Artist Away: Procedural Generation of Terrain

Sam Connolly

8 April 2017

A project report submitted in partial fulfilment for the degree of

**Bachelor of Science in Computer Games Development**

**School of Physical Sciences and Computing**

**University of Central Lancashire**

Abstract

The abstract is the summary of the project report within one page (aim for about 500 words). Unnumbered chapter headings, as above, are entered using the ‘Heading (Unnumbered)’ style, which automatically starts a new page.

This template starts the page numbering at the foot of this page. That is, the first page does not have a number.

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.

Currently throughout the modern-day games industry, artists are required to work alongside games developers to assist with the creation of models and levels. This presents problems throughout industry as artists and developers frequently have very different ideas about what is required from art assets, and how they should be implemented. There is a balance between implementing realistic models and artwork, and efficient rendering of an entire scene, however frequently this balance is not met due to a difference of opinion between artists and developers. Constant battling between artists and developers can be a drain on company funds, with so many games studios going out of business, if a company can save money by not having to hire an artist or have an artist’s input on a technical solution then it makes sense not to.

Artist Away set out to generate a realistic looking terrain which could be used as a level of a game, without having to plan the game level beforehand. This would involve the generation of the shape of the level, layering textures on the generated level shape to represent real world terrain, creating level entities such as trees and plants, presenting realistic day and night scenes, and finally presenting a body of water which runs underneath the level. I believe these are the core components of a game level, and all can be generated without the requirement for an artist. Artist Away would do all of this, and could save a company the salary of an artist, and potentially keep that company in business.

Artist Away makes use of procedural generation by generating a height map, the engine then controls plotting vertices per the provided height map, and decides which type of terrain area each vertex belongs to per the vertex’s relative height. The engine then textures the vertices with the appropriate texture depending on the area type, and then generates entities in each area. The engine then defines the vertices for a body of water, and positions it underneath the terrain, all properties of the water will be controlled through the engine. The body of water adds realism to the scene. Finally, day and night cycles are triggered through the engine class, it should simply perform a gradual change of the skybox colour which will represent a time of day in which the sun is at a different position.

Artist Away successfully uses Perlin Noise to generate a height map, and the engine then loads in this height map through one of two interfaces, a pre-generated ‘.map’ file or a dynamic two-dimensional array of values. The engine is responsible for dictating which vertex of the terrain belongs in which area, it then makes use of HLSL shaders to render the appropriate texture to each area of the terrain model. The engine also has a small change to generate a tree or plant in grass areas, and position them at a random rotation. Day and night cycles area achieved by changing the colour of the skybox. Water is created as a plane which sits underneath the terrain, and the properties of the water are controlled through the engine.

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.

**Signature**  **Date**

Acknowledgements

Acknowledge anyone who has helped you in your work such as your supervisor, technical support staff, fellow students or external organisations. Acknowledge the source of any work that is not your own.

Laurent Noel is my project supervisor and has helped with technical explanations of areas which I then went on to implement within my project.

ASSIMP model library is a third-party library which I used for the loading of models within my engine, while I have set up this library myself I have not implemented the core mechanics behind most model loading.

AntTweakBar is a third-party library which provides an interface for controlling things within my engine, while I have set up this library myself, I have not implemented the core mechanics behind drawing the tweak bar and responding to events in the tweak bar.

3D Game Programming with DirectX 11 – Book by Frank Luna which heavily influenced my implementation of the game engine timer class. While my implementation is not identical, the structure of the timer is the same. Also provided me with the methodology of implementing rain through the geometry shader within Prio Engine.

<http://www.rastertek.com> is a group of tutorials which supplied me with the structure for my Direct X 11 engine, while there are similarities between my classes and shaders, there are also a wide variety of differences.

Table of Contents

[Abstract i](#_Toc478568129)

[Attestation iii](#_Toc478568130)

[Acknowledgements iv](#_Toc478568131)

[Table of Contents v](#_Toc478568132)

[List of Figures ix](#_Toc478568133)

[List of Tables x](#_Toc478568134)

[List of Listings xi](#_Toc478568135)

[1 Introduction 1](#_Toc478568136)

[1.1 Background and Context 1](#_Toc478568137)

[1.2 Scope and Objectives 1](#_Toc478568138)

[1.3 Achievements 1](#_Toc478568139)

[1.4 Overview of Report 2](#_Toc478568140)

[2 Literature Review 3](#_Toc478568141)

[2.1 Introduction 3](#_Toc478568142)

[2.2 Game Engine Structures 3](#_Toc478568143)

[2.2.1 Class Hierarchy Based Engines 3](#_Toc478568144)

[2.2.2 Component Based Engines 4](#_Toc478568145)

[2.3 Procedural Generation 4](#_Toc478568146)

[2.4 Summary 5](#_Toc478568147)

[3 Project Planning 6](#_Toc478568148)

[3.1 Introduction 6](#_Toc478568149)

[3.2 Methodology 6](#_Toc478568150)

[3.2.1 Scheduling 6](#_Toc478568151)

[3.2.2 Development Lifecycle 6](#_Toc478568152)

[3.2.3 Testing Plan 7](#_Toc478568153)

[3.3 Requirements 7](#_Toc478568154)

[3.3.1 Selecting a Game Engine and Framework 7](#_Toc478568155)

[3.3.2 Creating a Game Engine 7](#_Toc478568156)

[3.4 Potential Solutions 8](#_Toc478568157)

[3.4.1 Implementation of a Class Hierarchy Based Engine 8](#_Toc478568158)

[3.4.2 Generating Noise 9](#_Toc478568159)

[3.4.3 Generating Terrain from Height Maps 9](#_Toc478568160)

[3.5 Tools and Techniques 9](#_Toc478568161)

[3.6 Legal and Ethical Issues 10](#_Toc478568162)

[3.6.1 Is It Ethical to Automate a Person’s Role? 10](#_Toc478568163)

[3.6.2 Automation In Industry 10](#_Toc478568164)

[3.6.3 Legal Use of Third Party Tools and Resources 11](#_Toc478568165)

[3.7 Potential Technical Issues 11](#_Toc478568166)

[3.8 Algorithms 12](#_Toc478568167)

[3.9 Summary 12](#_Toc478568168)

[4 Design 13](#_Toc478568169)

[4.1 Introduction 13](#_Toc478568170)

[4.2 System Design 13](#_Toc478568171)

[4.2.1 Game Engine Design 13](#_Toc478568172)

[4.2.2 Procedural Generation 14](#_Toc478568173)

[4.3 User Interface Design 14](#_Toc478568174)

[4.3.1 Prio Engine 14](#_Toc478568175)

[4.3.2 Artist Away 15](#_Toc478568176)

[4.4 Summary 16](#_Toc478568177)

[5 Prio Engine 17](#_Toc478568178)

[5.1 Introduction 17](#_Toc478568179)

[5.2 Timing 17](#_Toc478568180)

[5.3 Layout 17](#_Toc478568181)

[5.3.1 Build Structure 17](#_Toc478568182)

[5.3.2 Usage 17](#_Toc478568183)

[5.3.3 Debugging Tools 17](#_Toc478568184)

[5.4 Framework and Libraries 17](#_Toc478568185)

[5.4.1 Choice of Framework 17](#_Toc478568186)

[5.4.2 AntTweakBar 17](#_Toc478568187)

[5.4.3 ASSIMP 17](#_Toc478568188)

[5.5 Rendering Models 17](#_Toc478568189)

[5.5.1 Predefined Models 17](#_Toc478568190)

[5.5.2 Loading Models 17](#_Toc478568191)

[5.5.3 Rendering Models 17](#_Toc478568192)

[5.5.4 Rendering Terrain 17](#_Toc478568193)

[5.5.5 Rendering Water 17](#_Toc478568194)

[5.5.6 Rendering a Skybox 17](#_Toc478568195)

[5.5.7 Rendering Clouds 17](#_Toc478568196)

[5.6 Lighting 17](#_Toc478568197)

[5.7 Rendering 2D Images 18](#_Toc478568198)

[5.8 Particle Systems 18](#_Toc478568199)

[5.8.1 Rain 18](#_Toc478568200)

[5.9 Summary 18](#_Toc478568201)

[6 Artist Away 19](#_Toc478568202)

[6.1 Perlin Noise Generation 19](#_Toc478568203)

[6.2 Creating and Exporting Height Maps 19](#_Toc478568204)

[6.3 Implementing User Interface 19](#_Toc478568205)

[6.3.1 Controlling the Game 19](#_Toc478568206)

[6.3.2 AntTweakBar Usage 19](#_Toc478568207)

[6.4 Concurrent Programming 19](#_Toc478568208)

[6.4.1 Multithreading Noise Generation 19](#_Toc478568209)

[6.4.2 Preventing Multiple GPU Instructions 19](#_Toc478568210)

[6.5 Updating Terrain During Run Time 19](#_Toc478568211)

[6.5.1 Generating New Height Map 19](#_Toc478568212)

[6.5.2 Destroying Old Entities 19](#_Toc478568213)

[6.5.3 Placing New Entities 19](#_Toc478568214)

[6.5.4 Updating Water 19](#_Toc478568215)

[7 Test Strategy 20](#_Toc478568216)

[7.1 Introduction 20](#_Toc478568217)

[7.2 Section 20](#_Toc478568218)

[7.3 Section 20](#_Toc478568219)

[7.4 Summary 20](#_Toc478568220)

[8 Evaluation, Conclusions and Future Work 21](#_Toc478568221)

[8.1 Project Objectives 21](#_Toc478568222)

[8.2 Evaluation 21](#_Toc478568223)

[8.3 Applicability of Findings to the Commercial World 21](#_Toc478568224)

[8.4 Conclusions 21](#_Toc478568225)

[8.5 Future Work 21](#_Toc478568226)

[8.6 Concluding Reflections 21](#_Toc478568227)

[9 Bibliography 22](#_Toc478568228)

[Appendix 1 – Title of Appendix 23](#_Toc478568229)

[Appendix 2 – Title of Appendix 24](#_Toc478568230)

[Appendix 3 – Title of Appendix 25](#_Toc478568231)

List of Figures

[Figure 2‑1 Class Hierarchy Structure 3](#_Toc478949224)

[Figure 2‑2 Entity Component Structure 4](#_Toc478949225)

[Figure 3‑1 Overview of Prio Engine Class Diagram 9](#_Toc478949226)

[Figure 3‑2 Artist Away Use Case Diagram 11](#_Toc478949227)

[Figure 3‑3 Model Class Diagram Within Prio Engine 11](#_Toc478949228)

[Figure 5‑1 Engine Class Properties 21](#_Toc478949229)

[Figure 9‑1 Engine Class Diagram 1](#_Toc478949230)

List of Tables

Similarly, you can automatically generate a list of ‘Tables’. Select a table, right-click it and add a caption labelled ‘Table’ and ‘above selected item’. To update this after revisions, right-click in this table and choose Update Field (or use F9) and then choose to update the entire table. Delete this paragraph before submission.

[Table 1 - Test Results 6](#_Toc476373581)

List of Listings

This list only applies to you if you use code snippets in your report. If you don’t have any listings, remove this whole section including the heading ‘List of Listings’.

You can automatically generate a list of ‘Listings’. After formatting your Code, move the cursor to the first line below your code block and click ‘References’ -> ‘Insert Caption’ in the ribbon menu. Select the label ‘Listing’ or add a new Label called ‘Listing’ if it does not yet exist. To update this after revisions, right-click the table and choose Update Field (or use F9) and then choose to update the entire table. Delete this paragraph before submission.

[Listing 1 - [Main.java] The main class of the program 5](#_Toc476373582)

# Introduction

## Background and Context

With the release of No Man’s Sky (Hello Games, 2016), procedural generation has become a trending topic in recent times. Procedural generation is the creation of an entity, these entities can range from small entities such as trees, to larger entities such as a game level. Artist Away attempted to tackle the challenge of procedurally generating a level without the requirement for an artist, in theory a programmer would be able to instruct the program to design a level and the remainder of work would be handled by the engine.

This presented a challenge; which engine would be most suited for a procedural generation project? While many engines would allow for procedurally generated worlds, few would provide any form of flexible framework. By creating a bespoke engine, it is ensured that procedural generation would be fully supported by the engine. To implement Artist Away, a Direct X 11 based engine named Prio Engine has been designed.

## Scope and Objectives

Prio Engine is designed to support generation of terrain, however there are other core mechanics which are warranted by a game engine such as mesh loading, model control, render management, debugging logs, image loading, text rendering, accepting user input and lighting. Prio Engine must also support procedural generation specific functions, such as the loading of height maps, plotting terrains, the ability to integrate with the Artist Away project. All the while, Prio Engine should be kept as a flexible Direct X 11 engine to serve as a useful tool for future projects.

Artist Away on the other hand serves the purpose of creating height maps through noise algorithms, and must export the height maps in a format which Prio Engine can load. The ongoing integration between the two projects is crucial to success of this project.

## Achievements

I have successfully created a Direct X 11 Engine, which can be exported as a static library, and from the static library can be used to load, control and manipulate models. The engine can also accept height maps in one of two forms, a two-dimensional array, or a text file (exported from the Artist Away project) and from these height maps, procedurally generate terrain. The terrain is successfully textured using different textures per area, and a smooth blend is used between each of the textures. The engine defines a body of water, and successfully refracts the terrain tiling beneath the water.

I have also created a C++ program called Artist Away which uses Prio Engine as a static library, and can interface successfully with the engine to draw terrains, and supply the engine with height maps which are generated through Perlin Noise. Artist Away can also export ‘.map’ files which can be loaded using Prio Engine, as well as 2 dimensional arrays.

Summarise what you have achieved.

## Overview of Report

Briefly overview the contents of what follows in the report. Overview (1-2 lines per chapter):

'Chapter 3 describes the investigation of the problem and presents the top-level analysis as a Yourdon dataflow diagram. ... Chapter 4 contains an overview of the design architecture and examines the key design issues’.

# Literature Review

## Introduction

Each of your chapters should have an introduction to tell your readers what they will find in the chapter.

Summarise current knowledge and what others have done in the various topics of your dissertation – in the application area and in the various technologies that you might have used or did use. Write for someone familiar with computing, but not necessarily expert in the particular topics of your project. Give references to other work by using cross-references to entries in the References section, like this (Turner & Jennings, 2002).

## Game Engine Structures

### Class Hierarchy Based Engines

Class hierarchy structures are found throughout a wide range of industry software. Class hierarchies are typically found in object oriented programming due to the representation of real world objects within classes. However, Pereira, et al. (2013) explains that as a hierarchy grows larger it grows more difficult to maintain. This proves to be problematic during development of game engines, as flexibility is of the utmost importance and as a result, classes can be added at any point in time which results in a rapidly expanding hierarchy. Figure 2-1 demonstrates a real-world implementation of a class hierarchy.



Figure 2‑1 Class Hierarchy Structure

Despite the drawback of large networks of classes, class hierarchies are an incredibly effective design tool for games engines due to the constant ongoing communication amongst multiple classes per frame. The use of a class hierarchy ensures that objects only communicate with relevant instances of classes within their sub-hierarchy, and as a result reduces unnecessary coupling between classes.

Hierarchy based engines are reasonably flexible and can be extended to implement new features in a game. For example, a hierarchy based engine could be extended to implement a phone in a game through the structure shown in figure 2-1, allowing the engine to classify phones and apply different behaviours for different classifications of phones.

### Component Based Engines

Component based engines operate in a very similar manner to hierarchal based engines, in fact most still make use of hierarchies. What separates a component based engine from a standard hierarchal based engine is that a component based engine will define each world object as an entity, each entity will contain multiple components to help it display the properties of a real-world component. Figure 2-2 demonstrates the use of components with a theoretical game entity. Component based engines provide flexibility to add and remove components, for example if a vehicle were to run out of fuel, the movement component could be taken away from the instance of the vehicle entity and the object would remain in the game.



Figure 2‑2 Entity Component Structure

## Procedural Generation

Procedural generation is often found in games when generating heightmaps and drawing different areas based on their height in relevance to the rest of the height map. However, a wide variety of content can be procedurally generated such as levels, adventures, characters, weapons planets, and plants (Julian, et al., 2013). Procedural generation has very few limitations within games, in theory entire games could be procedurally generated, as demonstrated in No Man’s Sky.

Procedural generation usually occurs through artificial intelligence (AI) methods within computer science. For instance, height maps can be generated through Perlin Noise or Perlin Noise algorithms, a very basic form of AI can then plot these heights into a grid, and dictate how each position of a grid at that height should be coloured.

## Summary

# Project Planning

## Introduction

Writing a game engine is no small task, it requires a large amount of planning. Writing an engine which is to be flexible enough to be used by other applications in future complicates matters further. By planning the structure of how applications will interface with an engine, we can reduce unforeseen errors encountered when using the engine over a variety of applications. It also allows us to plan how the engine can be updated in future, as we know what can change and in what way the methods of the engine can change, if we are aware with how other applications will interact with the engine.

## Methodology

### Scheduling

Before commencing development, it can be incredibly useful to schedule out the project. This allows for iteration deadlines which aid with an agile development lifecycle, and helps to scope the project to analyse what elements were to be completed or omitted. Gantt charts are effective at planning, scheduling and tracking progress of a project, however they are most suited for brief overviews of projects. A Gantt chart was created for the project (appendix 1) as according to Wilson (2003) they are effective at displaying a quick overview of tasks, highlighting issues where work was scheduled but failed to be delivered, and allowing the developer to remedy the situation. It was made apparent throughout the development process that the Gantt chart could not serve as a hard rule which development would stick to due to its conflict with agile methodologies and the inability to accurately predict a development period in the future, however it still served as an estimation of where development should be at any point in time.

### Development Lifecycle

An agile approach was selected for Prio Engine since the overall project in fact consists of two separate sub projects. By adopting an agile development lifecycle, Prio Engine can run independently from Artist Away, and can be used with a wide variety of other projects.

Prio Engine consists of many components, and as such both the design and development are required to be as flexible as possible. It is well renowned how fragile IT systems can be if they are not developed in an agile manner (Jacobson, et al., 2016). Developing Prio Engine in an agile format enables the developers to support a wide range of unknown applications in future. However, development of Prio Engine could not be completely agile, as there is a time constraint in the form of the project deadline. This resulted in practicing certain agile methodologies while leaving out others. One agile methodology which enabled planning of Prio Engine was the use of tasks, stories, hills, and epics. A task is a small task, a story consists of many tasks to complete a larger challenge, a hill may have many stories and often refers to a project, and an epic is the combination of these and describes the product in its entirety. The use of tasks within Prio Engine allowed the developer to break up the required work into much smaller tasks, developers can then place importance and urgency on these tasks, and decide on a timescale for each task. This ensures that the crucial components of the engine are developed at the correct times.

The development of Artist Away while agile, was much smaller. This made it a simple development process where the developer could implement the core mechanics of Prio Engine, and then write the height map generation for Artist Away, furthermore Prio Engine could be adapted to accept this height map and the remainder of development and expansion could occur within Prio Engine.

### Testing Strategy

In an agile life cycle, development typically occurs in chunks and as a result so does testing. These chunks are known as iterations and are incredibly important to the success of continuous integration testing, as if the testing of an area is cut short the most important areas of each iteration can be prioritised to minimalize ineffective testing (Jiang & Chan, 2016).

In an ideal world, there would be a set of automated test cases which the engine must pass before progressing into the next iteration, however due to time constraints there is simply no available time for the developer to write a multitude of test cases. Instead, it proves more efficient for the developer to conduct manual test cases upon the engine and continuously test integration with Artist Away at the end of development in every iteration.

Bach (1998) outlines the methodology named ‘Good Enough Testing’ which is an alternative to exhaustive testing, and allows testers to determine when a solution has been deemed to pass testing. I believe ‘Good Enough Testing’ is both an important and effective methodology for the testing of Prio Engine and Artist Away, as the time constraints placed upon the project render exhaustive testing both impossible and ineffective. A period at the end of each iteration is dedicated to testing, with more important test cases prioritised. If minor bugs occur, the project will not be deemed to have failed testing, instead bug fixes will be placed on a development queue at low priority, if time allows towards the end of the project then bug fixing will occur.

## Requirements

### Selecting a Game Engine and Framework

The games industry provides are large list of existing engines to select from, however many of these engines prove not to be flexible enough and limit the scope of projects (Ali, 2016). Through careful planning, selected engines can be eliminated should they not meet the project requirements, however using an existing engine still prevents future flexibility of the project. Often in software development, there are requirements which go unforeseen at the start of a project, and later in the development lifecycle become more apparent. By creating a bespoke game engine, a project can be flexible in the future as the engine can be modify to suit project needs.

Direct X 11 has had overwhelming amounts of support from the development community since its release and has continued to thrive since the release of Direct X 12. This is primarily due to the complexity of Direct X 12 which makes it incredibly difficult to develop with, and is designed solely for experts. Microsoft recognised the complexity of Direct X 12, and released Direct X 11.3 in 2015 as a Direct X 12 alternative. As a result, Direct X 11 has continued to thrive throughout the release of Direct X 12, and has widely available support and resources, while supporting modern features such as geometric shading and tessellation.

Direct X 12 is also limited to Windows 10 platforms only. This drastically reduces the target market for any engine, and coupled with the complexity of the engine. Direct X 11 has been selected as the development framework of choice in this project due to its superior support and simplicity in comparison to Direct X 12.

### Creating a Game Engine

Prio Engine must be able to load, control and render models at 60 frames per second when using the minimum specification graphics processing unit (GPU) which is an NVIDIA GTX 960. Model control is the core mechanic behind games engines, however it is useless if it cannot render at a smooth and realistic speed.

The engine must be able to be exported as an external library, this will enable other applications to use it. Furthermore, the engine must be able to generate terrain from a height map provided by another application. Another requirement from the engine is that it must be able to control model position, rotation and scale and render them correctly, outside of the engine, and without being exposed to the complexity of the engine.

The height map must be generated by Artist Away and be stored in a two-dimensional array or a ‘.map file’, the engine will be able to read either of these methods. Artist Away must make use of the Perlin Noise algorithm to smoothly generate a realistic looking terrain shape.

The complete project must generate terrain which reflects a real-world environment not just in shape, but in appearance too. The terrain should be decorated with a variety of entities which result in the terrain looking more realistic, and less bland.

## Potential Solutions

### Implementation of a Class Hierarchy Based Engine

To plan Prio Engine, I started with a diagram of how I expected the engine to look: 

Figure 3‑1 Overview of Prio Engine Class Diagram

At a glance, it appeared this was a reasonably flexible solution that would account for future changes. For example, should I wish to change the engine from running on Direct X to OpenGL, theoretically, I should only be required to change the Direct3D class. However, this isn’t quite the case as there are some differences between the types of buffers and how they are used in Direct X and OpenGL. It also demonstrates that we could use it as a static library by providing methods for users through the Engine class, and allow the engine itself to handle the inner workings through a network of classes.

Another potential solution which was considered was an entity based system which would control all entities within the scene throughout an entity based manager. Overall, this may have been the better choice, however I believed it would not have brought enough benefit for the overall purpose of this game engine which is procedural generation of terrain to warrant implementation, it would only provide benefits in flexibility for future use.

### Procedural Generation

Different types of generators can create drastically different content and result in two incredibly different levels. It can be difficult to recognize and evaluate the difference in procedural generators (Horn, et al., 2014) and therefore careful planning of how the content will be generated is incredibly important.

Perlin noise is a very popular choice of noise used for height maps, however Perlin noise has now been released, which as described earlier is Ken Perlin’s new and improved version of Perlin noise. By creating a custom Perlin noise implementation, we can ensure that data generated is formatted correctly, and can be manipulated according to parameters which the user can define.

We can then allow Prio Engine to process the height map, create a terrain from the height map, and divide the terrain into areas according to their height in relevance to the rest of the terrain tiles. After dividing terrain tiles into areas, two things may occur; firstly, a relevant texture can be selected for that area, secondly, we can generate a chance to create an entity on that area of terrain. For example, a terrain tile may sit at 50% of the total height within the height map, the engine may decide this is a grass tile, the engine also then states each grass tile has a 5% chance to have a tree at this position. The engine can also analyse nearby tiles and take their properties into account. For example, if a tree exists in a tile within 10 units, then increase the chance to create a tree entity on this tile by 10%.

## Tools and Techniques

GitHub was used as a tool for source control, it enabled modifications to be submit and the ability to roll back to previous versions of the code base for both Artist Away and Prio Engine.

AntTweakBar was used to give users of Artist Away control over select variables used throughout the application. The use case diagram below details how users can interact with Artist Away while it is running, and provide custom parameters to artist away.



Figure 3‑2 Artist Away Use Case Diagram

ASSIMP was used to load a wide variety of models in different file formats into the engine. It is used within Prio Engine by parsing information about meshes and loading in the relevant files, and storing this information in member variables, before passing it over to buffers to be used with HLSL shaders. The class diagram below is an overview of how the mesh loading process works within Prio Engine.



Figure 3‑3 Model Class Diagram Within Prio Engine

## Legal and Ethical Issues

### Is It Ethical to Automate a Person’s Role?

As technology becomes more complex, users of technology are required to adapt to the latest versions of technology. In some scenarios this is perfectly acceptable, however in the modern-day games industry with the expectations of developers constantly rising, small independent companies simply cannot afford to spend time training their artists to use the latest technology and present it in the format required, as well as simultaneously developing the code behind the game. There is also no guarantee on how long it will take an artist to create a resource, due to a lack of reliability caused by human errors. According to (Neumann, 2016), we will continue to face these issues as they are a by-product of human labour. Automating the artistic process would result in a more consistent product being produced, as it programmatically executes a process, and there are very few variables involved in the production methodology. Given independent games companies are typically not renowned for being particularly wealthy, and are renowned for shutting down due to a lack of funds before games are released, it seems logical for smaller companies to risk hiring an artist to design levels, when a stable option of procedural generation can be used to complete the same process.

### Automation in Industry

As automation becomes increasingly popular and more attractive to employers, regulations and legislation regarding automation are being introduced. Unfortunately, we cannot predict what laws will be introduced in the future, however we can analyse the legislation introduced surrounding driverless cars, a popular form of automation. The Vehicle Technology and Aviation Bill imposed the idea that insurance companies which insured the automated vehicle would be responsible for reimbursing any damage caused by that vehicle. If we take similar logic and apply it to the process of procedural generation within games, we can rule out the requirement to pay damages as no insurance companies are involved, and no physical damage can come of a computer program which generates a game level. Because the company would be responsible for any mistakes within the game level, it would not matter if it were an artist or a computer program which designed the level.

While the legal side of automation is clearly not much of a barrier to the games development process, the ethical question remains: is it acceptable to remove an entire role from the industry to save money? There is no clear-cut answer to this question, and the answer is entirely situational and depends on the size, budget, and longevity of a games studio. This author believes it is only acceptable to automate a process when all other options have been exhausted, and it is simply deemed not feasible to hire an artist as a level designer. Business owners have a responsibility to provide careers to working class people, particularly larger business owners. This keeps the economy within the country stable, and provides workers with money to spend on business products.

### Legal Use of Third Party Tools and Resources

Prio Engine makes use of two third party libraries (AntTweakBar and ASSIMP), we firstly need to ensure that the licensing on these products allows for use within a final year project. AntTweakBar uses the zlib/libpng license, which claims the library is both free to use and redistribute, meaning that it is a completely eligible choice for use with Prio Engine and Artist Away. ASSIMP holds a 3 clause BSD license, the three clauses summarise to claim that the copyright notice must be displayed unmodified on binary and source files, and that ASSIMP cannot be used to endorse Prio Engine or Artist Away. Prio Engine can meet these conditions and therefore licensing of third party libraries has provided no issues.

Prio Engine makes use of textures and models, to ensure there were no legal issues Prio Engine has exclusively used free to use and royalty free resources.

## Potential Technical Issues

The first issue I face is keeping the game engine separate from the height map generation, yet keeping procedural generation itself within the engine. My design research resulted in the model where Prio Engine is a static library, which is included within Artist Away. This enabled us to generate height maps in two dimensional arrays, and pass them to Prio Engine as two dimensional arrays, from which Prio Engine could procedurally generate terrain. However, it did not seem feasible for future use to tie the generation of terrain to Artist Away, so a method was placed into Artist Away to export ‘.map’ files, which were rows and columns of floating point numbers which represent a height map. After implementing the ability to import ‘.map’ files in Prio Engine, and place the extracted values into a two dimensional array, we ended up with a method which meant Prio Engine could be run without any requirement for Artist Away, unless you wanted to modify the height map during runtime.

## Algorithms

Procedural generation can take many forms, but for an effective and smooth transition in height as is with terrain, some form of noise generation algorithm is required. (Mikuličić & Mihajlović, 2016) discuss the use of Perlin Noise to generate height maps, combined with Fractional Brownian Motion. This seems like a perfectly reasonable solution to generate height maps, however in 2001 Ken Perlin presented his improved version of Perlin Noise, known as Perlin Noise. According to (Perlin, 2002) Perlin Noise is quicker to generate than Perlin Noise, and in theory could be expanded into multiple dimensions rather than just two. While development of Artist Away has very little interest in the multiple dimension aspect of Perlin Noise, improving the efficiency of generating height maps is an important part of development. Perlin Noise was chosen as the more appropriate algorithm to use within Artist Away, as there was very little change in the complexity of the algorithm and implementation was worthwhile given the performance benefits.

## Summary

The planning behind Prio Engine and Artist Away was a sufficiently thorough process, it covered a wide range of areas from the broad development methodologies through to class diagrams, however the most important aspect of the design process was the ability to adapt to new and changing requirements. This is due to not knowing what issues may occur throughout development, by maintaining a flexible outlook and not tying development to a strict plan, development structure is able to adapt to any problematic situation which may arise.

# Design

## Introduction

Prio Engine targets developers while Artist Away targets users of the game. Having two separate target audiences requires two separate design techniques, as Prio Engine is required to be somewhat complex, while Artist Away is required to be as simplistic and intuitive as possible.

Prio Engine and Artist Away are both written in C++ making use of object oriented methods to simplify how components within both projects interact with one another.

## System Design

### Game Engine Design

(Gregory, 2012, pp. 11-12) defines a game engine as a piece of architecture in which the artistic resources are separated from the core components such as the rendering system, collision detection system or sound system. From this definition, we can interpret that users of the engine will be responsible for loading of resources, but not implementing the code behind it. This is applicable to our engine as a user may want to create a terrain height map as a resource, and have the engine load this. We accommodate for this feature by accepting a map file which simply contains the same information that would be contained in the two-dimensional array generated by Artist Away.

The first step to designing Prio Engine was to research game engine architectures which would enable Prio Engine to be flexible and run with a variety of projects. Continuing from architecture research, a class diagram was drawn up to help visualise interaction throughout the engine (Figure 1).

The next design choice was to move the engine components into a dynamically linked library (DLL) or static library. DLLs in Visual C++ (Microsoft, 2015) describes a DLL as an executable file with shared functions and libraries. Whereas static linking involves exporting code as objects, and importing the functions and code through library objects. While DLLs are more efficient in terms of file size as they only load the required functions into memory, it made more sense to use Prio Engine as a static library as there is a whole range of optimisations that could occur by the optimising compiler in release mode of visual studio, which developers would be completely unaware of. It was deemed the benefits of using the engine as a dynamically linked library simply did not outweigh the security of using the engine as a statically linked library.

### Procedural Generation

## User Interface Design

Onal, et al. (2014) found user interface of a game can drastically impact the level at which a user will co-operate with a game, and can define the level of interest a user has for a game. As Prio Engine and Artist Away are split into two projects, two user interfaces must be defined. Prio Engine will be used by developers and therefore the engine class will be the interface, whereas Artist Away can be used by any none technical user, and as a result requires a more simplistic user interface.

### Prio Engine

Prio Engine is a code driven engine, and does not make use of scenes, demonstrated in larger engines such as Unity and Unreal Engine 4. The user interface for Prio Engine is designed for developers and not end users, the engine can be controlled through the engine class. The engine class may return certain elements such as a pointer to a mesh, so that the user can create models from this mesh, however for the most part the engine will oversee managing the scene and this will be hidden from the developer. Through the engine class interface, the developer will maintain the ability to initialise, shut down and get the time it took to process the last frame for the engine. These three parts of functionality are the core of Prio Engine; however, the developer will not have to implement everything they require the engine to do as a lot of functionality is already covered.

#### Predefined Shape Creation

Prio Engine makes use of predefining shapes such as cubes and triangles, they are incredibly useful when debugging code and requiring a model, instead of the programmer defining the vertices and adding render code, they just call the CreatePrimitive function from the engine class, and Prio Engine will handle the rendering of the predefined vertices. The vertices and indices are all defined in the ‘PrioEngine’ namespace, which can be found in ‘PrioEngineVars.h’. Users can also remove the shapes by a function named ‘RemovePrimitive’.

#### Terrain Creation

Terrain can be created and initialised through the engine class interface. Two methods exist which allow it to import a text file which contains heights where each element is separated by a space, and each row is separated by a new line. Terrain can also be imported through a two-dimensional array of type double. The terrain creation functions return a pointer to the terrain, so the user can modify the terrain elsewhere. The user also maintains the ability to update an existing terrain entity through the engine class.

#### Meshes

Meshes are loaded and destroyed through the engine class interface, the model creation for a mesh is done through the mesh object which is returned through the engine. By loading meshes through the engine, we can ensure that optimisation techniques through frustum culling are handled by the engine, and all models and meshes are rendered in the correct order.

#### 2D UI Images

Prio Engine also offers the ability to load in user interface images which will be rendered last, and therefore sit on top of everything else which is rendered in the window. These are particularly useful for showing UI elements like health, completion, and mana pools in a wide variety of games. The ability to create and remove these from the scene through the engine is an incredibly powerful tool to developers.

#### In Game Text

Prio Engine offers a method of displaying text in a game through Direct X 11, after the functionality was removed in Direct X 10, this can prove to be an incredibly time consuming feature to implement. It also allows text to be updated without having to destroy and recreate the text object, you must simply pass the new string you wish to display to the text object.

#### Skybox Environment

Developers can also update the skybox through Prio Engine to give a level a different feel. By default, the skybox will cycle through day, evening and night times. However, developers can disable automatic cycling, change the time between cycles and select a specific colour they want the skybox to be.

### Artist Away

#### Tweak Bar

AntTweakBar allows users to interact with the program and modify variables from within the executable during runtime. Prio Engine fully supports AntTweakBar usage, and demonstrates the use of AntTweakBar within Artist Away. Through AntTweakBar, users can modify the height map during run time, which also allows for a wider variety of terrains to be generated. Furthermore, the user is given control over the parameters used for Perlin Noise.

Using call-back functions, the tweak bar can access any of the Prio Engine class functions and execute them, allowing flexible demonstrations without the requirement to recompile, and an intuitive interface which is presented to the user on the screen. This interface presents a list of modifiable in game options to the user, and prevents the user requiring to read any documentation to understand how the project works.

#### Control

Camera movement within artist away represents the standard camera controls for most first-person shooter games, and role playing games throughout the industry, ‘W’, ‘A’, ‘S’ and ‘D’ keys are used to control the movement of the camera, while left, right, up and down keys control the rotation of the camera. These keys are standard throughout the industry, and therefore any user who is familiar with a wide variety of games will not be required to read through large amounts of documentation.

Some keys are bound to the function keys on the keyboard, such as F1 which is bound to toggle wireframe, and F2 which is bound to toggle full screen mode.

## Summary

The user interface for Prio Engine exists for developers and thus is much more complex, while Artist Away is designed for end users and therefore is much more simplistic. A combination of allowing developers freedom over engine use and placing power in the end user’s hands allows for an incredibly powerful and flexible user interface, with unlimited potential.

# Prio Engine

## Introduction

Prio Engine is a Direct X 11 games engine written in C++ 11 for Windows 7 and developed by Sam Connolly in 2017. It is designed for experienced games developers with an in-depth knowledge and understanding of C++. It provides a method of developers loading game content and resources such as textures, models, and UI images without being concerned with the complex implementation behind the methods.

Prio Engine has also been specifically tailored for support of the Artist Away project, which enables it to load terrain files and support procedural generation of terrain. Support of procedural generation has resulted in quite a large engine which may present unnecessary features to some developers, however the engine can be rearranged and modified by developers should they so wish.

## Timing

Timing within games sits within the core of every engine. Typically, timing is implemented through one of two methods within the game world pattern, unlimited frames per second or fixed frames per second. Nystrom (2014) describes the use of both timing methods, and goes on to describe that an ‘unlimited frames per second’ method will noticeably stutter on low performance hardware, while a fixed timing loop relied on knowing the exact speed the hardware running the game could run at. After some investigation, it was clear that Prio Engine could potentially be run on a wide variety of hardware, therefore it proved logical to use an ‘unlimited frames’ method as opposed to a fixed timing loop. If physics were a key component within the project, then it would make sense to run at a fixed timing loop as physics calculations with small floating point numbers are much more accurate at fixed intervals.

Implementation of the timer class is done through the windows system clock class, the change in time is measured at start of every frame which is used as the update time of the game. Any form of movement or rotation is then multiplied by the calculated update time, which prevents transformations occurring at different speeds on varying hardware.

## Layout

### Build Structure

Prio Engine is a built around a composition class hierarchical structure, which entails frequent communication throughout many classes and destruction of all objects belonging to a class when it is destroyed. The compositional architecture helps to keep memory leaks to a minimum, as each class is responsible for cleaning up after any other classes which it may create. The structure of the engine is shown in a class diagram in Figure 9‑1 with some of the details of the graphics class excluded.

#### Engine Class

The engine class provides the interface to the overall engine functions for developers. The engine class oversees creating the graphics, logging, timing, and input classes. These classes form the core of a game engine. Any of the communication between the core elements of the game engine occurs through the engine class, this keeps the engine acting as a manager class which can control how each class may communicate with another.

#### Graphics Class

The graphics class is responsible for the perpetration and rendering of models. This includes plotting vertices and indices for any models which are to be rendered to the scene, initialising, and controlling shaders which are used for rendering models.

#### Input Class

The input class controls user input, events are triggered through windows messages which are processed in the engine class, the key corresponding to this event is passed into the input class which responds to the event by raising or lowering Boolean flags.

#### Timer Class

The timer class is responsible for keeping track of time within the game engine. It is effectively an interface to the system clock, which can measure change in time. The change in time is stored once per frame, this is known as the frame time or occasionally update time throughout Prio Engine.

#### Logger Class

The logger class is a debugging feature of Prio Engine, it’s purpose is to store all information about a run inside a text file, these text files can then be analysed through either an automated test or a manual investigation by a developer, to determine the success or failure of a run and the events that occurred within that run.

### Usage

Prio Engine is designed to be a flexible engine which is capable of being integrated with other projects. Prio Engine can be exported as a static library, and controlled through the engine class interface. Figure 5‑1 Engine Class Overview demonstrates the properties and methods belonging to the engine class, and what is available to developers to use.



Figure 5‑1 Engine Class Properties

The project which imports Prio Engine as a static library must be a win32 project, and is responsible for creating the window, and passing information such as the handle of the window through to the engine class. Win32 project’s must contain a WINAPI WinMain function in order to create a window which is usable by Prio Engine.

### Debugging Tools

#### Logging

The logger class is built around a singleton pattern as every class throughout the engine which has any potential to fail requires access to the logger. Many resources advise against the use of singletons within modern software architecture, however in Prio Engine’s architecture it is essential as it enables developers who are using the interface of the engine to access the logger, and removes the requirement for developers to define the logger in their code. If a developer were required to define a logger within the code, it would cause issues when they did not need to use the logger, however the engine expected the logger to be defined and as a result only held an extern definition of the logger.

The logger also enables effective logging of memory allocation and deallocation. By placing all memory allocation and deallocation into a separate log, the log can be analysed to quickly point out where memory leaks exist within the application.

To improve performance within Prio Engine, logs are only written in debug mode, as writing to a text file multiple times in the same frame does have a reasonable impact on the time taken to process a frame.

#### Wireframe Mode

Wireframe mode poses a method of viewing the indices which connect vertices, while bypassing the rasterizer stage of the graphics pipeline. This is particularly useful within Prio Engine as a wide variety of models can be loaded in through different methods, should a model look incorrect, the developer can analyse the model through wireframe mode to investigate what the problem with the model is.

Wireframe is implemented by supplying Direct X with a different rasterizer state, which uses the wire frame fill mode as opposed to the standard solid fill. This gives the illusion of bypassing the rasterizer stage when in fact, it is still occurring it just simply does not run the pixel shader on the inner areas of connected vertices.

## Framework and Libraries

Frameworks and libraries are particularly useful for providing flexible methods of achieving a goal which would otherwise be lengthy to implement, and likely prove not as effective.

### Choice of Framework

Prio Engine uses the Direct X 11 framework to utilise the graphics capability within a range of computers. Direct X 11 was selected as it supports geometry shading unlike Direct X 9, it also supports tessellation, whereas Direct X 10 lacks this feature. Direct X 12 was considered for the project, however the project was focused on procedural generation, and Direct X 12 while providing performance benefits, did not necessarily bring anything which benefited the specific project which would have made the complexity of development worthwhile.

### AntTweakBar

AntTweakBar is a graphical user interface library which provides a small box which allows users to modify variables in code. It is particularly useful for demonstrating technologies, as it provides the ability to manipulate variables without having to recompile the code with each change.

AntTweakBar fully supports a wide variety of frameworks including Direct X 11. This makes it a fantastic choice to implement into the engine, as developers can add a tweak bar with their own variables and Prio Engine will automatically handle the rendering and updates of the tweak bar.

### ASSIMP

ASSIMP (Open Asset Import Library) is a library designed to load a wide variety of model formats and information which can be tricky to extract from files such as the names of diffuse textures, alpha maps, and specular maps.

The use of ASSIMP within Prio Engine provides the flexibility to use a multitude of model formats, without being concerned as to whether a new loading function needs to be implemented into the engine. ASSIMP will also provide detailed error messages which Prio Engine writes to the debug logs, this ensures that if any errors do occur within ASSIMP when loading meshes, the developer is fully aware of the issue and can investigate possible solutions.

### SFML

SFML is a lightweight and flexible library which enables loading and usage of multimedia within a C++ application. SFML supports a wide variety of data types, resulting in a wider range of available sounds.

SFML is used within Prio Engine to enable the loading and playback of sound files within the engine. The use of sounds within an engine provides ambience to games, SFML has been implemented to improve the users experience by further immersing them within the game through sound.

## Models

### Predefined Models

Prio Engine possess the ability to create cube and triangle predefined models which can be created through the engine interface. The predefined models can be rendered with solid colours, textures, diffuse lighting, or specular lighting. While the use of predefined models within games is incredibly limited, it serves as a useful debugging tool as each model displays a visual representation of where a model exists in the game world.

### Loading Models

Models within Prio Engine are loaded in through the engine interface, and information about a model is stored in a mesh class to avoid the requirement to load identical model’s multiple times. The mesh class takes a string parameter which refers to the location of the model file, it then parses information about this file through ASSIMP, which loads in vertex, index, normal, UV, and texture data and stores it in member variables belonging to that mesh class.

The texture data which is loaded in through ASSIMP can involve diffuse maps, normal maps, specular maps, and alpha maps. It will check the ‘Resources/Textures/’ directory relative to the executable file for all the textures which a model file contains. Through parsing this information, Boolean flags can be set within the constant buffer of the shader file to indicate which maps should be used and which should be ignored. This allows for a flexible approach which will render the mesh using all data available instead of sacrificing techniques which the mesh is missing information for.  
Instances of models are created through the ‘CreateModel’ method belonging to the mesh class. The mesh class stores a list of instances of models, and as a result acts as a form of manager for the model class. Each instance of the model class contains information about the position, rotation, and scale of the model.

### Rendering Models

The vertex and index data are loaded into buffers and stored within the mesh class. By storing information in the mesh class, the same vertex and index buffer can be used to render each model at different positions. The world matrix is calculated for each model through the position, rotation, and scale properties. The world matrix is passed into the shader, and is the only property which changes within the vertex or pixel shader when rendering a group of models.

Prio Engine renders meshes in batches, this is due to the nature of loading information in computers. It is a slow process to retrieve information due to the physical distance between the random-access memory (RAM) and the graphics card, by processing identical data in batches there is a performance gain as the graphics processing unit (GPU) caches model information for it to be reused, and the retrieval of information only occurs once.

### Terrain

Terrain is generated through Prio Engine in a grid based format, each vertex of the grid matches the height provided through the height map, and each area of the terrain is textured depending on its height. The grid is divided into areas based on height, and through analysis of areas within the terrain entities can be positioned within the world.

#### Importing Height Maps

Height maps are generated through a Perlin noise function in Artist Away, however the issue of importing the height map into Prio Engine remains. To solve this, two solutions were presented: passing in height maps through text based files, and passing in height maps through two dimensional dynamic arrays of type double.

The purpose of having two separate standards is allowing for quick loading of a previously generated height map while a two-dimensional array approach allows for slower loading of a more flexible terrain.

In order to satisfy a standard for text based height maps, I have created a ‘.map’ file standard, which contains floating point numbers separated by spaces, and columns separated by new lines. Each number represents a height of a vertex. The dimensions of the height map by counting the number of columns and rows. These files are incredibly quick to read in, and enable the programmer to reuse a level multiple times. However, it provides no flexibility, once implemented the terrain cannot be changed by passing another text based ‘.map’ file to the terrain object, however it can still accept a two-dimensional array of type double.

To create a more flexible approach and fully support procedural generation of terrain, a two-dimensional array of type double can also be passed to the terrain object. It works in a similar way to the ‘.map’ standard, however requires the height and width of the map to be passed in as parameters to the function. Each element within the array represents the height of a vertex, and the values of the array are copied into the terrain object as to avoid exceptions by incorrectly deallocating an array which is still in use by another object.

See 5.5.4.4 for information on how to update terrains.

#### Calculating Terrain Type

The next step of procedural generation of terrain is to divide the terrain into areas. To provide a flexible solution which can adapt to a large variety of different shapes of terrain, Prio Engine divides terrain into areas based on their relative height. This is to say, above 60% of the highest point will be rock, above 30% of the highest point will be grass, above 15% of the highest point will be dirt, and anything below will be sand.

Using a percentage based calculation allows for terrains to be scaled, and tiles to be evenly distributed. Each tile is textured with an appropriate texture using triplanar texture mapping within the pixel shader, this prevents stretching of textures across scaled terrain models.

#### Positioning World Entities

After calculating each tile of terrain, entities are placed upon the terrain to improve the visual appearance of the terrain, and properly reflect the terrain of a real world. World entities are generated on chance given that certain conditions are met. The conditions for generating trees on terrain within Prio Engine are: Must not have another tree located within 30 units, and must be located on a grass or dirt tile. If both conditions are met, then a random number between 0 and 100 is generated, if the random number is less than 2, then a tree is put on a list to be created at that location and at a random rotation, this is the equivalent of giving a 2% chance to create a tree. A similar process occurs for plants; however, a higher percentage chance can be used as tree models have a much higher polygon count and drastically impact performance.

#### Updating Terrain

Updating terrain must be done via a two-dimensional array of type double, there is currently no support through ‘.map’ files to update an existing terrain’s height map, however the terrain object can be destroyed and recreated with another ‘.map’ file at run time.

Updating terrain occurs through the engine object, it accepts a new height map and recreates the terrain based on the new height map. This is quite a lengthy process, so it is recommended to run the process on a separate thread, and join the thread once the new terrain has been created.

After the terrain has been processed, the world entities are required to be recreated, this is another lengthy process and therefore should occur on a separate thread. The ‘RemoveScenery’ and ‘AddScenery’ functions within the engine object make this simple to do, Prio Engine’s graphics object will raise Boolean flags to avoid altering models while they are being raised, this will enable the use of concurrent programming. See 6.4 for more information on concurrent programming.

### Water

A body of water is created within the terrain object, and sits as a flat plane spanning the length and width of the terrain. The water depth defines how high the water will be, and in turn how much of the terrain it will cover. The body of water is created through 4 processes, these are; Rendering a water height map, rendering refraction, rendering reflection and finally rendering the surface.

The height map for the water is created on a separate render target, as it enables the height map to be passed around as a ShaderResourceView. The height map is created normal map at four positions, the normals are then totalled to produce the height of the map at that position.

The refractive surface works by only rendering objects which are below the surface of the water, acquiring the terrain colour in the same method as the terrain pixel shader would, and modifying the colour of that terrain depending on the depth of the terrain tile from the water plane.

The reflective surface copies the main cameras properties into a reflection camera, however the reflection cameras properties are manipulated in the following ways: the rotation about the X axis of the reflection camera is inverted, the Y position of the reflection camera is moved below the plane of water, however the distance from the water plane does not change, it is simply in the opposite direction, finally the Y axis of the reflection camera view matrix must be inverted, this will prevent reflections being displayed in the incorrect direction on the plane of water. Figure 5‑2 demonstrates how the reflection camera is altered from the properties of the main camera.



Figure 5‑2

The final process of rendering water is to calculate the water surface. In theory, a flat surface which blended reflection and refraction surfaces together would suffice to represent a body of water. However, ripples on the water surface can be generated through manipulating normals which result in light being reflected off the water at different angles, this causes the water surface to appear as though it is moving when in fact, it is a flat surface. A movement variable is updated every frame to alter where the normals are sampled from, this results in moving waves.

An alternative method of calculating waves are known as Gerstner Waves, this is a method which will simulate the vertex positions of a wave and provides both peaks and troughs (Finch, 2007). While these waves look somewhat more realistic, they have not been implemented within Prio Engine due to time constraints placed upon the project.

### Foliage

Rendering foliage has previously been considered an expensive process, and one which older hardware would struggle to handle due to the strain it would place upon the GPU. However, in modern day hardware this is no longer an issue, as a process named billboarding is commonly used upon three intersecting quads, and foliage drawn as a texture of these quads (Pelzer, 2007).

Prio Engine uses an intersecting quad method to billboard foliage textures, where the quads are positioned in the shape of an asterisks. This result in a thick looking foliage, however when viewed from above the shape is clearly visible. One method which would overcome this issue would be to have one more quad lying flat against the ground, however the appearance may be flawed when viewed from ground level.

The foliage pixel shader makes use of two textures, a diffuse texture, and an alpha map. The diffuse texture describes indicates each colour of the map, while the alpha map indicates which areas of the map are transparent. By discarding pixels which have an alpha value of zero and using an alpha blending state (see section 5.6.2), the intersecting quads appear thick and look as though they are a full 3D model.

Back face culling is a process done within Direct X in which the rear of models is not rendered to improve performance. Rendering of foliage through this method requires back face culling to be disabled, as when the camera views any one quad, the two intersecting quads will have their backs facing the camera, and as a result Direct X will not render all the quads to leave us with a full view of the foliage.

While foliage looks thick, it looks unrealistic without movement. To implement movement, each vertex is assigned a value, the value indicates whether the vertex is located at the top or the bottom of a quad. Top vertices are then displaced back and forth to provide the impression of waving grass. This process provides an incredibly cheap method of rendering grass; however, it is not an accurate representation of grass within the real world. To further improve the performance of rendering foliage, each quad is instanced (see paragraph 5.5.8.2).

### Skybox

The skybox within Prio Engine is implemented as a large sphere, in which back face culling (see section 5.6.3) is disabled. The sphere moves with the position of the camera, this gives the impression that the user never gets any closer to the camera, nor any further away. The skybox consists of two colours a horizon colour and an apex colour. The horizon colour is what colour the bottom of the sphere will be, while the apex colour is the colour of the top of the sphere. The two colours are blended together using a gradient, this results in a realistic change in colour representative of a real-world sky.

#### Day, Night, and Evening Cycles

Day, night and evening times can be represented purely through the skybox. The skybox has an apex and horizon colour which is passed into the pixel shader and rendered every frame, as described in section 5.5.7. The skybox also contains methods to change between day, night and evening pre-set colours. The colour of the skybox is adjusted over time, until it reaches the target colour for both apex and horizon colours.

To create a realistic effect upon the scene, the scene ambient light must be updated to be the same as the horizon colour, this alters the colours slightly of models within the scene, and results in scenes reflecting a real-world time of day.

#### Rendering Clouds

Prio Engine defines a plane which is slightly curved for clouds to move across. Each vertex within the plane is initialised with a texture UV value, to sample cloud textures at that point. Two cloud textures are passed into the pixel shader along with an offset for each texture. The offset is incremented every frame, this results in the texture being sampled at a slightly different position each frame, and as a result gives the illusion that the cloud is moving across the plane. This process is incredibly cheap to render and provides realistic looking clouds, however it does contain issues, the plane is visible when reaching the edges of terrain. Usually games would constrict where the user can go within the level to prevent the user seeing the edges of the plane, however that is not an option for Prio Engine currently as it serves as a technical demonstration.

### Optimisations

Achieving a constant 60 frames per second is of the utmost importance in any PC game. Optimisations within game engines cover a broad spectrum of issues, however optimisations within Prio Engine are specifically for reducing the load on the GPU.

#### Sphere Based Frustum Culling

Frustum culling is the process of defining planes which surround the cameras viewing angle, and checking if points lie within the planes. If a point sits outside of the planes, then it is not rendered, and potentially saves thousands of polygons being rendered unnecessarily. However, frustum culling does cause issues when large models are visible on the screen and the camera is moving, as the objects appear to pop out of view, particularly if the centre point of a model is at the bottom.

Sphere based frustum culling takes a model and places a theoretical sphere around the model, if the sphere intersects the planes at any point then it is deemed to be inside the planes. This solves the issue of models popping in and out of view, as the model will always be rendered when there is the potential for it to be in view.

#### Instancing

Instancing provides a performance benefit to the graphics pipeline as it removes the necessity to reuse multiple vertex buffers and index buffers, instead the vertex and index buffers are set once on the GPU, and a list of positions and other properties are passed to the GPU. The GPU stores information about each vertex on the geometry shader, and draws it at the position described in the instance data. The performance benefit comes from removing the requirement to pass data to the GPU, which is a lengthy process due to its physical distance from the random-access memory (RAM).

## Rendering Techniques

A wide variety of rendering techniques are available for use with Direct X. These techniques achieve different goals, ranging from appearance to performance.

### Lighting

Lighting improves the realism of a scene, and enables the use of depth and detail within textures. Lighting exists in two forms, directional and point. Directional lights emit light across an entire scene in a constant direction and the strength of the light does not change, while a point light emits from a position in a world in all directions but the strength of the light attenuates depending on the distance from light.

#### Diffuse Lighting

Diffuse lighting is a type of directional light which lights parts of a model which face the light. The facing direction of each vertex in a model is determined through the normal of each vertex.

#### Ambient Lighting

Ambient lighting provides a default lighting to each element on the scene, independent of the lights direction. This is typically applied to every model within the scene and helps to indicate the general colour of the lighting within the scene.

### Blending

#### Alpha Testing and Blending

Alpha testing is a method where the alpha channel of each pixel within a polygon is checked, and if the alpha value is below a threshold then the pixel is discarded. This is a particularly useful method for cutting out textures within pixel shaders using alpha maps. Figure 5‑3 Alpha Testing demonstrates an example of alpha testing used within Prio Engine.

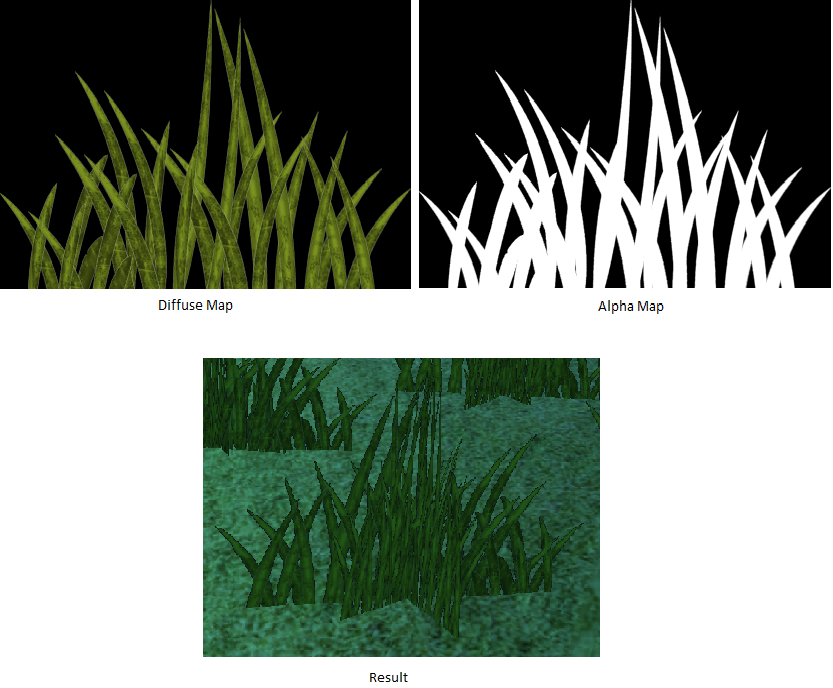


Figure 5‑3 Alpha Testing

Alpha blending allows areas of models to appear transparent, and for other models within the scene to be visible through the transparent areas. While alpha blending is particularly useful on areas where very little is occurring, alpha blending struggles to blend areas where objects block vision to other objects. Alpha blending is used scarcely within Prio Engine; however, it is used to blend clouds with the skybox (see Figure 5‑4 Alpha Blending, where black areas of a texture represent an alpha value of zero). This is due to the unlikeliness that there will be any sorting issues either now or in future development with clouds as they are separated from the rest of the scenery.

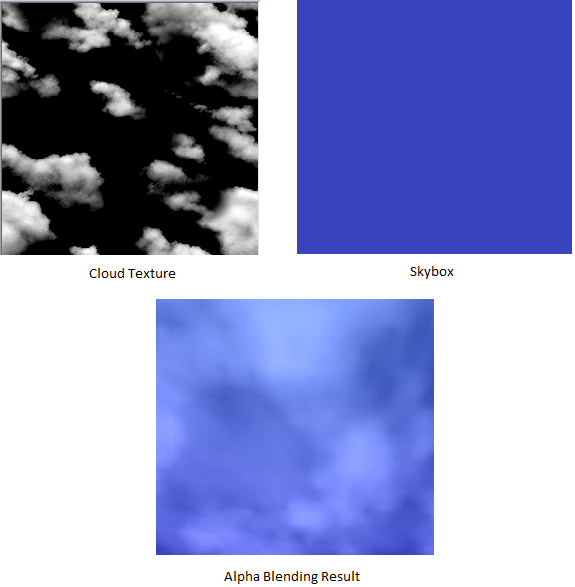


Figure 5‑4 Alpha Blending

### Back Face Culling

Back face culling is an optimisation technique that occurs when areas of a model should not be visible from the cameras direction, so they are omitted from the rendering process. Back face culling is used for most models in Prio Engine, however it is disabled when the rear view of models is required to be disabled, for example, when rendering the inside of the skybox sphere.

## UI Images

User interface (UI) designs within games are typically created through 2D images being displayed on the screen at a specified pixel coordinate. Prio Engine supports the loading of UI images through the engine class object. These images are particularly useful for providing the user with a heads-up display (HUD) which contains information about the current game or level.

To render UI images the graphics object within Prio Engine stores a base view matrix, which is the view matrix when the main camera is rendered at the origin of the world. This base view matrix is used in place of the camera view matrix when it is passed to the shader, to ensure that the image is displayed at the same position every frame.

## Particle Systems

Particle systems can be created on both the central processing unit (CPU) and the graphics processing unit (GPU). CPU based particle systems tend to be simpler and quicker to implement however data is still required to be passed to the GPU, while GPU based particle systems are slightly more complex and time consuming to implement, and prove to be less flexible than CPU based particle systems. GPU particle systems make use of the geometry shader, which is a stage which sits between the vertex shader and pixel shader in the graphics pipeline. The geometry shader is incredibly efficient at rendering large numbers of the same shape.

### Rain

Rain has been implemented within Prio Engine using a vertex, geometry, and pixel shader. Rain has been implemented as a GPU particle system as rain is something which has very little impact on a game, and where possible should be computationally inexpensive.

The rain particle system requires a vertex and geometry shader to update the rain particles, and a vertex, geometry, and pixel shader to draw the rain particles to the screen. The update geometry shader determines whether a particle is an emitter or a rain droplet, if it is an emitter then a new rain droplet is defined and positioned at a random location within a set radius, and this droplet is appended to the existing particles. The draw shaders are responsible for moving the position of the existing rain droplets, and setting the colour of the rain droplets.  
Each rain particle is rendered as a point line, this was selected for rain as droplets fall so fast that the user would not be able to recognize the actual shape of the droplet. A key benefit of rendering droplets as a line list is that it reduces the number of vertices which are required to be rendered, this takes more strain off the GPU and ensures a cheap method of rendering.

### Snow

Snow is rendered in an identical manner to rain described in section 5.8.1, the only difference lies in the colour which is used within the pixel shader and the speed at which the snow falls. The length of each line calculated within the geometry shader is determined by the current acceleration of the particle. By slowing the particle down, the line is reduced to a small dot, which resembles a very fine snowflake.

## Summary

Prio Engine makes use of a wide variety of techniques to create a flexible Direct X 11 based game engine which can serve for the use of multiple projects, however focusses on the support of terrain generation. The engine is based solely upon graphics, and relies on any physics calculations to be implemented by other developers. Due to Prio Engine’s design to be used as a static library, a multitude of features are required to be implemented, however it is not necessary that all of them be used in each project.

# Artist Away

Artist Away is responsible for controlling and generating height maps which are used within Prio Engine.

## Perlin Noise Generation

Perlin noise is a type of noise which generates a smooth style of noise between values 0 and 1. It was introduced by Ken Perlin, however he later released his improved Perlin noise algorithm (Perlin, 2002). Artist Away makes use of improved Perlin noise to generate height maps with values between 0 and 1. These values can be amplified by a control known as ‘gain’ to uniformly alter the values.

Perlin Noise can be calculated on either the central processing unit (CPU) or graphics processing unit (GPU) (Green, 2005). Artist Away pursues a CPU implementation of Perlin Noise, as it is simple to write noise generated on the CPU to a file, whereas it becomes slightly more complex when the noise is generated in a shader. Perlin noise can be generated in two-dimensional or three-dimensional space. Artist Away generates Perlin noise in two-dimensional space, as three-dimensional space poses very few benefits in generation of terrain.

Perlin noise accepts an x, y and z parameters which represent the coordinates of a point. These parameters are normalized to find the position of this coordinate within a square. Four pseudorandom gradient vectors are generated from the four points of the square. Four more distance vectors are generated which calculate four surrounding points on a grid about each point on a square. The dot product between the pseudorandom gradient vector and the distance vector generate influence values, which will determine how strong a noise value will be. Four influence values are generated as there are four separate points on a square in 2D noise, however in 3D noise eight values would be required. A linear interpolation is performed upon each of the influence values with one another to perform a smooth transition throughout each Perlin value generated, however due to the way in which linear interpolation this curve looks unnatural. To make the curve more natural, an ease curve is applied to further smooth the curve between each of the values.

Perlin noise accepts a series of parameters to manipulate noise, these include; amplitude, persistence, frequency and octaves. Amplitude controls how high a noise value can possibly be when it is calculated. Persistence calculates how quickly amplitude will deteriorate. Frequency controls how drastic the changes in the noise values will be. Octaves are the number of times which the Perlin value will be calculated, and then added to a total number. Each of these effects are controllable through Artist Away, as they enable a method of customizing height maps which can be used for both terrain generation and foliage generation.

## Creating and Exporting Height Maps

Height Maps are created through Perlin noise, described in section 6.1. The map itself is stored as a two-dimensional dynamic array of type double, and can be acquired through the ‘GetMap’ function belonging to the HeightMap class.

As mentioned previously in section 5.5.4.1, height maps must adhere to two standards, and Artist Away is responsible for adhering to each of these standards. The first standard is a two-dimensional dynamic array of type double. Artist Away generates this and stores all information regarding the height map in a member variable which is a dynamic array. The height map class allows access to the dynamic array, however if objects deallocate memory from this array then it will cause a runtime error during the shutdown function of the height map. For this reason, it is recommended that height maps are generated and stored in text files, matching the ‘.map’ file standard. The only time when a dynamic array based heightmap is recommended is when the height map is required to be generated in real time, this is the case with procedural generation, and as a result is the chosen method for Artist Away.  
Artist Away also exports height maps which are created as text file with a ‘.map’ extension. Each column is separated by a space; each row is separated by a new line. These text files can be loaded in to a dynamic array by analysing the text file, this process occurs within Prio Engine.

## User Interface

### Controlling the Game

The camera movement is controlled through W, A, S and D keys, while the rotation of the camera is controlled through up, down, left and right arrows. AntTweakBar is used to control elements within the scene.

### AntTweakBar Usage

The user interface within Artist Away is primarily implemented through AntTweakBar. The tweak bar allows for manipulation of height maps, control over weather effects and manipulation of water properties. The tweak bar is implemented through callback functions which are used as getters and setters to alter the values within the engine. Callbacks are used in place of basic read only and set variables as callbacks prevent the requirement to update variables within the engine every frame, and allow for concurrent programming.

## Concurrent Programming

### Multithreading Noise Generation

It was found that generating Perlin noise on the CPU is an incredibly intensive process, and thus takes a long time to process. To keep the game operational while height maps are being updated, Artist Away runs all generation of height map and updating of terrain on a separate thread. This enables the user to continue using Artist Away while the new terrain is being generated.

Multithreading is only possible through callbacks within Artist Away, as AntTweakBar does not run callbacks on a separate thread, they must be manually created and joined through the program which runs AntTweakBar. A Boolean flag is raised in Artist Away when the thread is ready to be rejoined to the main thread, this flag is checked in every frame of the main thread, once the thread is rejoined to the main thread, it no longer runs.

### Preventing Multithreading Issues

Multithreaded programs are prone to errors when two separate threads attempt to use the graphics device or information passed the graphics device at the same time. Multithreading within Artist Away operates upon modifying elements and member variables belonging to the engine object, all the while the engine object is still running on the main thread and uses many member variables throughout every frame. One area which proved particularly problematic was modifying the meshes and models used when updating the terrain object. Issues arise as the main thread attempts to render a list of models contained by each mesh, while the separate thread attempts to remove the models belonging to that mesh, to generate new models at the correct position. This is solved by raising and lowering Boolean flags when the render method for meshes starts and ends, the second thread will wait for the flag to be lowered before continuing and modifying the list of models. This does mean that the second thread will not run as quickly as it possibly could, however it ensures that the program will not crash.

## Updating Terrain During Run Time

The update process is triggered through a button click event in AntTweakBar, which in turn calls a callback function to begin the update. The update process is run on a separate thread as to avoid causing the program to hang.

### Generating New Height Map

Height maps are stored as two dimensional dynamic arrays of type double. Height maps can be updated through the ‘Update’ function found in the class declaration. It is required that the programmer sets the new height and new width of the height map before calling the update function, unless they wish for the height map to remain the same dimensions. Upon calling the update function, all memory for the existing dynamic array is deallocated, to apply the resize the array is allocated memory of the newly requested width and height. The initialisation process runs upon the new dynamic array to populate it with perlin noise values.

### Updating Entities

There are four elements which are built around height maps which will need to be regenerated; terrain, water, foliage and terrain entities (trees and bushes). Before updating occurs, the engine is instructed that updating of these entities is about to occur, this avoids render functions attempting to render entities which are being modified. The update function of terrain is called, the update function is responsible for shutting down allocated memory, and following that it will reinitialise the terrain by beginning the procedural generation process with the new height map.

Once the terrain has been regenerated, the water object is updated by Prio Engine. The water is destroyed, and recreated and the same width and height as the terrain. This ensures that the water sits as a flat plane throughout the terrain, and does not only cover certain areas within the terrain.

Following the water, a new height map is generated for foliage. Foliage is generated by analysing a map of equal height and width to the terrain height map, but the foliage height map should have a much higher frequency.

Finally, the terrain entities are updated. The simplest method of reinitialising the terrain entities is to remove all model instances belonging to the mesh object, and recreate them from the positions generated by the terrain object which was regenerated in the first stage of recreation.

# Test Strategy

## Introduction

Software testing is essential to ensuring the stability of a product and allowing for continued development without unexpected behaviour occurring because of previously written incorrect code. It is important to note that a failed test is a correct test, as it is uncovering bugs within the code. Testing also analyses the severity of the bugs within code to dictate a course of action, throughout this section bugs within the program will be uncovered, however some may not be fixed depending on severity, this is an informed decision made by the developer to progress development of Prio Engine and Artist Away as much as possible.

## Time Dedicated to Testing

The methodology behind testing and when to start testing was briefly outlined in section 3.2.3 however, when testing should stop and development should be resumed has never been clarified. A common approach is to analyse the reliability of the software and weigh up the cost of testing with risk of bugs in the software. While this method works to an extent, it is not incredibly flexible for a project with a fixed deadline, and therefore not an awful lot of use to the project.

The failure size proportional model considers the severity of items if they were to fail against the cost of testing (Zachariah, 2015). This provides the flexibility to continue development where necessary, and ensure that development efforts are not being focussed in the incorrect areas of the project.

## Test Cases

### Identifying Test Cases

An incredibly effective method of identifying software requirements and test cases is the ’10 Minute Test Plan’. It involves grouping up developers, and having each of them write down everything they expect a piece of software to achieve, and what is required of that software. At the end of the 10 minutes, each of the developers collaborates, crosses off any duplicate requirements and creates test cases from the remaining requirements. It has been found that this method is incredibly effective at defining attributes which describe the software, components which will be required to be implemented within the software and the capabilities which the software can achieve (Whittaker, 2011).

The idea of a ten-minute test plan is not strictly applicable to Artist Away or Prio Engine, as all development is the product of one person, however a similar concept can be applied and the single developer (in this case, myself) can quickly outline all core concepts and requirements of the software. The resulting requirements can be found in FIGURE XXXXXXX

### Prio Engine

#### Performance

Table 1 – Memory Test Result

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case** | **Pass Criteria** | **Result** | **Severity** | **Action** |
| Run Prio Engine executable, monitor RAM usage in Windows Task Manager. | RAM usage should be consistent, should not gradually rise as that indicates memory leaks exist within the program and it will eventually crash. | Passed | Urgent | None required. |
| Run Prio Engine in Visual Studio Debug Mode, use the CRT library to check for memory leaks. | No memory leaks should be found in the program, Prio Engine should be responsible for cleaning up allocated memory. | Failed | Low | Go through all code and ensure all Direct X types are correctly released if they have been initialised, and a corresponding delete exists for any object which is allocated memory. |
| Run Prio Engine release executable, monitor the FPS through in game text. | The FPS counter should never drop below 60 frames per second on GTX 960 (minimum spec graphics card). | Passed | Urgent | None required. |

## Section

Text goes here.

## Summary

Write a short summary at the end of each chapter.

# Evaluation, Conclusions and Future Work

## Project Objectives

Summarise what you have achieved.

## Evaluation

Stand back and evaluate what you have achieved and how well you have met the objectives. Evaluate your achievements against your objectives in section 1.2. Demonstrate that you have tackled the project in a professional manner.

(The previous paragraph demonstrates the use of automatic cross-references: The ‘1.2’ is a Cross-reference to the text in a numbered item of the document, it is not literal text but a field. The number that appears here will change automatically if the number on the referred-to section is altered, for example if a chapter or section is added or deleted before it. Cross-references are entered using Word's **Insert** or **References** menu. Cross-references are set to update automatically when printed, but may not do so on-screen beforehand; you can update a field manually on-screen by right-clicking on it and selecting Update field from the pop-up menu or by selecting the whole document and pressing F9.)

## Applicability of Findings to the Commercial World

Summarise what you have achieved.

## Conclusions

Summarise what you have achieved.

## Future Work

Explain any limitations in your results and how things might be improved. Discuss how your work might be developed further. Reflect on your results in isolation and in relation to what others have achieved in the same field. This self-analysis is particularly important. You should give a critical evaluation of what went well, and what might be improved.

## Concluding Reflections

Summarise what you have achieved.

# Bibliography

Ali, Z. (2016) *'A Framework for Game Engine Selection for Gamification and Serious Games',* San Francisco, IEEE, pp. 1199-1207.

Bach, J. (1998) 'A Framework for Good Enough Testing', *Computer,* 31(10), pp. 124-126.

Finch, M. (2007) *Effective Water Simulation from Physical Models,*   
http://http.developer.nvidia.com/GPUGems/gpugems\_ch01.html  
(accessed 9 April 2017).

Gregory, J. (2012) *'Game Engine Architecture',* 2nd edition, Natick: A K Peters, Ltd..

Hello Games, (2016) *'No Man's Sky',* Guildford: Sony Interactive Entertainment.

Horn, B. et al. (2014) *'A Comparative Evaluation of Procedural Level Generators in the Mario AI Framework',* Lauderdale, Society for the Advancement of the Science of Digital Games, pp. 1-8.

Jacobson, I., Spence, I. & Seidewitz, E. (2016) 'Industrial-Scale Agile--From Craft to Engineering', *Communications of the ACM,* 59(12), pp. 63-72.

Jiang, B. & Chan, W. K. (2016) *'Testing and Debugging in Continuous Integration with Budget Quotas on Test Executions',* Vienna, IEEE.

Julian, T., Preuss, M., Beume, N. & Wessing, S. (2013) 'Controllable Procedural Map Generation via Multiobjective Evolution', *Genetic Programming and Evolvable Machines,* 14(2), pp. 245-277.

Microsoft, (2015) *DLLs in Visual C++,*   
https://msdn.microsoft.com/en-us/library/1ez7dh12.aspx  
(accessed 25 03 2017).

Microsoft, (2015) *Walkthrough: Creating and Using a Static Library (C++),*   
https://msdn.microsoft.com/en-us/library/ms235627.aspx  
(accessed 25 03 2017).

Mikuličić, N. & Mihajlović, Ž. (2016) *'Procedural Generation of Mediterranean Environments',* Opatija, Information and Communication Technology, Electronics and Microelectronics (MIPRO).

Neumann, P. G. (2016) 'Risks of Automation: A Cautionary Total-System Perspective of Our Cyberfuture', *Communications of the ACM,* 59(10), pp. 26-30.

Nystrom, R. (2014) 'Game Programming Patterns', in *Game Programming Patterns,* USA: Genever Benning, pp. 123-139.

Onal, E. et al. (2014) *'Decision-making in abstract trust games: A user interface perspective',* New Orleans, Cognitive Methods in Situation Awareness and Decision Support (CogSIMA).

Pelzer, K. (2007) *Rendering Countless Blades of Waving Grass,*   
http://http.developer.nvidia.com/GPUGems/gpugems\_ch07.html  
(accessed 9 April 2017).

Pereira, D., Zárate, L. & Song, M. (2013) *'Applying Formal Concept Analysis to Assist Class Hierarchy Construction in Biomedical Systems',* Hangzhou, IEEE.

Perlin, K. (2002) 'Improving Noise', *ACM Transactions on Graphics,* 21(3), pp. 2-3.

Whittaker, J. (2011) *The 10 Minute Test Plan,*   
https://testing.googleblog.com/2011/09/10-minute-test-plan.html  
(accessed 3 April 2017).

Wilson, J. M. (2003) 'Gantt charts: A Centenary Appreciation', *European Journal of Operational Research,* 149(2), pp. 430-437.

Zachariah, B. (2015) 'Optimal Stopping Time in Software Testing Based on Failure Size Approach', *Annals of Operations Research,* 235(1), pp. 771-784.

Appendix 1 – Project Gantt Chart

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Task Mode | Task Name | Duration | Start | Finish | Predecessors |
| Auto Scheduled | Setup git project | 0 days | Thu 01/09/16 | Thu 01/09/16 |  |
| **Manually Scheduled** | **Initial program setup** | **12 days** | **Thu 01/09/16** | **Thu 22/09/16** | **1** |
| Auto Scheduled | Create Window | 1 day | Thu 01/09/16 | Sun 04/09/16 |  |
| Auto Scheduled | Set up swap and chain | 2 days | Sun 04/09/16 | Wed 07/09/16 | 3 |
| Auto Scheduled | Set up remaining areas of Direct X | 4 days | Wed 14/09/16 | Wed 21/09/16 | 4 |
| Auto Scheduled | Create a logger | 1 day | Wed 21/09/16 | Wed 21/09/16 |  |
| **Manually Scheduled** | **Create engine** | **10 days** | **Thu 22/09/16** | **Sun 09/10/16** | **2** |
| **Manually Scheduled** | **Create primitives** | **2 days** | **Thu 22/09/16** | **Mon 26/09/16** |  |
| Auto Scheduled | Create triangle primitive | 1 day? | Thu 22/09/16 | Sun 25/09/16 |  |
| Auto Scheduled | Create cube primitive | 1 day? | Thu 22/09/16 | Sun 25/09/16 |  |
| Auto Scheduled | Implement polymorphism for primitives | 1 day | Sun 25/09/16 | Mon 26/09/16 | 9,10 |
| **Manually Scheduled** | **Create pixel and vertex shaders** | **2 days** | **Mon 26/09/16** | **Thu 29/09/16** | **8** |
| Auto Scheduled | Create solid colour pixel shader | 1 day? | Mon 26/09/16 | Wed 28/09/16 |  |
| Auto Scheduled | Create texture pixel shader | 1 day? | Mon 26/09/16 | Wed 28/09/16 |  |
| Auto Scheduled | Create texture and diffuse lighting pixel shader | 1 day? | Wed 28/09/16 | Wed 28/09/16 |  |
| Auto Scheduled | Implement loading of models | 2 days | Thu 29/09/16 | Mon 03/10/16 |  |
| Auto Scheduled | Implement camera and model control | 3 days | Mon 03/10/16 | Sun 09/10/16 | 16 |
| **Manually Scheduled** | **Procedurally generate level** | **26 days** | **Sun 09/10/16** | **Wed 23/11/16** | **7** |
| **Auto Scheduled** | **Generate height map** | **8 days** | **Sun 09/10/16** | **Mon 24/10/16** |  |
| Auto Scheduled | Create perlin noise maps | 5 days | Sun 09/10/16 | Wed 19/10/16 |  |
| Auto Scheduled | Compile perlin noise maps together | 3 days | Wed 19/10/16 | Mon 24/10/16 | 20 |
| **Manually Scheduled** | **Apply terrain properties to areas of height map** | **5 days** | **Mon 24/10/16** | **Tue 01/11/16** |  |
| Auto Scheduled | Create basic textures to be positioned height map | 4 days | Mon 24/10/16 | Mon 31/10/16 |  |
| **Manually Scheduled** | **Testing** | **16 days** | **Sun 27/11/16** | **Sun 25/12/16** | **2,7,18** |
| Auto Scheduled | Write test script | 1 day? | Sun 27/11/16 | Mon 28/11/16 |  |
| Auto Scheduled | Write unit tests | 10 days | Mon 28/11/16 | Wed 14/12/16 | 25 |
| Auto Scheduled | Run unit tests | 1 day? | Wed 14/12/16 | Sun 18/12/16 | 26 |
| Auto Scheduled | Conduct manual testing | 4 days | Sun 18/12/16 | Sun 25/12/16 |  |
| **Manually Scheduled** | **Procedurally generate terrain elements** | **21 days** | **Wed 28/12/16** | **Wed 01/02/17** | **22** |
| Auto Scheduled | Generate grass | 5 days | Wed 28/12/16 | Wed 04/01/17 |  |
| Auto Scheduled | Generate rocks | 5 days | Thu 05/01/17 | Sun 15/01/17 |  |
| Auto Scheduled | Generate trees | 5 days | Sun 15/01/17 | Mon 23/01/17 |  |
| Auto Scheduled | Generate sand | 5 days | Mon 23/01/17 | Tue 31/01/17 |  |

Appendix 2 – Engine Class Diagram



Figure 9‑1 Engine Class Diagram

Appendix 3 – Title of Appendix

You may have one or more appendices containing detail, bulky or reference material that is relevant though supplementary to the main text: perhaps additional specifications, tables or diagrams that would distract the reader if placed in the main part of the dissertation. Make sure that you place appropriate cross-references in the main text to direct the reader to the relevant appendices.

Do not blindly include all of your code in the appendix or the body. Only include the parts you refer to in the report. You can but those parts either in the appendix or in the body (e.g. in the “Implementation” part).