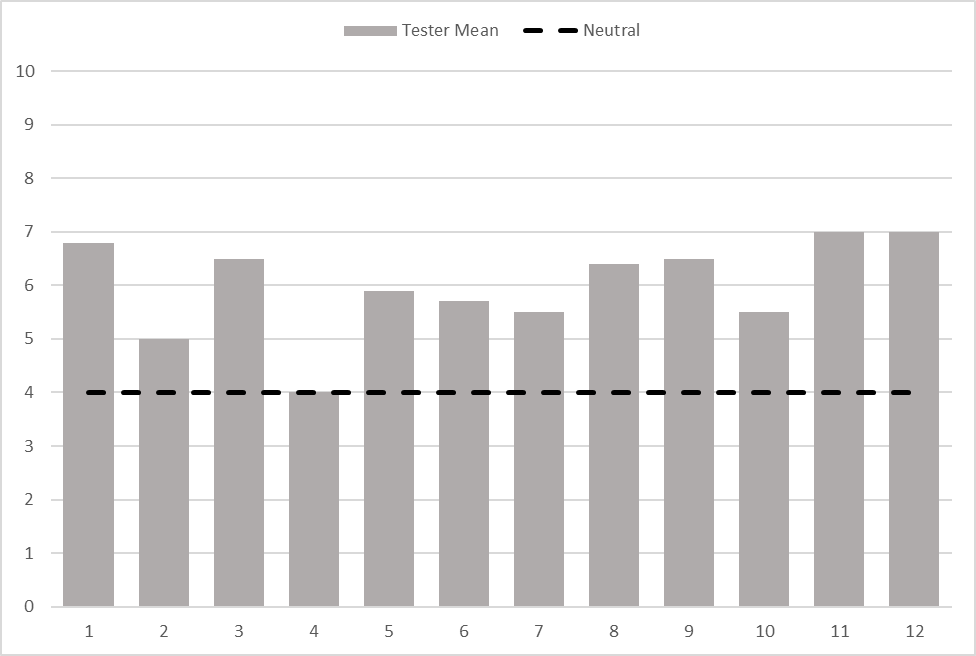
As can be seen in the data shown in Table 4, the responses to the Likert Scale questions received from the participants show vast variation, with no pattern easily discernible. The calculated averages appear much less chaotic and it is much easier to make generalised comparisons using these values. From this data, it is apparent that the algorithm was rated highest in both its speed and its reliability of generating objects at ground level. The algorithm was rated lowest for its ability to generate locations for small objects without them clipping.

To improve the ease of comparison and evaluation, the numerical responses to each question were reviewed to determine the frequency of each. These frequencies were plotted using a diverging stacked horizontal bar graph to produce a representation of the data that improved the clarity. Using a seven-point scale, meant that a response of four was neutral, less than four was unfavourable and greater than four was favourable. To improve the legibility of the graph, neutral was used as the centre point. As is clear from the graph shown in figure 4, the vast majority of the responses for all questions fell within the positive end of the spectrum. The final question of the survey asks the participant to rate the speed of the procedural generation algorithm. This question was rated at a seven by almost all of the participants, which seems to indicate that the improvements required to improve ratings in other areas could all be implemented without causing the algorithm speed to deteriorate too much.

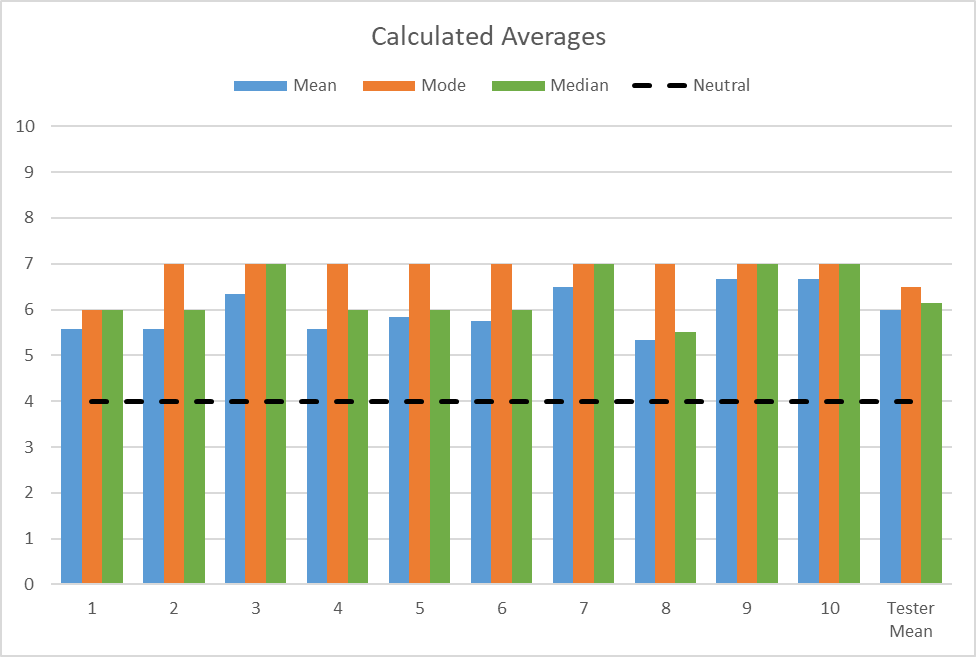
Figure 5 - Graph of Responses to Likert Scale Questions. A diverging stacked bar chart displaying the rating frequencies for each question in the survey, which is centered on neutral, with more negative feedback on the left and positive on the right. Also added, for additional clarity is a dashed line clearly marking neutral.



The calculated average for each tester was plotted in a standard column chart, shown in figure 4, which revealed that, although there is a wide variation between these computed averages, none of these mean values are lower than four, with just one that is four exactly. This means that the overall response from these testers lies on the positive side of the scale, indicating that the prototype is effective, yet needs improvement and refinement to reach its full potential.

Figure 6 - Graph of Calculated Tester Response Mean Value. A column chart depicting the calculated rating average for each participant. Also shown in the graph is a line depicting neutral.

A clustered column chart was used to display the results of the calculated averages. This was selected to simplify the comparison of these results and improve legibility. As can be seen from the graph in Figure 6, the responses to each of the questions were, on average, neutral or better, with questions nine and ten receiving the best response overall. This indicates that the algorithm’s speed was at least adequate and the generated locations for small objects and enemies were snapped to the terrain. From this graphical representation, it appears that question eight received lower responses, on average, than the other questions, which would seem to indicate that the placement of small objects appear to have clipping issues.

Figure 7 - Comparison of calculated mean, mode and median values of each question. Also shows the calculated mean, mode and median for the survey overall, which is calculated based on each user’s mean response. Also included is a marker denoting neutral on the Likert Scale.

#### 4.2.2.2. Additional Comments

The testers were provided with space for additional comments relating to each of the factors being assessed, as well as an additional space for non-specific comments, which was located at the bottom of the survey. An examination of a selection of the received comments follows.

1. It was noted that there was an obvious lack of textures in the generated scene, intimating that the prototype would have been improved with their inclusion.
2. It was also noted that there was an issue with the depth buffer, which made it more difficult for the tester to analyse positions of generated objects.
3. Enemy locations were observed to be very close together, appearing to overlap in one case. This should not cause an issue to the algorithm, as these locations each denote a group of enemies and close proximity would have the effect of combining the enemy groups, to form one larger group.
4. Enemy positions were also observed to be very close to, or overlapping with, small object locations. This would have the effect, in a game environment, of producing a guarded small object, which features frequently in games.
5. One survey mentions that an enemy was observed out with the path area on one occasion. Investigation required.
6. One survey mentions a long generation time, whilst blaming the device used to test. This indicates that there may be some devices that struggle to run this algorithm. Investigation required.
7. Possible lighting bug was reported. Lighting is disabled in the scene, meaning that this is more likely to be an issue with the colour selection method implemented. If the terrain is very close to the level of the path and does not have much of a change in height, it would appear similar to the submitted bug.
8. The inclusion of mountainous terrain was suggested. It is already possible for the algorithm to produce this style of terrain, although, it is possible that this was not observed within the test.
9. Clipping between the terrain and the building markers was noted. Unsure if this was a recurring issue or if it only occurred within certain circumstances.
10. Inclusion of a controllable camera was suggested to improve analysis of the generated scene.
11. Suggested expansion to include a top-down view that could be exported for use in table-top games.
12. It was mentioned that the render distance could have been increased to improve the ease of analysis.

## 4.2.3. Evaluation of Qualitative Study

The testers seemed to have difficulty analysing the generated scenes due to several factors, with the fixed camera rotation and lack of a controllable camera attributing to the difficulties. It also seemed that the colours chosen to represent features of the generated scene may have added to the confusion. The path colour is too similar to the colour chosen to represent land lower than path level, which may make it more likely to appear that buildings are generating on the path. In retrospect, the colour grey may have been a more suitable colour to represent the path.

The quantitative testing performed for this study utilised the participants own android device, resulting in several different brands and models being used, including Sony, Samsung, Amazon and Huawei devices Several prospective participants were ultimately unable to take part in the study due to a rendering issue discovered when the application is run on the Samsung Galaxy S9. On this device type, the scene renders as fully black, which effectively prevents analysis of the generated scene. A full investigation into the cause of this issue is required. It is, however, known that not all models of Samsung devices are affected by this issue, as another participant successfully ran the application on a Samsung Galaxy J3. An investigation to determine the cause of this issue and the affected devices should be undertaken.

The volume of constructive comments and information gained from this test was invaluable but, with the limited number of participants, it proved difficult to assess conflicting comments. To optimise the results in the event of retesting, twenty to thirty participants would be preferred. If possible, it would be optimal to perform the study with twenty to thirty participants representing each of the following, programmers, non-programmers, game players, mobile game players and non-game players. Including questions for programming and games experience in the survey, and employing such a group to participate, would, perhaps, provide the optimal data for analysis. Capturing the participants experience level in each of these areas within the survey would perhaps be beneficial in analysing the results of any further qualitative studies.

The questions used in the survey may have been too vague in the way they were worded. An effort was made to keep the wording of the questions short, but this may not have been the best idea. Question eight, in particular, asks the participant to review the generated locations for small objects, looking specifically at whether there is clipping with other objects. For each question, the specifics of what is acceptable and unacceptable should be explained to the participant. In an effort to keep the questions concise, it may be preferable to include these details of what to look for in the instructions provided to the participants. Information that should perhaps have been provided to the participants includes that clusters of enemy locations are acceptable in the algorithms result, that small object locations are permitted to be in close proximity to enemy locations, but also that overly condensed clusters of enemy locations and small object locations should not be observed.

# Chapter 5 – Conclusions

The implementation of OpenGL functionality of Android proved more difficult than initially anticipated. The main issue encountered was that, despite many weeks that were ultimately wasted trying to enable the more modern functionality of this graphics library, it was to no avail. The drawbacks of this lack of functionality severely limited the current prototype, as the alternate methods that were selected produced their own issues, which had to be overcome. In some instances, however, alternative methods were not feasible, causing the affected features to be omitted from the prototype entirely, as is the case with per-vertex texturing.

Vertex Buffers were not accessible in the prototype, limiting the prototype to the use of vertex arrays, which have much stricter size limits. If the indices are passed in as unsigned shorts, the range of values able to be used as indices is 0 to 65,535, whereas if unsigned bytes are used, the range is 0 to 255. In the current implementation, the use of shorts was not working as expected, so the indices were passed as bytes. In order to implement this, the original design had to be modified. Instead of one large, subdivided plane, 512 vertices wide and 512 vertices long, it became necessary to create many smaller planes. The first attempt at this, was to create an array of simple, four vertex planes 511 wide and 511 long and to arrange these seamlessly. This method technically worked but the number of draw calls necessary to render this number of planes caused an unacceptable delay of over 15 seconds every frame, where the application appeared to freeze. It became clear from this that the number of draw calls must be reduced. To do this, slightly larger planes, 16 vertices wide and 16 vertices long, were devised. These smaller planes were arranged 34 planes wide and 34 planes long, with a one vertex overlap on the edges to mimic the effect of using a single large plane. The effect of this on the frame rate of the application was substantial, with an estimated fifteen to eighteen frames per second. Even with this improvement in the frame rate, there was still a noticeable jitter when the camera was moving, which indicated that further modifications were required to ensure that the application produces less choppiness. The use of vertex buffers appears to be the optimum solution for reducing the delay caused when rendering the scene.

In order to improve the frame rate still further, it was decided that only visible chunks should be drawn and that distant chunks should also be culled. This would have the effect of limiting the number of draw calls to a minimum. The application now draws only chunks that are in front of the viewpoint. The current configuration of this calculates the chunk that corresponds to the camera’s current position. The area of visible chunks is then determined as being twelve chunks either side on the X axis, fourteen chunks in front of the camera on the Z axis and two chunks behind the camera on the Z axis.

Due to the limitations present in the available OpenGL functionality, it was also not possible to implement a per-vertex texturing algorithm. This was due to a lack of access to the rendering pipeline. Without access to shaders, it is not possible to apply any texture blending techniques, which require access to the pixel shader. It was decided that it was extremely difficult to visually analyse the formation of the terrain as a flat colour, so height-based colouring was implemented, which is divided into three parts. First, any vertices with a height equal to the height of the path have their colour set to brown. Next, vertices with a height greater than that of the path, have a shade of green applied, which is calculated by multiplying RGB full green by the height value of the vertex. Finally, vertices with a height less than that of the path, have a colour shade modified from the brown of the path area. The difference between the chosen shade of brown and full yellow is multiplied by the difference between the height of the vertex and the height of the path and the resulting value is added to the RGB values. This creates a gradient effect, fading from the brown of the path to a pale yellow as the height decreases. It was noticed during testing, that the gradient of the area lower than the path never seemed to reach any form of yellow. The calculation used to determine the correct colour should be examined and amended.

The user controls are considerably more simplistic than originally planned. Due to time lost attempting to link libraries to the project, there simply was not enough time to implement control methods to the degree that was originally envisaged. The full extent of the control methods included within this prototype is that the application registers a tap event, at any point on the devices screen, which allows the user to control when the application will generate a new world space.

It was not possible to create a fully working camera using the available OpenGL functionality. Without full access to OpenGL, all that was available was to move the entire scene, while the view remained motionless. Initial attempts to replicate rotation of the camera were to no avail. Although this is possible, there was not sufficient time to commit to implementing this feature. In an attempt to ensure that the user is provided with an adequate view of the generated world space, the camera setup in this prototype is configured to give the appearance that it is flying over the terrain and back. The camera travels along the X axis, from one end of the world space to the other, at which point, Z axis movement occurs to move to a new row of chunks. This implementation produces an endless zig-zag style of movement, to allow the user a comprehensive view of the generation output.

The produced prototype does not include all of the functionality originally intended but it does successfully showcase a simple application that can generate and display a terrain, generate locations for and display markers for objects and determine a very simple water level for the world space. The application prototype produced seems to indicate that procedural generation is a feasible method for the creation of unique world space within mobile games. The computation times recorded indicated that the use of procedural generation for the production of entire levels for mobile games would incur comparable delays to the loading times incurred by games not currently using procedural content generation. This therefore indicates that such a system could be viable if the results of the algorithm are reliable.

The results of the qualitative study highlighted several areas in which the application is in need of improvement or fine tuning, but the overall response from the participants indicated that the prototype successfully demonstrates the use of a procedural generation framework to create entire world spaces for games on Android devices. The feedback received from the qualitative study that was conducted indicates that a fully implemented, polished framework would be capable of reliably generating viable, unique scenes. The recorded computational times, coupled with the user responses regarding the generation time indicates that a procedural generation framework would also be able to produce the world space within an acceptable time frame, comparable to an acceptable loading time.

# Chapter 6 – Future Work

During the development of this prototype, it was not possible to link additional libraries. This resulted in a reduction in the level of OpenGL functionality available and limited the features of the base application of the prototype. The use of vertex buffer objects is one of the features that was not available during the implementation, which made the instantiation of a single plane containing 262144 vertices impossible. The use of vertex buffer objects would allow the terrain object to utilise just one draw call, whereas, in the current setup of the prototype, there are 1156 draw calls when drawing the entire terrain, which has been limited to 272 draw calls by limiting the distance. This would have the effect of significantly reducing the time taken to render the scene.

It was not possible to implement per-vertex texturing in this prototype, as, without access to further functionality, OpenGL is configured with a fixed pipeline. Successful integration of the appropriate libraries would allow access to the programmable pipeline, after which, it would be a simple matter of enabling texture blending and applying the chosen textures to each vertex in the pixel shader, using the planned combination of height and gradient based calculation to select the correct texture.

The Perlin Style algorithm that was implemented differs significantly from the algorithm that Perlin designed. Implementing a true Perlin may be beneficial to the performance of the generation algorithm. This is not necessary with the current prototype but may become more imperative as the applications functionality is extended and the computational time increases.

The procedural generation method that was developed for this prototype was chosen as it was perhaps the simplest, and therefore the most computationally economic. Having determined the feasibility of this method, it would be logical to start testing alternate methods of increasing complexity. A Cellular Automata based algorithm, implemented with a solid set of rules, could possibly produce much more reliable results due to the fact that it is a rule-based algorithm. A well devised set of rules could allow the algorithm both the freedom needed for the creation of unique levels and the restrictions required to ensure a viable and aesthetically realistic output every time.

At present, the prototype is only able to produce one style of scene, which is of an exterior, natural landscape. With some modification, this could be extended to include algorithms designed to produce cave, dungeon and urban area levels. Each of these scene styles would require a totally different set of methods for generation. Rural and cave levels would benefit from more natural, curved pathways, whereas urban and dungeon levels would ideally feature more right-angled corners. Dungeon and cave levels would not require large scenic objects to be generated, though they would require high walls to be produced, whereas urban levels would require a higher volume of large scenic objects to be generated near the edge of the path.

The prototype is currently designed to generate locations for enemy placement, but not for generating the types or numbers of enemies at each generated site. A simple expansion to this would allow the algorithm to generate the number of enemies for each generated location and, at least, the base type of each of these enemies. Using similar methods, the algorithm could also be configured to generate quests for within the world space.

It would be beneficial to implement a user-controlled camera. This would allow the user the ability to visually inspect the generated world space with a much higher degree of accuracy. To implement this, it would perhaps be best to have movement on the X and Z axes be manipulated by use of on-screen buttons or a virtual thumb stick. Rotation could be implemented around just the Y axis, but it may be of more benefit to also include rotation around the X axis. The rotation controls would ideally be similar to those of movement. A zoom method could also be implemented to allow the user to view chosen aspects in even more detail.

Additional control methods may become necessary depending upon any prototype extension. An example of this would be that object selection via tapping would become necessary if the enemy generation or object generation methods are extended to produce additional data. This would also necessitate the addition of a button to be used to trigger the generation trigger instead of the current configuration, which accepts a tap anywhere on the screen as the trigger method for the generation algorithm.

The prototype would greatly benefit from the implementation of a user interface, with pertinent information displayed clearly for the user. The information from the current implementation, which could be displayed for the user includes the number of each type of generated object and the current coordinates of the camera. This user interface could also be extended to match any additional features, including information on selected objects and details of generated quests. Another user interface feature that could be very beneficial would be a mini map. This would provide the user with a clear view of the scene as a whole, improving the ease with which the generation result could be assessed. With the use of a marker to indicate the cameras current position, the user would also be provided with a simple method of visually identifying their current location within the scene, further increasing the ease of use.

# List of References

Gonzalez, P and Lowe, J n.d., The Book of Shaders, *The Book of Shaders*, viewed 25 September, 2019, <https://thebookofshaders.com/11/>.

Jeremiah 2011, Multi-textured Terrain in OpenGL, *3D Game Engine Programming*, viewed 14 February, 2020, <https://www.3dgep.com/multi-textured-terrain-in-opengl/>.

Jiang, Y, Kaidan, M, Chu, CY, Harada, T and Thawonmas, R 2016, Procedural Generation of Angry Birds Levels using Building Constructive Grammar with Chinese-Style and/or Japanese-Style Models, *arXiv:1604.07906 [cs]*.

Johnson, S (Lylirra) 2013, Reaper of SoulsTM First Look: Adventure Mode, *Diablo III*, viewed 8 October, 2019, <http://us.battle.net/d3/en/blog/11516518/reaper-of-souls%E2%84%A2-first-look-adventure-mode-11-8-2013>.

Lewis, J 1993, Multipoint Scales: Mean and Median Differences and Observed Significance Levels, *Int. J. Hum. Comput. Interaction*, 5, pp. 383–392.

Linden, R van der, Lopes, R and Bidarra, R 2014, Procedural Generation of Dungeons, *IEEE Transactions on Computational Intelligence and AI in Games*, 6, (1), pp. 78–89.

Moss, R 2016, Want to see gaming’s past and future? Dive into the ‘educational’ world of PLATO, *Ars Technica*, Technology Review, viewed 12 September, 2019, <https://arstechnica.com/gaming/2016/10/want-to-see-gamings-past-and-future-dive-into-the-educational-world-of-plato/>.

Nicholson, K and Naicker, A 2008, GPU Based Algorithms for Terrain Texturing.

Nielsen, J 2012, How Many Test Users in a Usability Study?, *Nielsen Norman Group*, viewed 23 March, 2020, <https://www.nngroup.com/articles/how-many-test-users/>.

Preston, C and Colman, A 2000, Optimal Number of Response Categories in Rating Scales: Reliability, Validity, Discriminating Power, and Respondent Preferences, *Acta psychologica*, 104, pp. 1–15.

Regier, J 2012, Diablo III, Blizzard Entertainment.

Roden, T and Parberry, I 2004, From Artistry to Automation: A Structured Methodology for Procedural Content Creation, in M Rauterberg (ed.), *Entertainment Computing – ICEC 2004*, Springer Berlin Heidelberg, pp. 151–156.

Scorpia 2015, 1978 – Beneath Apple Manor, *The CRPG Book Project*, viewed 6 October, 2019, <https://crpgbook.wordpress.com/review-index/1978-beneath-apple-manor/>.

Shaker, N, Togelius, J and Nelson, MJ 2016, About the book | Procedural Content Generation in Games, Book, viewed 25 September, 2019, <http://pcgbook.com/about-the-book/>.

Smelik, RM, Tutenel, T, de Kraker, KJ and Bidarra, R 2008, A Proposal for a Procedural Terrain Modelling Framework, p. 4.

Stephenson, M and Renz, J 2016, Procedural generation of complex stable structures for angry birds levels, *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 1–8.

# Bibliography

Amato, A 2017, Procedural Content Generation in the Game Industry, in O Korn & N Lee (eds), *Game Dynamics: Best Practices in Procedural and Dynamic Game Content Generation*, Springer International Publishing, Cham, pp. 15–25.

Anon n.d., OpenGL ES 2.0 Reference Pages, viewed 15 February, 2020, <<https://www.khronos.org/registry/OpenGL-Refpages/es2.0/>>.

Bouranta, N, Chitiris, L and Paravantis, J 2009, The relationship between internal and external service quality, *International Journal of Contemporary Hospitality Management*, 21, pp. 275–293.

Buttle, F 1996, SERVQUAL: review, critique, research agenda, *European Journal of Marketing*, 30, pp. 8–32.

Colman, A, Norris, C and Preston, C 1997, Comparing Rating Scales of Different Lengths: Equivalence of Scores From 5-Point and 7-Point Scales, *Psychological Reports*, 80.

Cornut, O 2019, Ocornut/imgui, C++, Ocornut.

Finstad, K 2009, Response Interpolation and Scale Sensitivity: Evidence Against 5-Point Scales, 5.

Gonzalez, P and Lowe, J n.d., The Book of Shaders, *The Book of Shaders*, viewed 25 September, 2019, <<https://thebookofshaders.com/11/>>.

Greggman 2019, Could ImGUI be the future of GUIs?, viewed 7 October, 2019, <<https://games.greggman.com/game/imgui-future/>>.

Hendrikx, M, Meijer, S, Van Der Velden, J and Losup, A n.d., Procedural content generation for games: A survey: ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM): Vol 9, No 1, viewed 15 February, 2020, <<https://dl.acm.org/doi/abs/10.1145/2422956.2422957>>.

Holden, G 2018, Review: Diablo III Eternal Collection (Switch), *SA Gamer*, viewed 7 October, 2019, <<https://sagamer.co.za/review-diablo-iii-eternal-collection-switch>>.

Iii, E 1980, The Optimal Number of Response Alternatives for a Scale: A Review, *Journal of Marketing Research*, 17, p. 407.

Jeremiah 2011, Multi-textured Terrain in OpenGL, *3D Game Engine Programming*, viewed 14 February, 2020, <<https://www.3dgep.com/multi-textured-terrain-in-opengl/>>.

Jiang, Y, Kaidan, M, Chu, CY, Harada, T and Thawonmas, R 2016, Procedural Generation of Angry Birds Levels using Building Constructive Grammar with Chinese-Style and/or Japanese-Style Models, *arXiv:1604.07906 [cs]*.

Johnson, S (Lylirra) 2013, Reaper of SoulsTM First Look: Adventure Mode, *Diablo III*, viewed 8 October, 2019, <<http://us.battle.net/d3/en/blog/11516518/reaper-of-souls%E2%84%A2-first-look-adventure-mode-11-8-2013>>.

Lewis, J 1993, Multipoint Scales: Mean and Median Differences and Observed Significance Levels, *Int. J. Hum. Comput. Interaction*, 5, pp. 383–392.

Linden, R van der, Lopes, R and Bidarra, R 2014, Procedural Generation of Dungeons, *IEEE Transactions on Computational Intelligence and AI in Games*, 6, (1), pp. 78–89.

Moss, R 2016, Want to see gaming’s past and future? Dive into the ‘educational’ world of PLATO, *Ars Technica*, Technology Review, viewed 12 September, 2019, <<https://arstechnica.com/gaming/2016/10/want-to-see-gamings-past-and-future-dive-into-the-educational-world-of-plato/>>.

Nicholson, K and Naicker, A 2008, GPU Based Algorithms for Terrain Texturing.

Nielsen, J 2012, How Many Test Users in a Usability Study?, *Nielsen Norman Group*, viewed 23 March, 2020, <<https://www.nngroup.com/articles/how-many-test-users/>>.

Preston, C and Colman, A 2000, Optimal Number of Response Categories in Rating Scales: Reliability, Validity, Discriminating Power, and Respondent Preferences, *Acta psychologica*, 104, pp. 1–15.

Rauterberg, M (ed.) 2004, Entertainment computing -- ICEC 2004: third international conference, Eindhoven, the Netherlands, September 1-3, 2004: proceedings, Springer, Berlin ; New York.

Regier, J 2012, Diablo III, Blizzard Entertainment.

Roden, T and Parberry, I 2004, From Artistry to Automation: A Structured Methodology for Procedural Content Creation, in M Rauterberg (ed.), *Entertainment Computing – ICEC 2004*, Springer Berlin Heidelberg, pp. 151–156.

Saleh, F and Ryan, C 1991, Analysing Service Quality in the Hospitality Industry Using the SERVQUAL Model, *Service Industries Journal - SERV IND J*, 11, pp. 324–345.

Scorpia 2015, 1978 – Beneath Apple Manor, *The CRPG Book Project*, viewed 6 October, 2019, <<https://crpgbook.wordpress.com/review-index/1978-beneath-apple-manor/>>.

Shaker, N, Togelius, J and Nelson, MJ 2016, About the book | Procedural Content Generation in Games, Book, viewed 25 September, 2019, <<http://pcgbook.com/about-the-book/>>.

Smelik, R, Tutenel, T, de Kraker, KJ and Bidarra, R 2010, Integrating procedural generation and manual editing of virtual worlds, *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, Association for Computing Machinery, Monterey, California, pp. 1–8.

Smelik, RM, Tutenel, T, de Kraker, KJ and Bidarra, R 2008, A Proposal for a Procedural Terrain Modelling Framework, p. 4.

Stephenson, M and Renz, J 2016, Procedural generation of complex stable structures for angry birds levels, *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 1–8.

Togelius, J, Whitehead, J and Bidarra, R 2011, Guest Editorial: Procedural Content Generation in Games, text, viewed 15 February, 2020, <<https://www.computer.org/csdl/journal/ci/2011/03/06017222/13rRUwkfB22>>.

Unknown n.d., Sample Informed Consent for Online Interview Research Survey, viewed 24 February, 2020, <<https://www.surveymonkey.com/r/RP7W5VY>>.

# Appendices

## Appendix 1 - User Testing Survey

**For each of the following statements, please mark the box that best describes your reactions from using the software, with 1 being very poor and 7 being excellent and, where applicable, clarify your response in the comment section.**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Question | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Comment |
| 1 | Aesthetic quality of the generated terrain. Is the shape realistic? |  |  |  |  |  |  |  |  |
| 2 | Buildings generated in logical locations with no parts clipping the path area. |  |  |  |  |  |  |  |  |
| 3 | Buildings generated at ground level and not floating above. |  |  |  |  |  |  |  |  |
| 4 | Buildings generated without clipping with other objects. |  |  |  |  |  |  |  |  |
| 5 | Enemy positions generated in logical locations within path area. |  |  |  |  |  |  |  |  |
| 6 | Enemy positions generated without clipping with other objects. |  |  |  |  |  |  |  |  |
| 7 | “Small object” positions generated in logical locations within path area. |  |  |  |  |  |  |  |  |
| 8 | “Small object” positions generated without clipping with other objects. |  |  |  |  |  |  |  |  |
| 9 | Enemy positions and “small objects” generated at ground level and not floating above. |  |  |  |  |  |  |  |  |
| 10 | Time taken to generate level. |  |  |  |  |  |  |  |  |
| Additional Comments | | | | | | | | | |
|  | | | | | | | | | |