***Figure 1 – Game Winning Screen***



C++ Role Playing Game

Richard Came

13030333

Supervisor – Diego Perez

Second Assessor – Michael Gardner

Computer Science BSc

# Acknowledgements

I would like to thank my supervisor Diego Perez for his efforts in keeping me on track and providing insight into technical problems that I was facing throughout the project. I would also like to thank the many lecturers from the last few years at University that have provided such informative and interesting lectures.

# Abstract/Summary

The project is to create a C++ Role Playing Game. The game is a 2-Dimentional fantasy Role Playing Game that is created using the C++ language and the SDL2 (Simple DirectMedia Layer 2) library. The game is designed to have a retro feel with pixelated graphics and simple real time combat system which is based around a levelling system.

The games terrain is completely procedurally generated, this will allow for greater re-playability as each time a new game is started it will be different to the last.

The objective of the game is to complete the quests that you are given. Some quests will be given to you at the start of the game, and others will be found while you play the game. Quests will range from locating treasure to defeating monsters. Once all quests have been completed the game is completed and the player wins.

Players level up their character by defeating monsters and completing quests. The monsters and quests will scale in difficulty with the players level, so that the game is always a challenge.

Contents

[Acknowledgements 1](#_Toc448848024)

[Abstract/Summary 2](#_Toc448848025)

[Acronyms and Defintions 2](#_Toc448848026)

[Literature 3](#_Toc448848027)

[Literature Survey 3](#_Toc448848028)

[Procedural Content Generation 3](#_Toc448848029)

[Pathfinding 3](#_Toc448848030)

[Legal 4](#_Toc448848031)

[Project Goals 5](#_Toc448848032)

[Project Aims 5](#_Toc448848033)

[Project Objectives 6](#_Toc448848034)

[PRIMARY Objectives 6](#_Toc448848035)

[SECONDARY Objectives 8](#_Toc448848036)

[Technical Achievement 10](#_Toc448848037)

[Introduction 10](#_Toc448848038)

[Development 10](#_Toc448848039)

[Introduction 10](#_Toc448848040)

[Development Environment 10](#_Toc448848041)

[Development Language and Libraries 10](#_Toc448848042)

[Game Structure 10](#_Toc448848043)

[User Interface 14](#_Toc448848044)

[In Depth Analysis 15](#_Toc448848045)

[Introduction 15](#_Toc448848046)

[Representation of and Rendering of Textures and Fonts 15](#_Toc448848047)

[Game World Generation 15](#_Toc448848048)

[Non-Player Characters 15](#_Toc448848049)

[Something to do with the way the game works 15](#_Toc448848050)

[Project Planning 17](#_Toc448848051)

[Conclusions 18](#_Toc448848052)

[References 19](#_Toc448848053)

# Acronyms and Defintions

The following provides definitions for terms and acronyms used in this report:

SDL2 – Simple DirectMedia Layer version 2

NPC – Non-player character

Replayability – The amount of enjoyment that is received after playing a game multiple times, the higher the replayability the more times you can play a game while still enjoying it.

IDE – Integrated Development Environment

CLion – An IDE produced by JetBrains

IntelliJ IDEA – An IDE produced by JetBrains

CMake – A cross-platform free and open-source software for managing the build process of software using a compiler-independent method.

UI – User Interface

AI – Artificial Intelligence

# Literature

## Literature Survey

This project has mainly tried to solve the same problem that almost all games try to solve, and that is to solve the problem of boredom and the general tedium that is everyday life. The project tries to extend on the solutions to this problem that have been presented by other games in the past by trying make use of procedural content generation so that the game can be re-played many times while still providing a fresh experience to the player. This, it is hoped will help solve the problem for a larger period of time compared to a game that only has static content and is the same every time it is played.

This problem of boredom has been one that has existed for as long as humans have had free time, and probably even before that. It is fundamental in our drive to make use of our time in new and exciting ways, however sometimes in our modern lives there is nothing to do and our boredom can run wild resulting in unhappiness. This is where games can come in, they can fit into someone’s daily life and fill the spots where there is nothing else to do and provide some much needed excitement or pleasure. This can be hard to find in other mediums where you’re not actually partaking in the action, like films or books.

### Procedural Content Generation

The vast majority of the project was solving the problem of procedurally generating the games terrain in a way that looked realistic while providing enough features to make it interesting to both the eye and from a gameplay perspective. To this end perlin noise [1] was essential in creating the terrain of the game, the continuous noise that is produced by the noise function allowed for a realistic looking terrain to be created. The noise produced can then be manipulated to create a terrain that suits the specific requirements.

### Pathfinding

The game makes use of pathfinding to create roads, as well as making use of it to move the NPCs (non-player character) around the map. As the game uses a simple 2D grid based system the pathfinding used was A\*. [2] A singular general pathfinding class was created for the game, this was then used by both the NPC characters and for placing roads on the terrain.

## Legal

Fair use and how to use references

Use references in this section to increase references, some referenception is required

# Project Goals

## Project Aims

The main aims of the project are to create a fun and immersive gaming experience that has a large amount of replayability. This will be achieved by using procedural content generation to create a unique experience every time the player starts a new game, by generating a new game world each time. Alongside this the game should make use of procedural content generation to create quests and objectives for the player that make use of the games map so that the game has different quests each time making the player feel like they are in a completely different world compared to the last.

The game should also contain items, weapons and armour that provide bonuses for the player character and allow an extra element of strategy. These items should be created procedurally so that the player has different items to use rather than using the same items in each game they play. These items should be purchasable in shops in the game, as well as found as treasure and achieved as rewards for completing quests.

The game should have NPCs that the player can interact with; these characters will provide the player friendly interaction as well as a combat enemy. The two main types of NPCs will be the friendly characters that the player can interact with and talk to and the enemy characters that the player can fight with.

The game should have a fluid combat system, where the player can fight the enemy NPCs. The combat system should be designed around the fighting parties’ statistics, where the player’s level and items denote how much attack and defence they have and the enemy characters attack and defence is denoted by their level and status.

The game should be displayed using a 2D overhead world view where the characters, buildings and certain terrain features such as trees are the only objects that appear to be in sudo-3D.

The game should run well without any bugs that cause it to crash.

## Project Objectives

The project objectives are as laid out in the Interim Report [3] the final state of these objectives will be given below, including any changes that were made to them since the Interim Report.

### Primary Objectives

The primary objectives are the objectives that are essential to the game, and are deemed the highest priority.

#### 1 - PROCEDURALLY GENERATED TERRAIN - Completed

**Description**:

The games terrain should be generated procedurally using Perlin Noise [1] to create sections of terrain within a blank map. These sections will be evaluated using an evaluation function to make sure that the terrain is suitable; if it is not suitable it will then be re-generated.

**Changes**:

The terrain is created in one go, rather than in sections, when the original objective was created the theory behind generating the terrain was not fully understood and as such this was changed when a better understanding was achieved.

There is no longer a requirement for an evaluation function as the way the terrain is generated a suitable terrain is created every single time, this was found to be the case after extensive testing.

#### 2 - PROCEDURALLY GENERATED STRUCTURES AND TOWNS – partially compete

**Description**:

Structures and towns should be generated using pre-made pieces that are slotted together; these generated structures should then be evaluated to make sure that they are suitable using an evaluation function.

**Changes**:

Due to time constraints the generation of structures was put on hold as it was not deemed essential to the overall game, and instead towns are now represented by a single graphic and then placed onto the map at different locations.

#### 3 - PLAYER CHARACTER IMPLEMENTATION – Partially Complete

**Description**:

The player character should be controlled by user input and should be customisable in both the way that they look and play. There should be an editor that allows the player to make cosmetic changes to their character.

**Changes**:

The player character’s movement and controls have been fully implemented, however due to time constraints the character customisation was omitted due to it being deemed a non-essential feature.

#### 4 – NON-PLAYER CHARACTERS (NPC’S) AI – Partially Complete

**Description**:

The NPC’s AI depends on their purpose, enemy NPCs should have pathfinding and combat features but do not need communication features, whereas friendly characters do not need combat features but do require communication features.

**Changes**:

The NPC’s pathfinding and combat features have been implemented fully; however the communications features have not been implemented as no friendly characters have been implemented.

#### 5 - PROCEDURALLY GENERATEd Characters – Partially Completed

**Description**:

Characters in the game should be procedurally generated; as such their looks and statistics should be generated procedurally. Friendly characters should look friendly, whereas enemy characters should look like monsters.

**Changes**:

The cosmetic element of the character generation has been dropped due to time constraints.

#### 6 - COMBAT MECHANICS AND HIT Detection – Partially Completed

**Description**:

Combat mechanics should consist of both ranged and melee attacks, where ranged attacks are used at a distance and melee attacks are used at close range.

The hit detection will work so that if an attack should hit a character, it does hit that character.

**Changes**:

The ranged aspect of combat has been dropped due to time constraints.

### Secondary Objectives

The secondary objectives are the objectives that are deemed as useful in creating a fun game, however they are not essential to making a functional product.

#### 1 - LEVELING SYSTEM - Completed

**Description**:

All characters in the game will have a level, this directly translates to the strength of the characters and the abilities that the character can have. The level of all characters is linked to the players level, when the player levels up all of the other characters in the game get stronger with the player.

**Changes**:

No changes.

#### 2 - PROCEDURALLY GENERATED ITEMS – Not Implimented

**Description**:

Items in the game should be created based on a pool of possibilities, such as abilities, and variable statistics. These items should grow in strength with the player as the player levels up. Visually each item should be generated using different components that are joined together, once joined together the item can then be tinted to indicate different levels or rarities.

**Changes**:

No changes.

#### 3 – MULTIPLAYER – Not Implimented

**Description**:

“The games multiplayer will consist of up to four player characters. One player will act as the host server and the other players will connect to them. The number of human players is decided when the game is first started, and that save game will only be playable when all the players are connected. The complexity lies in making a system that sends all of the relevant information and can cope with data loss. This will be achieved by sending information only when something in the game changes, each update will be small in size and contain a check-sum so that the other player’s clients know that the information received is correct.” [3]

**Changes**:

No changes.

#### 4 - PROCEDURALLY GENERATED QUESTS – Partially Implimented

**Description**:

“Quests will be generated using the terrain as a basis, meaning that locations that are suitable for a quest objective will be found when the terrain is generated and quests will be created at these locations. There will be a number of different types of quests, including but not limited to; killing enemies at a location, find item(s) at a location and escort someone to somewhere.” [3]

**Changes**:

Due to time constraints procedurally generated quests were not implemented, however part of the way through development it was decided that they are required to give the player some form of objective. Therefore quests were implemented at a basic level, where they are not procedurally generated rather there is a small amount of pre-made quests available to the player when they start to play the game.

# Technical Achievement

## Introduction

This section will provide the main body of this report and will describe the processes that went into creating the game as well as the key decisions that were made during the games development. It will also cover the areas of the development cycle that were particularly challenging.

All of the project code and materials can be found at the following link on GithHub –

**https://github.com/rcame/IndividualProject/**

## Development

### Introduction

The development section will outline the processes that were involved in developing the game as well as describe the environment and libraries that were used during the development process.

### Development Environment

Throughout the development process the JetBrains IDE CLion was used. The CLion IDE environment is very similar to the JetBrains IntelliJ IDEA environment which is an easy to use and powerful IDE. As CLion is very similar to IntelliJ IDEA it is very accommodating to those that have used IntelliJ IDEA in the past and as such was a good choice of IDE.

### Development Language and Libraries

#### Language and Language Tools

As the title of the project suggests the programming language used to make the game is C++ 11. To accompany the language, the CMake build system was used to help with the development process; the version of CMake used was 3.3.

#### Development Library

The game is built using the SDL2 [4] (Simple DirectMedia Layer version 2) development library as a tool to provide access to the low level hardware. This is required to allow the games visuals, user inputs and general game behaviour to be easily manipulated within the same framework.

### Game Structure

The game is based around a structure where initialisation is completed and then a main loop is started and continues until the game is closed. Firstly all of the games media is loaded, then the game is generated, this includes all of the terrain and NPCs, after the game has been generated the main loop is started. The main game loop handles all of the user inputs and passes them to the relevant classes, as well as calling all of the updates. Once the updates have been completed, such as NPC movement, the game is then rendered to the window, and then the loop repeats.

***Figure 2 - Game Structure***

Load Media

Generate Game Content

Start Game Loop

Handle User Input

Handle Game Updates

Render the Game to the Window

#### Loading the Games Media

The game uses a number of textures, as well as a range of fonts. These are all loaded during the initialization phase of the game. That is to say that every piece of media that the game needs is loaded from the start, this is done because loading this during the game could affect the performance and cause a loss of frame rate. The second reason that all of the media is loaded at the start is because the media itself is small and as such does not use a lot of memory; this is due to the fact that the game uses low resolution textures as a base and then combines them together and re-uses them to create its visuals.

#### Game World Generation

The game world is generated in four main stages, in the first stage the base terrain is created, in the second stage towns are placed on the map, the third is to add roads between the towns and lastly trees and waves are placed on the map to add some life to it.

During the first stage the base terrain is created, the terrain is essentially a 2-Dimentional array of integer values that correspond to textures. During the terrain generation stage the array of values is created using Perlin Noise [1] as the initial input, the noise values are then assigned to a texture based on a set of ranges. This is manipulated so that at the edges of the map, ocean is created and following that beaches. This is achieved by using two methods in combination; the first is the application of a gradient that gives the values at the edge of the map a bias towards being lower in value and therefore creating oceans and beach areas. The second method is to make it so that oceans and beaches can only be produced within a certain range of the edge of the map, meaning a singular large island is always created, with the possibility of small islands being created at the edges of the map.

The second stage adds towns to the map, there are always four main towns placed on the map, one in each quarter. Each town is placed in its own unique environment; the different environments of each town are; a town surrounded by water, a town in the desert, a town on top of a mountain and a town in the middle of the plains. The way the towns are placed is by looking over the terrain and each time a spot that is suitable is found a random number is generated and if the number is above a threshold the town is placed. This process is repeated until the correct number of towns has been placed.

Roads are added to the map during the third stage. To add roads to the map, a pathfinding algorithm is used to find the best route between two towns, once the route has been found the road is placed along it. The pathfinding algorithm takes note of the terrain that the road is traversing, such that it prefers to go around mountains and hills. The algorithm will choose to go over bodies of water rather than go around them, and when the road is placed on water a bridge is added to the map instead of the pavement texture.

The fourth stage is to place trees and waves around the map. Trees are placed by going over each suitable tile and generating a random number, if the number is higher than a threshold value a tree is placed. The threshold value depends on the terrain tile being evaluated, for example, the bases of mountains are densely populated whereas plains are sparsely populated. Trees are also placed in such a way that if a tree has been placed in a tile it is more likely that a tree will be placed in an adjacent tile, this is to create the effect of forests around the map. There are also multiple textures for trees; these textures correspond to certain types of terrain so that the trees match their environment.

Waves are placed on ocean tiles in the same way that trees are placed on ground tiles, that is to say that each ocean tile is evaluated and a random number is generated, if the number is over a threshold value a wave is placed on the map. As with trees if a tile has a neighbouring tile with a wave on it, the threshold is lowered creating a system where large waves can be created.

#### NPC Generation

After the game world has been generated NPCs are the next stage of generation. There is only one type of NPC in the game, and these are the enemy characters. There are a fixed number of these characters created at the start of the game and the number is based on the size of the game world. This is done so that there are always enough enemies to fill the world and make conflict necessary, while not overpopulating it. There are two types of enemy characters, the blob character and the slime character, and depending on the terrain one or the other is created, slimes are created on beaches and mountains, and blobs are created on plains. To decide where they are spawned, a random tile on the terrain is chosen and if it is a suitable terrain type for one of the characters that character is spawned there, if it is not another tile is chosen. This repeats until the correct number of characters is created.

#### Handling User Input

User input is handled by the SDL2 library, each time an input is detected it is pushed onto the event stack, and each event is then passed to the updated class to be handled. The update class then detects if the input is a valid command, if it is then the correct action is take, such as if the ‘W’ key is pressed, the players movement changes to up.

#### Updating game States

The first game state that is updated, is the players position, this is only done if the user has entered a movement command. If the player is to move, the states required for the movement are updated.

The second game state that is updated are the players current quests, the relevant states for the quests are passed and if there are any changes the quests are updated.

The third game state that is updated are the NPC’s positions. The NPC’s positions are only changed if they are within the player’s sight, if they are within the player’s sight they are updated according to their pathfinding algorithm.

Finally the player’s health is regenerated, the player’s health is regenerated every few seconds and as such a check is made to make sure that the regeneration delay is completed. This delay can change based on in game factors, such as an increase in character level.

#### Rendering the Game

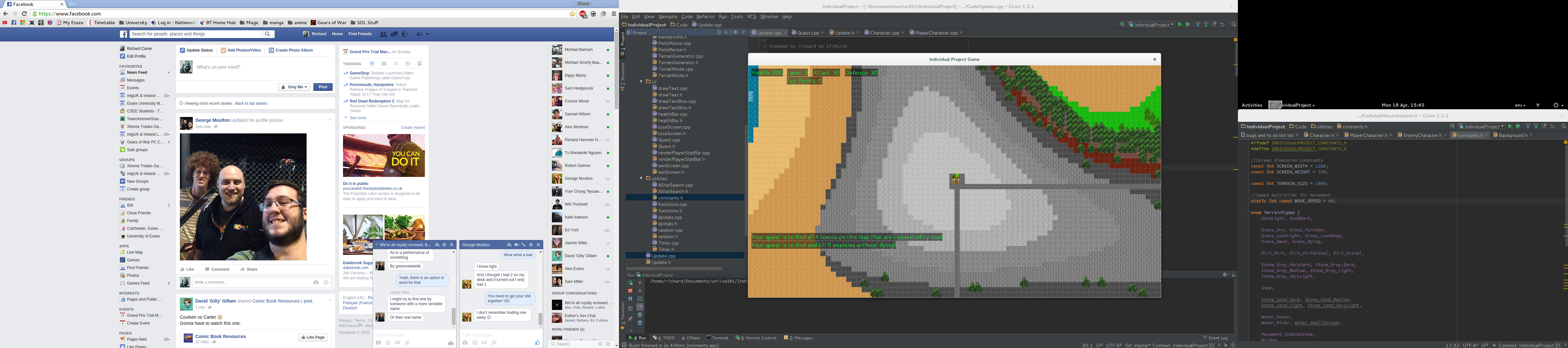
The game is rendered at the end of each iteration of the main game loop. This is always done last to make sure that all of the updates have been completed. If this was not done last irregularities could become present, such as some characters being in the wrong position.

When the game is rendered, the terrain is rendered first, and then all of the NPCs are rendered, and then the player character, and finally the user interface elements are rendered. The rendering is done in this order so that the correct elements are shown above and are always visible, for instance if the UI was rendered first, it would be completely covered over by the terrain.

### User Interface

The user interface shows the player characters statistics; health, level, attack, defence and experience points. The interface also shows the players current quests that need to be completed.

***Figure 3 – Showing the User Interface***



## In Depth Analysis

### Introduction

This section will provide a detailed technical analysis of the systems that are within the game, as well as a detailed analysis of how they fit together to make the final product.

### Representation of and Rendering of Textures and Fonts

At the base level the SDL2 library handles the rendering of textures and fonts, however for ease of use and to increase the robustness a wrapper class is used for textures and pre-defined wrapper functions are used for fonts.

#### The Textures Wrapper Class

The texture wrapper class keeps track of the basic elements of a texture, including the pixel width and pixel height of the texture, which is stored as an integer. A pointer to the raw pixels of the texture is also stored, along with the pitch of the texture, which is required if the texture it to be manipulated on a per pixel level. Pixel manipulation is not actually used in the game, however originally this was intended to be possible so that the player could edit the cosmetics of their character. This functionality was dropped due to time constraints and the fact that it was deemed as non-essential to the game.

***Figure 3 – Texture Class Diagram***

Texture

- SDL\_Texture\* mTexture

- void\* mPixels

- int mPitch

- int mWidth

- int mHeight

+ Texture()

+ ~Texture()

+ loadFromFile(string) bool

+ loadFromRenderedText(string, SDL\_Color, TTF\_Font) bool

+ createBlank(int, int, SDL\_TextureAccess) bool

+ render(int, int, SDL\_Rect, double, SDL\_Point, SDL\_RendererFlip)

+ setAsRendererTarget()

+ clearTexture()

+ free()

+ setColor(Uint8, Uint8, Uint8)

+ setBlendMode(SDL\_BlendMode)

+ setAlpha(Uint8)

+ getWidth() int

+ getHeight() int

+ lockTexture() bool

+ unlockTexture() bool

+ getPixels() \*

+ copyPixels(\*)

+ getPitch() int

+ getPixel32(unsigned int, unsigned int) Uint32

##### Creating a New Texture

To create a new texture, a Texture object must be created. Once the object has been created there are three options available for creating a new texture, the first option is to pass a string path of an image to the ‘loadFromFile’ method. Another method is to create a blank texture of a specified size, using the ‘createBlank’ method. The third option is to create a texture from a string of text using the ‘loadFromRenderedText’ method; this method is used by the UI elements.

The ‘loadFromFile’ method creates a base texture within the SDL2 framework from the string path to an image that is passed to it and assigns values to all of the elements within the Texture class.

The ‘createBlank’ method creates a blank SLD2 texture of the given dimensions, and then assigns the class width and height values to be the given dimensions.

The ‘loadFromRenderedText’ method creates a texture that is a string of text with a transparent background, the text is coloured in the supplied colour and is written in the supplied font. The size of the texture is denoted by the font.

##### Rendering the Texture

To render the texture to the window, or to the current render target, the ‘render’ method is used. This method takes the pixel position of where the texture is to be rendered on the current render target, and the dimensions that the texture should be rendered with, if none are supplied the textures native dimensions are used. The method also has elements for flipping and rotating the texture, if the texture is to be rotated or flipped the method can take an angle of rotation, the point around which the texture is to be rotated and how the texture is to be flipped, if it is not to be flipped a null value is used.

Once the render method has been called and all of the required elements supplied, the SDL2 texture render function is called, and this then renders the texture to the current render target.

#### UI and Font Wrapper Functions

To games UI utilizes a number of different functions in combination to create a text based UI that provides the player with all of the information that they need. Each of the elements in the UI uses a text box function; this function creates a semi-transparent black background behind the text of the element. This background is created so that the text can be read over any in game element with the game below still visible.

To draw a UI element, the ‘drawText’ function is used.



This function creates a Texture object that uses the ‘loadFromRenderedText’ method to create the texture from the supplied string. After the texture has been created the ‘drawTextBox’ function is called, and this creates the background of the UI element as described above. The UI element is then rendered to the window at the supplied coordinates.

Extensions of the ‘drawText’ function have been created to create specific reusable elements of the UI, such as the ‘winScreen’ function, this displays the words “You Win!” in large letters over the middle of the screen, as shown in figure 1. Other functions that perform similar functions include the functions: ‘healthBar’, ‘loseScreen’ and ‘renderPlayerStatBar’. The Quest class also makes use of these function, this will be explained in greater detail further in ‘Quests’ section of the report.

### Game World Generation

The game world is generated in four main stages; this section shall detail this process.

***Figure 4 – TerrainGenerator Class Diagram***

TerrainGenerator

- int \*\*terrain;

- int \*\*terrainDetail

- int const MAX\_OCEAN\_LINE

- int const MAX\_BEACH\_LINE

- int const TERRAIN\_SCALE

- int numTowns

- enum TownType

+ TerrainGenerator(int\*\*, int\*\*)

+ getTownLocations () int\*\*

+ getNumTowns() int

- generateTerrain()

- generateOcean(int, int, int, double) int

- generateLand(double) int

- placeTowns()

- decideRoads()

- placeRoads(shared\_ptr<TerrainNode>)

- placeTrees()

- placeWaves()

#### Terrain Generation

The first stage of world generation is to create the base terrain, as all other elements of the world are built upon this and use the base terrain as part of the generation algorithms. The terrain for the world is stored in a 2-Dimentional array of integers, each integer corresponds to a specific texture, and these textures are defined in an enum called ‘terrainChooser’. An array of integers is used as it requires a smaller amount of space, and is simpler to access when compared to an array of objects. Using an enum to assign the integer values in the array means that it is easy to tell which texture each cell in the array refers to.

To populate the array Perlin Noise [1] is used, this is seeded with the current time to make sure that the pattern of noise produced is unique. To get the noise values, a for loop is used, this loop creates an x and a y coordinate for each cell in the array, the values produced are then divided by the ‘TERRAIN\_SCALE’ constant and passed to the noise function and a noise value is returned. The ‘TERRAIN\_SCALE’ value is a constant integer that was achieved by thorough testing of the terrain that the noise produced, if the value was too low, the terrain produced was too sparse, if it was too high there were too many features in too small of an area.

Once the noise value has been generated, it is passed to the relevant generation function; there are two generation functions; ‘generateOcean’ and ‘generateLand’. If the cell in question is within a certain range of the edge of the world, the ‘generateOcean’ function is called; otherwise the ‘generateLand’ function is called. The range of the edge of the world is decided by the ‘MAX\_BEACH\_LINE constant; this constant was achieved by thorough testing and many generations of terrain.

The ‘generateOcean’ function creates the terrain at the edge of the map, and is tasked with ensuring that the main landmass is always surrounded by oceans and beaches.This is required because the game world is designed to be a singular island to give the player a natural barrier that they cannot pass. The same effect could be achieved by making the edge of the map be unpassable mountains, however it was decided that making the game world an island was more natural. The ‘generateOcean’ function ensures that the edge of the map is always surrounded by oceans and beaches by using a gradient to give a bias towards creating ocean and beach tiles. The gradient is created based on the proximity of the current cell to the closest edge of the map; this value is then divided by the ‘MAX\_OCEAN\_LINE’ constant. By using a gradient to manipulate the value used to determine the type of terrain a depression is created around the edge of the map, causing oceans and beaches to be artificially created all of the way around the edge of the map.

The final value that is attained once the gradient is applied to the original noise value is then passed to a series of if statements that decided what type of tile is to be used based on a series of ranges. If the value is below 0.3 an ocean tile is created, if the value is a between 0.3 and 0.5 beach tiles are created, and above this the original value without the gradient applied is passed to the ‘generateLand’ function. Using this technique a natural looking island is created.

The ‘generateLand’ function works by taking the noise value and assigning a texture to it by using a set of ranges in a switch case statement. This is a simple method that works very effectively to create a terrain based on the fact that the noise values can be used as a height map. As such, the higher the noise value generated is, the higher the altitude that the texture assigned is associated with, for instance below 0.56 is grassy terrain and above that is mountainous terrain. Unlike the ‘generateOcean’ function, the generateLand’ function is not able to assign ocean or beach tiles; this is to make sure that there is only one large island created rather than multiple smaller islands.

Creating the island and terrain was by far the most complex part of creating the game.

#### Town Placement

The second stage adds towns to the map, there are always four main towns placed on the map, one in each quarter. Each town is placed in its own unique environment; the different environments of each town are; a town surrounded by water, a town in the desert, a town on top of a mountain and a town in the middle of the plains. The way the towns are placed is by looking over the terrain and each time a spot that is suitable is found a random number is generated and if the number is above a threshold the town is placed. This process is repeated until the correct number of towns has been placed.

#### Road Placement

Roads are added to the map during the third stage. To add roads to the map, a pathfinding algorithm is used to find the best route between two towns, once the route has been found the road is placed along it. The pathfinding algorithm takes note of the terrain that the road is traversing, such that it prefers to go around mountains and hills. The algorithm will choose to go over bodies of water rather than go around them, and when the road is placed on water a bridge is added to the map instead of the pavement texture.

#### Tree and Wave Placement

The fourth stage is to place trees and waves around the map. Trees are placed by going over each suitable tile and generating a random number, if the number is higher than a threshold value a tree is placed. The threshold value depends on the terrain tile being evaluated, for example, the bases of mountains are densely populated whereas plains are sparsely populated. Trees are also placed in such a way that if a tree has been placed in a tile it is more likely that a tree will be placed in an adjacent tile, this is to create the effect of forests around the map. There are also multiple textures for trees; these textures correspond to certain types of terrain so that the trees match their environment.

Waves are placed on ocean tiles in the same way that trees are placed on ground tiles, that is to say that each ocean tile is evaluated and a random number is generated, if the number is over a threshold value a wave is placed on the map. As with trees if a tile has a neighbouring tile with a wave on it, the threshold is lowered creating a system where large waves can be created.

### Non-Player Characters

### Something to do with the way the game works

# Project Planning

# Conclusions

# References

|  |  |
| --- | --- |
| [1] | K. Perlin, “An Image Synthesizer,” *SIGGRAPH,* vol. 19, no. 3, pp. 287 - 296, 1985. |
| [2] | D. Perez, *High-Level Games Development lecture Slides,* Colchester: University of Essex, 2015/16. |
| [3] | R. Came, *Computer Game - Your Own Idea Interim Report,* Colchester, 2015. |
| [4] | “SDL - Simple Directmedia Layer,” [Online]. Available: https://libsdl.org. [Accessed 17th April 2016]. |