# Final Year Project Dissertation

Conor Rabbitte, Darren Butler

October 2020

GitHub Repository: https://github.com/dnbutler64/FinalYearProject

# Contents

# List of Figures

# 1 Introduction

When choosing our project we wanted to pick something that would challenge our computer science and problem solving skills, as well as engage our interests. We wanted to explore the game development process and build a fun game we would play ourselves. Having worked together before on our third year project, we were excited at the prospect of working together again and developing our skills further as a team. With this in mind, we set out to brainstorm our ideas for the project.

We both share a passion for games and game development. A particular favourite genre of ours is 'Roguelike'. These games use procedural generation to dynamically build entire worlds each time the game is played. The objective of a Roguelike game is usually to fight and survive as far into the dungeon as possible before, meeting an untimely end. The procedural generation and randomness that these games are built on, provides an evergreen experience that is always a little different each time you play.

For our game we wanted to take the core Roguelike mechanics. Namely, procedurally generated content and A.I. controlled characters. We also wanted to incorporate our own unique twist on the classic Roguelike. To make our game stand out, we intended to develop a Roguelike that can be played cooperatively online with friends. We also aimed to reproduce other favourite game mechanics of ours, such as an inventory system and character customisation. We are acutely aware that this is first and foremost a software development project. To this end, we decided to focus almost entirely on the development of core gameplay systems and features. Some elements, like sound design, art assets, animations, and storytelling, although nice to have and essential for our game to be consumer ready, were not our primary focus.

We looked forward to the challenge of learning new technologies and computer science concepts, as well as collaborating to achieve something we could not do as individuals. We hoped to build a game that we would enjoy to play and be proud to have created.

## 1.1 Context

Our project centres around game development. Specifically, the mechanics and techniques used within the discipline to create immersive experiences for players. Our game will feature procedurally generated worlds, populated with A.I. driven enemies. Players will be able to connect and play with each

other over the internet. These core mechanics will lead to exciting and varied gameplay for the players.

## 1.2 Objectives

The primary objective of this project is to research and explore specialist areas of computing, to combine these technologies and produce an application that adheres to game development industry standards. We aim to create a working game that is centered around these core features: networked multiplayer, procedural level generation, and A.I. controlled enemies. The following is a list of the project objectives:

1. **Learn New Technologies**
   This project, being a 15 credit, year long module, affords us a great opportunity to explore areas of software development of personal interest.

2. **Gain Insight Into Game Development**
   We aim to learn tools and technologies that are used in actual industry game development.

3. **Networked Multiplayer Gameplay**
   Many modern games incorporate some form of online multiplayer to bring players together in shared experiences. Incorporating networked multiplayer in our game is an objective of ours.

4. **Procedural Generation**
   Procedurally generated content is a fascinating area. It involves programmers building systems that dynamically create unique game worlds, without the tedium of level design by hand. We plan on developing a system capable of procedurally generating gameplay levels that are different each time the game is run.

5. **A.I.**
   Simulating intelligent behaviour for enemy characters is ubiquitous in the game development industry. There are a myriad of techniques used to accomplish this. We intend to research and implement behaviour-driven enemy characters.

6. **Reproduce Favoured Gameplay Systems**
   As we share a common interest in games, we would like to emulate fun and interesting gameplay systems from games we have played.

## 1.3 Outline

This project paper is broken down into different sections. The outline of each section is explained briefly below. All sections contain their own subsections and will be explained in more detail throughout this paper.

### 1.3.1 Methodology

The Methodology section discusses the Agile framework and development approach taken throughout the project. It contains the Ad-Hoc method of validation and testing for game testing. It outlines all the collaborative tools used for communication, version control, and development.

### 1.3.2 Technology Review

The Technology Review section reviews the various technologies used in the project. These technologies include different engine choices, networking standards, procedural generation algorithms, and A.I. methods. Also explored in this section is the use of C++ programming language and Unreal Engine 4's visual scripting blueprints.

### 1.3.3 System Design

The System Design section details the overall system architecture and design. It explains why specific technologies were chosen and how they were implemented. It details the game design and structure. It also discusses the core features upon which the game was built: Networking, Procedural Generation, and A.I.

### 1.3.4 System Evaluation

The System Evaluation section highlights the performance metrics of the project. It analyses the limitations of the project's core features and the opportunities for expansion.

### 1.3.5 Conclusion

The Conclusion section summarises the objectives achieved and lists the project outcomes. It also discusses the possible improvements and expansions that could be made to the project. Finally, it considers what was learned throughout the project and how these learning outcomes impacted the project.

# 2 Methodology

The methodology section will discuss the direction the project took. This is split into four subsection each detailing a major approach to the project. These are, in order of appearance, Development Approach, Validation and Testing, Collaborative Tools, and Research Method. Game development is an iterative process and the methodology was conducted in this manner.

## 2.1 Development Approach

Game development is a software engineering discipline in which requirements are constantly in flux. The end goal of game development is to create a fun and engaging player experience. The degree to which a particular game mechanic or system lives up to these metrics is hard to quantify without hands on testing. As such, game development requires a constant, iterative development approach, where requirements can be implemented, evaluated, and modified. In developing the "Leviathan Axe", a weapon and core gameplay mechanic in the critically acclaimed 2018 God of War (developed by Sony Santa Monica), lead combat designer Vince Napoli described the process as taking "several years of tweaking"[1] to fully develop and finish that gameplay mechanic alone. The Agile software development framework is a perfect development methodology to meet these needs.

## 2.2 Validation and Testing

Due to the nature of game development, outlining hard metrics for validation and testing can be challenging. Therefore, it was decided that an Ad-hoc testing approach would be taken. Ad-hoc testing is a "less structured way of testing" [1], where the main aim is to verify a feature's implementation. Buddy testing, a subtype of Ad-hoc, was also used extensively to test and validate the application. This involves two team members, "one from development team and one from test team mutually work on identifying defects" [2]. This testing will be done after any moderate to large scale feature has been implemented. Given that this project will be undertaken by a small team of two, this approach to testing may be more appropriate when compared with a larger testing framework.

## 2.3 Collaborative Tools

Collaboration is at the core of Agile development. The importance of easy-to-use collaborative tools that enable friction-less workflow between multiple team members cannot be overstated.

The process of game development requires a consistent and iterative approach. As stated above, the Agile framework is perfectly suited to this. However, it was very important that the project development process was done collaboratively and efficiently. This was ensured by the use of collaborative tools which assigned development tasks, managed milestones, deadlines, and source control which ensured no major conflicting work was made permanent. The tools used to achieve efficient collaborative work were Git, Github, Git-LFS, Trello, Microsoft Teams, and Discord.

Git was used to manage and keep track of the project's source code history, known as version control. Version control tracks and logs the changes made to the project files. This allows project file reviews, in the event of a disaster, to restore an earlier version. This is achieved through the use of snapshots, which can be accessed at anytime throughout the development process to compare and restore as needed. All project files were managed and maintained through the use of Git's version control.

Github is an online cloud-based database that allows its users to manage and share their Git version control projects outside of their local computer. The project files were uploaded to a shared git repository, allowing for the use of collaborative project work while maintaining version control. As this was a small development team, it was deemed appropriate to adhere to a simple branching strategy consisting of a single master branch. Github was also used as an initial project management tool. However, the project management process was ultimately transitioned to the more appropriate software tool Trello.

Git-LFS stands for Git Large File Storage which replaces large files with text pointers inside of Git. Since the project is a game, large files such as audio and video samples, graphics, models and large data-sets were common place. The use of Git-LFS allowed for the uploading of these files, which would have otherwise been impossible as Git has an upload limit size.

Trello is a browser-based list-making application, that enables teams to split a project into granular tasks. These can then be assigned to different team members, and progress can be monitored in an open and transparent way. A prioritisation technique called MoSCoW was employed for managing

the requirements. The acronym MoSCoW stands for four different categories: must-haves, should-haves, could-haves, and won't-haves. This technique was employed onto the Trello board along with two extra headings: bugs and done. A weekly meetup on Thursdays took place to review and update the Trello board. Tasks were added to each list upon agreement of both project members and assigned a title, description, member (to complete task), and optional screenshot, used mostly in the case of application bugs.

Microsoft Teams is a collaboration platform used to help teams stay organised and have communication all in one place. Teams was used to manage weekly meetings, occurring on Thursdays at 9:20am with the project's supervisor. Video calls occurred to discuss project updates, milestones and deadlines, as well as questions team members had about the project brief. It should also be noted that emails were regularly exchanged between the project members and the supervisor, between weekly meetings. Online Google Docs shared between project members and the supervisor were recorded during meetings as meeting minutes and emailed to the supervisor after the video call meeting ended.

Discord is an instant messaging and digital distribution platform designed for creating communities. Users can communicate using voice and/or video calls, text messaging, and sharing of media and files. These all occurring in private or larger communicates called servers. An official server was created and titled the 'Final Year Project'. This server was used as the primary source of communication between the two project members. Daily updates and check-ins occurred and major decisions were discussed in this server.

# 3 Technology Review

## 3.1 Game Engine

The biggest decision to be made when developing a game is whether or not to use a pre-existing engine or to create one. All subsequent work will be done in the perspective of the chosen game engine. In the early stages of game design the idea was to create a multiplayer Rougelike game. Research was narrowed down to three distinct choices: OpenGL C++, Unity Engine and Unreal Engine 4 4.

### 3.1.1 OpenGL C++

OpenGL was considered for the project as the first prototype game engine to be used. Preliminary drafts of the project included the idea of developing a game engine from scratch using OpenGL with C++ as the building blocks. OpenGL is a low-level graphics modelling and rendering software package. Low-level rendering routines give a huge advantage to the programmer when it comes to control and flexibility over the program [6]. Unlike conventional game engines, the OpenGL package enables more granular control over the direction of the program. One of the most famous and earlier made 3D games, Quake 3, was developed using OpenGL.

However, it was considered that the overhead in learning the OpenGL was complex. Given the limited time and scope of the project, the game's end result would most likely be considered too small and too simple. Furthermore, OpenGL does not include any support functionality for other major computer game features, such as sound, input, or networking. This further limited the time allocation for learning OpenGL, as the other aspects of the game development process would need be allocated time.

When compared to modern and conventional game engines, OpenGL lacked a complete package. Game engines like Unity Engine and Unreal Engine 4 4 offered complete modelling and rendering as well as a vast array of other game development tools. These included management systems for sound, input, networking, level design, A.I., and much more. The research into OpenGL revealed the game development process is far more than just the rendering of graphics. It was decided in order to have a more complete and appropriate project scope and result, OpenGL was not an applicable solution. A more complete game engine needed to be considered.

### 3.1.2 Unity Engine

The Unity Engine is a cross-platform game engine used to creat 2D and 3D games, interactive simulators, architecture designs, and even films. Unity Engine is widely popular and considered a game development industry standard [20]. Unlike the previously mentioned OpenGL, Unity Engine consists of far more systems dedicated to constructing a complete video game. Unity Engine is primarily written in the C-Sharp and JavaScript languages. It supports many different types of 2D sprite and images, 3D modelling types, and a range of sound and music types. Unity Engine is thoroughly documented and possesses an active community to help solve problems and issues.

Both project team members have worked with Unity Engine in past modules and projects. The level of experience was a huge advantage when considering Unity Engine as the game engine choice. The engine was thoroughly understood by both project members. However, when considering the scope of the project and industries constantly changing atmosphere, it was decided that Unity Engine would not provide the learning outcomes desired. Considering the comfort level and understanding possessed by project members with the Unity Engine and C-Sharp language, it was ultimately decided a new engine would produce a more appropriately scoped project and learning outcome.

### 3.1.3 Unreal Engine 4

The Unreal Engine 4 is comprised of a unique set of development tools for working with real time technology. Like Unity Engine, the Unreal Engine 4 has evolved over time to reach a wider set of developers across different industries like films, games, and even transportation. Unreal Engine 4 is an industry standard and leader in advanced real time 3D creations. Unreal Engine 4 is written in the C++ language and a visual scripting language called blueprints, unique to the Unreal Engine 4. The documentation on the Unreal Engine 4 is vast for blueprinting and video tutorials. However, it lacks somewhat in the C++ department. This is remedied by the large and active community online in support of the Unreal Engine 4.

The Unreal Engine 4 possesses development tools such as the replication system for creating fully networked games and behaviour trees for use in A.I. These were considered highly advantageous for the development process. When considering the scope of the project, the learning curve required for

C++ and visual scripting blueprints was agreed appropriate.

## 3.2 Programming and Visual Scripting

The Unreal Engine 4 consists of two primary forms of development: C++ and visual scripting blueprints. The project was developed using a hybrid of the two. This allowed for flexibility during the development. The procedural generation was programmed entirely using C++ and produced highly effective results. Meanwhile, A.I. was development through the use of blueprints.

### 3.2.1 Blueprints

The visual scripting blueprint system uses a node-based interface to assist in creating gameplay elements from within the Unreal Engine 4 editor. Blueprints use an object-oriented approach to classes and objects defined in the engine. While not strictly a programming language, the use of blueprints requires a fundamental understanding of programming languages. Conditional statements like if-statements and boolean branching, looping structures like for-loops and while-loops, all exist within blueprints and a intuitive understand is needed to extract their full potential.

In designing the player movement, combat, animations and inventory systems, blueprints were incorporated. This was extremely advantageous in many ways. It allowed for a quick and easy setup of features for testing. Customised and loosely coupled blueprints could be attached to multiple objects in the game world. Changes could easily be made in the parent blueprint and would be updated in all child objects containing the blueprint. Interfaces and functions were encapsulated within their own nodes, which could then be dragged and dropped into any inheriting blueprint or object.

### 3.2.2 C++

Unreal Engine 4 is written in and supports development of features and systems in C++. As part of our second year Procedural Programming and Advanced Procedural Programming modules, we had both been given good exposure to C, and were familiar with concepts like memory management and pointer arithmetic. Using C++ would be a great opportunity to expand

on our previous studies in C and combine that with some object-oriented programming and procedural programming concepts.

## 3.3 Networking

Networked multiplayer gameplay is a core objective of this project. The implications of networking on gameplay and systems must be fully thought through prior to any development [4]. Multiple geographically distributed instances of the application will need to exchange data in real time and with low latency. This means that any network architecture, by necessity, will be tightly coupled with other elements of the codebase. It is unrealistic to think that networking implementations could be swapped and changed throughout the development process. The importance of proper research and understanding of planned gameplay mechanics and systems, in the context of networking, cannot be overstated.

Most modern game engines have libraries for building networked multiplayer, either officially or community supported. Regardless of the game engine or library, there are two primary models for implementing networked gameplay: peer-to-peer, and client-server. Both topologies have their advantages and disadvantages and needed to be extensively researched to find the best fit for our game's needs.

### 3.3.1 Peer-To-Peer

In general computing terms, a peer-to-peer network is a form of decentralised, distributed computing whereby multiple computers running an instance of an application that share resources and data to achieve some goal [16]. In the context of game development [3], a peer-to-peer network architecture may slightly deviate from this definition. Where in a conventional peer-to-peer topology, decision making authority is generally shared between nodes in the network, a game implementation would likely have one instance act as both host and client, and all other players on the network as clients, connecting to that host. This has important gameplay implications, as one player is running both client and host code, they will have a distinct latency advantage over the other connected players. In a competitive game, where milliseconds of latency can make the difference, a peer-to-peer networking architecture may not be suitable if well balanced gameplay is a priority.

A distinct advantage of peer-to-peer would be its cost effectiveness and

scalability. Having all gameplay data be shared directly between players in a given session negates the need for a centralised server to be hosted and maintained by the developers. For our game, we expect each session to be comprised of a handful of players, no more than 5. If our game were to be downloaded by millions of players, the peer-to-peer architecture means that each session group of players would connect to each other and thus, cover the networking costs of running the game themselves. The only consideration we need to make is facilitating the initial setup of those connections. The 2016 space-faring strategy game "Stellaris" handles this by hosting a server browser, this provides the means for players to offer to host games as well as find games to join. Once a game has been setup, all networked gameplay data is handle by that individual cluster of players.

### 3.3.2 Client/Server

In a client/server model "clients typically host a small subset of the data in the application process space and delegate to the server system for the rest" [8]. This definition is suited to general computing but in terms of game networking, a more commonly found client/server implementation involves some sort of replication system, in such a system, "specific objects and data members are transmitted on creation or changes" [3]. The server has authority and is responsible for propagating any changes in the game world to each client. When a client takes an action that would change the data of a game object, they must tell the server first, if the server authorizes this action, it communicates the updated game object to the other clients. For example, the player presses the space bar to jump, this action would change the players position, so before this change is approved and propagated, the server might have to check if the player is allowed to jump.

The client/server model offers the best decoupling of game code from networking implementation, if game objects can be tagged as replicated i.e. the server needs to tell the clients if it's state has changed, any scripts that act on those game objects don't need to be aware of this replication. This decoupling is a huge advantage of the client/server model. [3]

### 3.3.3 Local Multiplayer

Local multiplayer is by far the most primitive implementation of multiplayer gameplay. Sometimes referred to as "Couch Co-Op", it involves all players

using different input controllers to the same device and application. This form of multiplayer doesn't require any network technology to work and has existed since the early 60s in games like "Spacewar!", the 1962 shooter [9]. As networked multiplayer adds an extra dimension and domain of study to our project, forgoing it, in favour of local multiplayer would really make a dent in the scope of our overall achievement. Local multiplayer might best serve as a last resort alternative, should networked multiplayer prove to be too challenging.

### 3.3.4 Steam Integration

For a professionally developed game, end users should not be expected to be experts in networking technology. All the complexity involved in establishing UDP/TCP connections, handshaking, packet switching and more should be totally abstracted out of sight. A user should expect to only need an internet connection, and to press a button, to be launched into a networked multiplayer session. In addition to this, an important consideration to be made in game development is, where our finished product could actually be sold. Our target platform is the PC, but even within the PC game market, there are several sales platforms that offer libraries and SDKs for integrating your game with their platform. Steam is one of the largest PC game distribution platforms, in 2019, the platform reported over "121 million visits to its downloadable content pages" [21]. Steam also support a networking SDK for the Unreal Engine 4 which provides the means to connect steam users to each other's gameplay sessions. This solves both the problem of abstracting networking complexity away from the end user, as well as a providing a platform to potentially publish our game on. An end user need only download the game from steam and can then easily join multiplayer games with other steam users.

## 3.4 Procedural Generation

In light of the fact that this is a software development project and not a game design project, early on it was decided that a core objective would be to develop a system capable of building a gameplay level at runtime, instead of hand placing game objects in the world i.e., traditionally more of a game designer's role. Procedural generation is used to solve a wide array of problems in game development, from world building and level design to

procedural textures and animations. Our focus will be on procedurally generating levels, games like the 2008 2D platformer "Spelunky" or the 2011 action-adventure sandbox game "Terraria" both make heavy use of procedurally generated levels [13]. This approach means that the overall technical challenge of the project will be greater than if it was decided to hand build levels.

It was decided that our game levels would be built from square tiles in a 2D grid-based system, these tiles would be placed randomly until a dungeon of the required size was constructed. There is a plethora of methods for procedural generation, for the purpose of this project, two approaches were primarily considered, random walk tile placement and a room/hallway graph approach.

### 3.4.1 Random Walk

Random walks are probably the simplest approach to procedural level generation. In a mathematical sense, a random walk is defined as a "stochastic or random process, that describes a path that consists of a succession of random steps on some mathematical space" [11]. To apply this concept to procedural level generation, consider a 2D grid-based coordinate system with a random "walker" at the origin. The algorithm works by picking an axis, and direction along that axis to move, this can be done in code [12] with random number generation to simulate a 50/50 chance i.e. a coin toss. So, at the origin, flip a coin to determine whether the walker will move on the x or y axis, and then again to decide if they will move in the positive or negative direction along that axis (+1 or -1). Once they move, place a tile on that square of the grid, then repeat the process until enough tiles have been placed.

The biggest advantage of this method is its simplicity to both understand and implement; however random walks have several shortcomings with important implications. Implemented as described, the random walker has no way of knowing what co-ordinates they have already been to, this will likely result in some number of duplicate tiles being placed on top of each other. This doesn't mean that a random walk is unsuited to procedural level generation, it is however, an important characteristic to keep in mind, especially when considering the scalability of the system. A few hundred extra tiles being placed in the game world is not likely to be a substantial strain on computational resources, but if the map size was to be in the order of thousands or billions of tiles, rendering the textures of all these duplicated tiles

may cause a serious reduction in frames per second output.

### 3.4.2 Prims Algorithm

Employing some graph theory may work as a better solution to procedurally generating a game level. Unlike in random walking, where the level was one sprawling area of connected tiles, consider a level to be a collection of rooms, and hallways connecting those rooms. Taking these rooms and hallways to be nodes and edges in a graph, it would be feasible to build complicated, intricate, and most importantly, fun to play levels. A set of rooms could be generated using random functions to determine their dimensions and location on a 2D grid. It is important that each node in the graph should have a pathway to each other node, so that the player is never placed into a level that cannot be traversed. These nodes should also be connected by an edge to their nearest neighbour, to stop edges intercepting multiple nodes.

Prim's algorithm provides a near perfect solution to these problems. A greedy algorithm that "finds a minimum spanning tree for a weighted undirected graph" [10]. In this case the weight of an edge will be the calculated distance between those two hallways. The algorithm works by first deciding on a node to start with, this node is added to the tree. The next step is repeated until all nodes have been connected: for each node in the tree, construct an edge with the lowest weight that would connect the next node to the tree.

## 3.5 A.I.

The use of A.I. (A.I.) in gaming has become an industry standard. When considering the approach and implementation of A.I. 3 types were researched. Goal-Oriented Action Planning (GOAP), Finite State Machines (FSM), and Behaviour Trees (BT) were all researched. The project needed responsive and efficient A.I., simple design with complex decision making ability, and the ability to easily build upon what has already been developed. Finally, the chosen engine, Unreal Engine 4, and the ease of which the different A.I. types could be implemented was considered.

### 3.5.1 Finite State Machines

Finite-State Machines (FSM) are a mathematical model of computation with a limited number of conditional states. FSM are commonly found in behaviour driven development and highly popular in the earlier years of game development. The simple controller button layout found on many different gaming consoles are a real-world and practical example of a FSM. FSM wait on key inputs to change states which then result in a desired behaviour or outcome. They are simple in design often dividing different desired behaviours into encapsulated logical states.

A review on *"Current AI in Games"* [19] found FSM are used commonly across the gaming industry and possess the benefit of being simple in design and required little tuning, freeing up developers time and resources, compared to alternatives like Goal-Oriented Action Planning (GOAP). However, the review went onto conclude the growing weariness and predictability gamers feel when FSM are used heavy-handedly in A.I. driven characters.

The use of FSMs in the project would fit suitability due to the common use in the industry and simple development process. However, the simplicity of FSMs may lead to the A.I. driven non-playable characters to responding poorly to changes real-time changes and appear to be dull and unappealing. A study [15] found that FSM were slower in response time when compared to alternative A.I. driven techniques like GOAP.

### 3.5.2 Goal-Oriented Action Planning

Goal-Oriented Action Planning (GOAP) is an A.I. design pattern that enables the planning of a series of actions to achieve a desired goal. The sequence in which these actions unfold is entirely dependent on two major factors. The first is the current goal the A.I. is trying to achieve and the second is the current state of the application and A.I. current state.

A study [15] comparing GOAP and FSM found that GOAP was superior in its decision making ability when sharing a common goal. Furthermore, it found FSM under-performed because of its reactive nature, always waiting on an external input. GOAP instead *"weighs up possible options by evaluating the relevancy of all its goals"* and and comes to a decision based on the applications, and its own, current state. The declarative design of GOAP allows for more complex and intuitive A.I. behaviours.

Another advantage in using GOAP, is the development and coding pro-

cess is more structured, re-usable, and maintainable then FSM. One study [18] found A.I. driven entities possessed *"more varied, complex, and interesting behaviours using GOAP"*. These A.I. entities could determine their behaviour at runtime and could dynamically solve goals using its current surroundings. Moreover, the same study found that as A.I. driven behaviours advanced to become more complex, the need for more structured solutions in creating A.I. grow. GOAP is a highly useful design strategy for A.I. driven entities.

### 3.5.3 Behaviour Trees

Behaviour Trees (BT) are a form of A.I. which has a mathematical model possessing a plan execution. The plan execution consists of all the possible states the A.I. behaviour can switch between. Each state can consist of many simple tasks which can create entirely new and more complex tasks. BTs are visually easy to understand and easy to debug as each task is separate form the next. BTs have become increasingly popular in the games industry and have dedicated implementations in game engines like Unity Engine and Unreal Engine 4.

A study [14] comparing BT with hand-coded FSM machines found BT driven bots were *"capable of beating the original, hand-coded DEFCON AI-bot more than 50% of the time"*. The research shows BT have the ability to outperform their predecessors FSM. Proving this point further another study [17] concluded the BT driven A.I. *"achieved a success rate superior to 50%"* when pitted against the standard A.I. controller. BT possess a more inherently understood data structure, known as a tree, which comfortable software developers would have a greater understanding of. Couple this with the ability to build simple tasks, group them together and build larger more complex tasks, BTs are more desirable when developing advanced and unpredictable A.I for gamers.

BTs are extensively documented in the Unreal Engine 4 docs. Unreal Engine 4 has a dedicated tool for creating and maintaining BT driven entities within the engines application. This is a huge advantage over GOAP and FSM becasue Unreal Engine 4 doesn't possess dedicated tools or documentation to either of the alternatives.

# 4 System Design

## 4.1 Unreal Engine 4

It was decided that the Unreal Engine 4 would be used in the development of this project. To properly meet the scope requirements of this project, it was advised that we make use of technologies or frameworks not previously studied. This was the primary reason for choosing Unreal Engine 4 and C++/Blueprints. Each of these technologies are completely new to us and required independent learning.

## 4.2 Network

The networked multiplayer gameplay was implemented using a combination of in-built Unreal Engine 4 technologies [4] like Replication and the Listen-Server architecture, as well as the Online Subsystem for Steam API [5].

### 4.2.1 Listen-Server

This project makes use of the "Listen-Server" feature of Unreal Engine 4 to build networked multiplayer. This follows a straightforward enough Client-Server model with one notable difference. As in any Client-Server architecture, there is a single instance of a server which holds decision making authority over all connected clients. This means the server is in essence running the "real" instance of the game state and propagates changes out to any connected clients. Normally the server would have to run on a dedicated machine in a remote location, and all the clients would have to connect to this server. The Listen-Server system provided by Unreal Engine 4 enables one player to "host" the server, while also playing as a client connected to that server. This architecture is perfectly suited to co-operative games, where latency doesn't have to be unbiased between all clients connected to the server.

### 4.2.2 Online Subsystem for Steam

The Online Subsystem Steam API was used for its matchmaking features. It streamlines the development of multiplayer games by offering functions that manage everything to do with establishing connections between steam users.

Having the steam SDK integrated means that in the future, the game could be published on the steam marketplace for sale.

### 4.2.3 The Matchmaking Process

If a player wants to host a multiplayer game, they must first have steam running in addition to this application. Once they navigate to the multiplayer menu and opt to host a game, they will be prompted to specify a number of players allowed in the session. Upon launching the server, the player will be moved to a lobby map where they can control their custom character and can see the lobby menu, which shows the number of players currently connected as well as a button to launch the game.

There are two core blueprints that handle the communication flow between server and client. Lobby GameMode and Lobby PlayerController. Each instance of the application has both blueprints, but only the server will execute code in the Lobby GameMode. This blueprint has an entry point event "OnPostLogin" which fires when a client successfully connects to the server (including the host client). Before any other code is executed following this event, a check is done to see if the instance of the application running the code has authority i.e., they are the host. Simply put, every player connected to the server will receive an "OnPostLogin" event, but only the host will execute code to handle that event.

Once a new player is connected, the server will do several things to set them up in game. First the server will ask for the players character data, things like avatar and name, by calling InitPlayerData on the newly connected players PlayerController. InitPlayerData checks the players local storage for a save game file and provides that data to the server and requests that the player be spawned in by calling "CallUpdate". The CallUpdate function is run on the PlayerController, it first requests that the server spawn the player in, and then requests that the server propagate this new player data to all other connected clients. At this point in execution, a player has successfully connected to and been spawned into the game and should be able to control their character as well as see other players in game.

Once the players have been connected, the next step is to transport them all to the game map at the discretion of the host. The host can press the start game button which will call the LaunchGame function on the Lobby GameMode, this function transports all connected players into the gameplay map.

## 4.3 Procedural Generation

The procedural level generation uses an implementation of Prim's algorithm to build a minimum spanning tree of nodes and edges (rooms and hallways). All the code involved in procedural generation can be found in DungeonGenerator.cpp and its associated header file. The BeginPlay() function of this file is Unreal Engine 4's Actor class, and is run at the start of the game. This function contains all other function calls involving the procedural level generation.

### 4.3.1 Generate Rooms

GenerateRooms() is called first. This function works by iterating over the number of rooms and placement attempts, both specified in the editor by a designer. On each iteration a room with a random position and dimensions is generated, Overlap() is then called to check if this room would clash with any rooms already generated. This was done to ensure all the rooms would be separate from each other in the game world. If the room would not overlap with anything, it is added to the collections of rooms.

### 4.3.2 Overlap

Overlap() is a simple enough function that takes a potential room, and checks if that room would overlap with any rooms that were already generated. Calculating the overlap of two rectangles was done by comparing the coordinates of their vertices. An interactive and visual explanation, and the primary resource used to write this function can be found here https://silentmatt.com/rectangle-intersection/. Note, in our implementation, rooms are stored as a struct of three vectors, position (of the bottom left corner), size and centre. The coordinates of the bottom right corner, for example, are calculated by taking the Position.X and adding Size.X, then the Y coordinate would just be Position.Y.

### 4.3.3 Calculate Centers

CalculateCenters() is called once all of the rooms have been generated. The function iterates over each room and calculates the position of its centre by dividing its width and height by 2 and adding this to the x and y value of the position of its bottom left corner. For the sake of simplicity and

decoupling, all co-ordinates and dimensions are dealt with as integers, a logical representation of the game world. It isn't until the level has been entirely generated that the actual position in the game world of these rooms is calculated by multiplying each integer by an offset.

### 4.3.4 Construct Graph

ConstructGraph() is the implementation of Prim's Algorithm [10] and is called once all the rooms have been generate and their centres are known. There are two collections of rooms, the first, "Rooms" is all the rooms that have yet to be added to the graph i.e. unvisited, the second is collection "Nodes" these are rooms that have been visited and therefor are connected to the graph by at least one edge. The first step of this function is to find the room with index 0 and add this to the graph. This is only done to initialize the graph. The next step is repeated until all nodes have been connected to the graph by an edge. Iterate over each node already in the graph and calculate the distance between that node and each node not already in the graph. When this is done, the visited node, with the shortest distance to an unvisited node will have an edge joining them, thus adding another node to the graph.

### 4.3.5 Calculate Shortest Edge Distance

CalculateShortestEdgeDistance() iterates over two collections of rooms and returns an edge with the lowest weight that would link two rooms.

### 4.3.6 Generate Halls

GenerateHalls() is called next, at this point a logical representation of the level exists in the form of a minimal spanning tree of nodes and edges. This function uses those logical edges to construct the in-world hallways that would connect two rooms. It works by iterating over each edge and checking how the two rooms line up next to each other. For two rectangles next to each other on a 2D plane, there are 3 possible ways they can line up. 1) Horizontal i.e. room A is to the left or right of room B. 2) Vertical i.e. room A is directly above or below room B. 3) Corner i.e. room A is up and to the left of room B. Once the relative orientation of two rooms is known, a hallway can be constructed to connect those rooms.

### 4.3.7 Build Dungeon

BuildDungeon() is the final stage of procedural level generation. At this point, there is a collection of rooms, and hallways that connect those rooms, all represented in simple integer coordinates e.g. (2,4), (1,3). These integer coordinates were used to logically decouple the graph representation of the level from its in-game representation. This function iterates over all the logical rooms and hallways and translates them into in game world coordinates. It also, based on a chance configurable in the Unreal Engine 4 editor, spawns in enemies and potions.

### 4.3.8 Relative Position of Two Rooms

CheckHorizontalOverlap(), CheckVerticalOverlap() and CornerHallway() all take two Rooms a and b, and construct a hallway that will connect those rooms. These functions mostly work in the same way by checking how two rooms line up next to each other and then picking random coordinates within an overlap range such that a straight-line hallway can be drawn to connect the two rooms. The corner hallway has a few more conditions to be checked, if A is to the bottom left of B, a hallway should move right from A and then up to B.

## 4.4 A.I.

The A.I. in this project was developed using the Behaviour Tree (BT) Artificial Intelligence design method. The A.I. was used to create humanoid enemy characters within the game world that would function independent of the player's input. There are three major components which make up the A.I. controlled enemy characters in the Unreal Engine 4. These are the **Behaviour Tree, Blackboard, and AI Controller**. The BT executes on event-driven logic. These events act as passive triggers, sending feedback to the BT. The Blackboard is used to store all information that may be associated with the AI controlled enemy. The stored information is referred to as **Blackboard Keys** (see Fig. 2) and can be accessed to inform the BT in decision making. Lastly, the AI Controller is used to take control of and direct a Pawn (the entities "body" or object) to perform actions and run the associated BTs. The AI Controller contains the **AIPerception** component with "AISense_Sight" which enables the pawn to receive data from the envi-

ronment.

The BT implements a tree structure (see Fig. 1) which has branching nodes called **Composite** nodes. Composite nodes (Grey in Fig. 1) are the entryway to a branch. They define the root and base rules for how a branch is to be executed. The first node in the BT is the "root" node. This is the Composite node where all subsequent Composite and Task nodes branch from. The types of Composite nodes used in the BT design are the **Selector** and **Sequence** nodes. The Selector Composite nodes execute their child nodes from left to right and stop execution when one of their child nodes succeed. Sequence Composite nodes also execute from left to right and stop execution when one of their child nodes fails. Each Composite node can have attached **Decorators** and **Services** (Blue and Green, respectively in Fig. 1). Decorators are conditionals which define whether or not a branch in the tree can be executed. Services execute as long as their parent branch is executing. Finally, Composite nodes can have multiple **Tasks** (Purple in Fig. 1) attached. These contain different behavioural logic, which is desired to achieve different objectives or **Tasks**.

The BT was designed with four major Composite nodes for implementing the A.I. actions: PursuePlayer, CirclePlayer, AttackingPlayer, and Patrol. All Composite nodes contain a number of Decorators and additionally contain one Service, excluding the Patrol. Furthermore, each Composite node also contains a number of different custom developed Tasks. Each of the major Composite nodes are discussed below in more detail.

### 4.4.1 PursuePlayer

The PursuePlayer Composite node (see Fig. 3) contains the logic used by the enemy to find and move to a player character. It is the left most Composite node in the *root* and is the first branch to execute. The branch is a Sequence node comprised of two Decorators, one Service, and four Tasks. Upon completing its execution, it returns to the 'AI Decision Root' Composite node. The newly updated information is stored in the Blackboard.

The branch executes when the two Decorator conditions have been met. The first Decorator is the 'HasTarget?'. This checks whether the enemy character has a reference to a player character. This is an object reference variable set in the Blackboard and updated by the AIPerception component. When a player character comes into the field of view of the AIPerception, it will return the player character's object. This object reference contains the

player location and is stored in the Blackboard Key TargetActor. The second Decorator is called 'InRange?', which checks to see if a player character is not in range to the enemy character. This is a boolean variable set in the Blackboard and updated by the attached Service Composite node. When the player is not in range it will update the boolean variable stored in the Blackboard Key InTargetRange?. Last in the PursuePlayer Composite node is the Service 'InRange_BTService'. This Service sets a range value to 300 and attaches it to the InTargetRange? Blackboard Key. When a player enters into the range set in the Service, it updates and informs the InTargetRange? Blackboard Key.

After both Decorators and the Service have been executed, the branch Tasks can now be executed and perform the logical behaviour of the enemy character. Inside the PursuePlayer Composite node are four Tasks each performing a different set of logic. The order of execute is dictated by the Sequence Composite node which is from left to right. The four Tasks are, in order of execution, **SetMovementSpeed_BBTASK, DrawWeapon_BBTASK, Move To,** and **RotateNPC_BBTASK**. The Tasks are explained below.

**(1)** SetMovementSpeed_BBTASK gets a reference to the Base NPC Blueprint and updates the enemy characters Walk Speed. The Walk Speed is a floating point variable set in the Task at 550 (see Fig. 4).

**(2)** DrawWeapon_BBTASK gets a reference to the Base NPC Blueprint checks a boolean 'WeaponDrawn?'. If it is true it finishes execution. If it is false it will call on the function 'Draw Weapon' in the Base NPC Blueprint. This will perform all the logic needed to have the enemy character draw its weapon (see Fig. 5).

**(3)** Move To is a default Task available in the BT system. It takes the Blackboard Key TargetActor, which is an object reference to the player character's pawn. It then moves the enemy character to the TargetActor location within a set range value of 25.0.

**(4)** RotateNPC_BBTASK passes an object reference of the player character into a function called 'SetFocus'. The function sets the enemy character to face and remain focused on the passed player character (see Fig. 6).

### 4.4.2 Patrol

The Patrol Composite node (see Fig. 7) is the logic that enables the enemy character to patrol the map and wander around the level area. It is the last Composite node branching off of the *root* and is executed last. The Patrol was designed to execute last in the event that none of the alternative branches executed. The desired default behaviour was to have the enemy character patrolling. The branch is a Sequence Composite node comprised of two Decorators and six Tasks. As with the PursuePlayer Composite node, all child Task nodes execute from left to right and return to the "AI Decision Root" Composite node upon completion.

The branch executes when the two Decorator conditions have been met. The first Decorator is the 'Cooldown'. This is one of the many default Decorators Unreal Engine 4 BTs provide. It takes in a floating point number which decides the amount of time in seconds to elapse before continuing on in the branch. This is set to 1.5. The last Decorator is the HasTarget? mentioned previously in the PursuePlayer section. However, the HasTarget? now requires the Blackboard Key TargetActor to contain a reference to the player character. If it doesn't have the reference it will exit the branch.

After the two Decorators have been passed successfully, the branch will then execute the six Task nodes. The six Tasks are, in order of execution, **UnRotateNPC_BBTASK, RotateStrafeResetNPC_BBTASK, SheatheWeapon_BBTASK, GetRandomLocation_BBTASK, Move To** and **Wait**. The Tasks are explained below.

**(1)** UnRotateNPC_BBTASK retrieves a reference to self (enemy character) and calls the function 'ClearFocus'. This function clears the focus of the enemy character, allowing the enemy character to rotate freely (see Fig. 8).

**(2)** RotateStrafeResetNPC_BBTASK retrieves a reference to self (enemy character) and sets a variable on the pawn 'Rotation Yaw' to false. This enables the enemy character to face the direction it moves in once its focus cleared (see Fig. 9).

**(3)** SheatheWeapon_BBTASK retrieves a reference to the BASE NPC Blueprint and checks the boolean 'WeaponDrawn?'. If the return is false it finishes execution. If the return is true it will call on the function 'Sheathe Weapon' in the Base NPC Blueprint. This performs the logic to sheathe the enemy

character's weapon (see Fig. 10).

**(4)** GetRandomLocation_BBTASK first gets the location of the enemy character. It then calls the function 'GetRandomPointInNavigableRadius' which is passed in a floating point number 1000 into the variable 'Radius'. The resulting vector is set to the Blackboard Key 'PatrolLocation' and finishes execution (see Fig. 11).

**(5)** Move To is similar to the previously stated Task in section 4.4.1 PursuePlayer. However, the Blackboard Key TargetActor sets the range value to 5.0 instead.

**(6)** Wait is another default available Task in the BT system. It contains two variables 'Wait Time' and 'Random Deviation'. Wait Time is the number of seconds the BT will wait when this Task is called. The Random Deviation is the time deviation in seconds from the Wait Time. Wait Time is set to 3 seconds and Random Deviation is set to 1 second. This creates a random number of seconds to wait between 2 and 4 seconds.

The middle two Composite nodes (see Fig. 12) branch out of the Selector Composite 'StrafeOrAttack'. This is the middle Composite node that branches between the PursuePlayer and Patrol Composite nodes. Once the execution reaches the StrafeOrAttack Composite node it will work from the 'CirclePlayer' Composite node to the 'AttackingPlayer' Composite node. The StrafeOrAttack node holds the logical behaviour for the enemy character's combat. The AI will switch between strafing and attacking the player character. This switching behaviour is achieved through the use of 'Observers' attached to the Decorators. The Observers are notified of any change with the Decorator's associated Blackboard Keys. The Observers are set to abort 'Both' which will allow the branch to abort its execution on any Blackboard Key update.

### 4.4.3 CirclePlayer

The CirclePlayer Composite node (see Fig. 13) contains the first of the combat mechanics that allows the enemy character to strafe and encircle a player character. It is the first Composite node in the StrafeOrAttack Composite node. The branch is a Sequence Composite node comprised of

four Decorators, one Service, and five Tasks. All child Task nodes execute from left to right.

The branch will execute when the four Decorator conditions have been met. The first two Decorators HasTarget? and InRange? are the same as previously mentioned in section 4.4.1 PursuePlayer. The HasTarget? Decorator needs a player reference to pass. The InRange? Decorator checks to see if a player character is within the appropriate range. This range value is set in the Service. The third Decorator is the 'AttackReady?' Decorator. This is a boolean variable called 'Attacking?' which is passed in from the Blackboard as a Key. It needs to return false to continue the branch execution. The last Decorator is the Cooldown Decorator with its value set to 1 second. This Decorator is previously discussed in section 4.4.2 Patrol. The 'InRange_BTService' Service is the exact same as previously discussed in section 4.4.1 PursuePlayer. The range value is set to 300.

Once all the Decorators and Services have been successfully executed, the CirclePlayer Composite node will move onto executing the five Task nodes, from left to right. The five Tasks in order of execution are **RotateStrafeNPC_BBTASK, SetMovementSpeed_BBTASK, GetStrafeLocation_BBTASK, SendToLocation_BBTASK** and **AttackPlayerCounter_BBTASK**. The Tasks are explained below.

**(1)** RotateStrafeNPC_BBTASK retrieves a reference to self (enemy character) and sets the variable 'Rotation Yaw' on the pawn to true. This allows the enemy character to move around the player character while always facing the player character. This is followed directly after the PursuePlayer Composite node's last Task is executed and the player character is within range (see Fig. 14).

**(2)** SetMovementSpeed_BBTASK retrieves a reference to the Base NPC Blueprint and updates the enemy character's Walk Speed. The Walk Speed is a variable set in the Task at 90 (see Fig. 4).

**(3)** GetStrafeLocation_BBTASK retrieves a reference to the Base NPC Blueprint and its pawn's world location. It multiplies the pawn's forward vector by 30 and the pawn's right vector by 150. It adds the world location, forward vector, and right vector to calculate a new vector. This vector translates to an angle of roughly 22.5 degrees from the enemy character's location, pointing in the forward-right direction. At the same time the Task is calculating the

same 22.5 degree angle, but with the pawns right vector multiplied by -150. This effectively translates to the forward-left direction. A random integer is generated between 0 and 1 which corresponds to a branch node. 0 is equal to true and 1 equal to false. True takes the calculated 22.5 degree angle to the right and sets its value to the Blackboard Key 'StrafeLocation', a vector variable. False takes the calculated 22.5 degree angle to the left and sets its value to the same Blackboard Key StrafeLocation (see Fig. 15).

**(4)** SendToLocation_BBTASK retrieves a reference to self (enemy character) and calls the function 'Simple Move to Location'. This takes in the Blackboard Key passed StrafeLocation and its value as a vector. This enables the enemy character to move to the calculated 22.5 degree angle, facilitating the strafing movement (see Fig. 16).

**(5)** AttackPlayerCounter_BBTASK generates a random integer between 0 and 2. A branch node returns true if the random integer is equal to 1, otherwise it will return false. If true the Task sets the Blackboard Key 'Attacking?' boolean variable to true. If false the Task sets Attacking? to false. The Attacking? variable will be set true one-third of the time and false two-thirds of the time (see Fig. 17).

### 4.4.4 AttackingPlayer

The AttackingPlayer Composite node (see Fig. 18) contains the remaining combat mechanics that enable the enemy character to attack the player character. It is the second Composite node in the StrafeOrAttack Composite node. The branch is a Sequence Composite node comprised of four Decorators, one Service, and three Tasks. All child Task nodes execute from left to right.

Branch execution will take place once all the Decorator conditions have been met. All Decorators contained in the AttackPlayer branch are identical to the previously mentioned CirclePlayer branch in section 4.4.4. The HasTarget? Decorator needs a player reference to pass and the InRange? Decorator needs the range value, set in the Service, to be met. The AttackReady? Decorator needs the Attacking? Blackboard Key to return true in order to continue execution. The Cooldown Decorator has a value set to 0.5 seconds.

After all the Decorators have been successfully executed the Task nodes will execute, from left to right. The three Tasks in order of execution are **SetMovementSpeed_BBTASK, AttackPlayer_BBTASK** and **AttackPlayerCounter_BBTASK** . The Tasks are explained below.

**(1)** SetMovementSpeed_BBTASK retrieves a reference to the Base NPC Blueprint and updates the enemy character's Walk Speed. The Walk Speed is a variable in the Task set at 0 (see Fig. 4).

**(2)** AttackPlayer_BBTASK retrieves a reference to Base NPC Blueprint and calls the 'Attack Player' function. This enables the enemy character to perform the attack and combat logic (see Fig. 19).
**(3)** AttackPlayerCounter_BBTASK is explained in section 4.4.4 (see Fig. 17).

## 4.5 Gameplay Systems

The 3 Pillars of the project were explained above in detail. This section will explain the remaining features of the project that are note worthy. It will discuss: Combat, Animations, Inventory, Level Design, Character Entities.

### 4.5.1 Combat

The combat was designed to be simple. It was not intended to be a core feature of the project but instead used to facilitate a way to interact with the A.I. driven enemies. The combat in this project was developed using freely acquired assets from the Unreal Engine 4 marketplace. The assets used for combat animations is an asset pack called 'Paragon: Greystone'. It contains all asset components from the Greystone character in the popular game Paragon made in the Unreal Engine 4. The animations were imported and retargeted to 3rd person skeleton mesh. The combat animations included three attack animations and one hit reaction animation. These were used to create three subsequent attack animation montages and one hit reaction animation montage. The animation montage enables the animation effects to be exposed within the Blueprint visual scripting. This allowed for the animations to be used after a desired logic has been performed. Two more animations

were custom made. These are the DrawWeapon and SheatheWeapon animations and were used to created two animation montages of the same name. Additionally, the player character can approach a weapon object in the game and press the 'E' key to equip it. If the player already possess a weapon with will switch the weapons.

The combat logic possesses the 'WeaponDrawn?' boolean variable. This is used to check if the player weapon is drawn and is set to false by default. When the player click the left mouse button it will turn true. This will then play the DrawWeapon animation montage. The player can then attack or sheathe their weapon. To sheathe their weapon the player clicks the right mouse button which sets WeaponDrawn? to false, and calls the SheatheWeapon animation montage. When WeaponDrawn? is true the play can now use the left mouse button to attack.

The attack function use three different attack animation montages. The player character possesses a integer variable 'AttackCounter'. This is used to count the number of times the player click the left mouse button when attacking. The AttackCounter increments with every left mouse click. Three different attack animation montages are mapped to the AttackCounters from 0 - 2. As the player continues to click the left mouse button the animation montages will play out the Attack: A, B, and C. When the player clicks the left mouse button for the 4th time in a row the AttackCounter is set to 0 and the animation montage will reset to the first attack A. If the player pauses between attacks for longer then 1 second the attack counter is reset to 0.

### 4.5.2 Inventory

The inventory system was designed to be simple and easy to use. It was not a core feature of the project design but was added in regards t0 a suggestion proposed by the project supervisor. The inventory system is made up of a structure (similar to the c structs) containing a string 'Name', Texture2D 'Icon', boolean 'Stackanle?', and Object 'ItemClass'. Next and interface was created called 'Item_Interaction' which contains to functions 'Interact' and 'UseItem'.

Three blueprint classes were created to make three consumable items. These were the 'HealthPotion', MaxHealthPotion', 'StrengthPotion'. Each of the consumable items implemented the interface Item_Interaction. When the player overlapped with the object and pressed the 'E' key they would called the interface function Interact. This would add the consumable item

to their inventory. The player could open their inventory by pressing the 'I' key and see the added consumable item. If the player picked up more then one of the same consumable type, i.e. 3 HealthPotions, they would be stacked in the same inventory slot with the number of stacks 3 showing. The player can left click on an item stack and a popup window will appear. It will present two options: Use or Drop. The Drop option will drop 1 from the item stack and spawn that consumable item in the world at the players feet. The Use option will call the interface function UseItem. This function is implement in each consumable item differently and explained below:

- *HealthPotion :* Heals the player character for 50% of health.
- *MaxHealthPotion :* Increases the player character's maximum health from 100 to 150.
- *StrengthPotion :* Increases the player character's weapon damage by 50%.

The inventory UI is designed to hold six unique items and can be expanded to hold more.

### 4.5.3 Misc Features

The character models in the project were freely acquired from the Unreal Engine 4 marketplace. Theses character assets came form an asset pack called 'Infinity Blade: Warriors' The weapon models in the project were freely acquired. These weapon assets came from an asset pack called 'Infinity Blade: Weapons'. The level design models in the project were freely acquired through the asset packs called 'Infinity Blade: Firelands' and 'Infinity Blade: Effects'.

# 5 System Evaluation

### 5.0.1 Procedural Level Generation

The procedural level generation system is efficient at dynamically generating a set of rooms that are all connected by hallways. It was crucial that every room was accessible in some way to the player. This system was primarily tested by running the "Dungeon" map in simulation mode, in the Unreal Engine 4 editor. This mode runs any gameplay functions not directly associated with the player controls. We positioned the editor camera in a bird's-eye view (see Fig. 20), and reran the simulated game mode hundreds of times over the course of development. This enabled us to constantly test and ensure that the key requirements of the procedural level generation were being met.

### 5.0.2 Networking

The networked multiplayer system works individually. However, it is not totally robust and complete when integrated with the other core systems, namely the procedural generation. The system currently enables players who have a steam account, to host, and join games with other steam users. This portion of the system works for the most part, although we were limited to testing a max of two players. Two players can join a game together and see each other's movements replicated across the network. However, integrating this system with the procedural level generation was fraught with issues. Chiefly, it appears that each client generates their own procedural map. This means that both players are seeing a totally different layout; an obvious failing of the networking portion of the project.

### 5.0.3 A.I.

The A.I. enemy characters contain behavioural logic independent of the player's input. They possess the ability to patrol the Dungeon map freely, navigating around the Dungeon's lava. Likewise, it was crucial that every room was accessible in some way by the enemy character. The A.I. controlled enemy characters were tested by running the Dungeon map in simulation mode. This allowed the viewing of the 'Navigation Mesh' (in green) which enables the A.I. controlled enemy character to navigate and move around the map (see Fig. 21). Next, the A.I. controlled enemy character was tested

again by running the Dungeon map in the standard mode. This allowed the player character to be controlled. Debug mode was enabled and allowed the viewing and testing of the A.I. controller component 'AISense' (see Fig. 22). As development continued for the A.I. enemy character, we returned to testing both the navigation and AISense frequently.

## 5.1 Performance Metrics

Frame Per Second (FPS), Network Latency (ms) and RAM usage (mb) are some of the more straightforward metrics for measuring the robustness of a game. The Unreal Engine 4 editor provides insights into these metrics, which can be seen in real time during gameplay. The following statistics were gathered by taking a snapshot of the metrics at four typical points in gameplay. These include the start of the game, mid-way through idle, mid-way through in combat, and at the end of the game. This was repeated three times to get averages for all three metrics.

| Activity | Frames Per Second (FPS) | Latency (ms) | RAM Usage (mb) |
|---|---|---|---|
| **start** | 39 | 25 | 782 |
| **mid idle** | 60 | 16 | 798 |
| **mid combat** | 59 | 16 | 791 |
| **end** | 60 | 16 | 789 |

*These metrics were gathered on a PC with the following specs:

**Processor :** Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
**Installed RAM :** 16.0 GB (15.7 GB usable)
**System type :** 64-bit operating system, x64-based processor
**GPU :** NVIDIA GeForce GTX 1650

According to the Steam Hardware and Software Survey [7], one of the largest ongoing surveys of computer gamer hardware, 45.52% of users have at least 16GB of RAM and nearly all users have some for of NVIDIA GPU. The most are popular GPU is the NVIDIA GeForce GTX 1060, a comparable graphics card in specs to the one used in testing.

## 5.2 Limitations

The biggest limitation of our project would be the completeness of the application as an overall game product. The individual systems like networking, procedural generation, and A.I. work very effectively in isolation, as was demonstrated in the above section. However, when integrating these systems, our objectives were not fully met.

## 5.3 Opportunities

There is a plethora of avenues for expansion upon the systems that were developed as part of this project. The procedural generation, for example, has a great deal of scope for broadening the variety of enemy types, weapons, and items that are populated in the world. The inventory system was built in a way that promotes extensibility, enabling future development of new item types for the game. This is the also case for the weapons and combat systems. More branches could be added to the behaviour tree of the enemy A.I. to diversify and improve the perceived enemy intelligence. Behaviour trees are inherently extensible due to their tree like structure. Tacking on additional branching nodes is certainly an area of interest. Fully integrating the networked multiplayer with the other core systems would be the most pressing opportunity for expansion.

# 6 Conclusion

Modern games apply a vast range of computer science concepts, from graphics shaders to fuzzy inference. They merge many of these computer science aspects to produce an engaging experience that people have come to know and love. Games are an interactive media that provide a perfect environment to apply potentially obscure computer science theories, with compelling and tangible results. In this project, we set out to research and explore new technologies to produce an application that adheres to the game development industry standards. Our aim was to create a game that incorporated the core features of networked multiplayer, procedural level generation, A.I. controlled enemies, and other minor gameplay systems. It is our belief that we have predominately achieved our project objectives. We have produced a well-rounded game that combines procedurally generated levels with A.I. driven enemies. Although not fully robust, we managed to implement a networked multiplayer matchmaking system with limited networked gameplay. In addition to these core objectives, we managed achieve other minor gameplay systems. These include locally persistent character customization, inventory item & weapon systems, combat & animations, and online hosted scoreboards.

## 6.1 Project Outcomes

When we set out our project objectives, as outlined in section 1.2, the overarching goal was to immerse ourselves with technologies with which we had no prior experience. Complementary to this we both share an avid interest in games and their development.

Outcomes we achieved:

- A working Unreal Engine 4 application
- Procedurally generated levels
- A.I. driven enemies with authentic behaviours
- Networked multiplayer matchmaking and session management
- Limited networked gameplay

- Real time combat mechanics with targeted sparring
- Inventory system with stackable consumable items and equipable weapons
- Online scoreboard hosting player scores
- Customisable character options persisted in local storage
- Fully animated character actions

## 6.2 Areas For Expansion

Throughout development we encountered obvious areas in need of improvement. In addition to this, we grew to enjoy and became fascinated with several features that deserve further exploration.

### 6.2.1 Publishing

Games and the game development process are a shared passion of ours at both a personal and professional capacity. Going forward, a top priority of ours would be to continue to add robustness to the existing features. Modern game development encompasses a vast array of skills and talents beyond programming. Game artwork and sound design are important areas of expertise that were not the central focus, as both of these skill sets were outside the scope of the project. Adding a finer polish to our game and potentially collaborating with other game designers for art and sound would be our primary interest. To this end, we could endeavour to create a fully polished game with the intent to publish on the popular Steam marketplace.

### 6.2.2 Procedural Generation

Procedural Generation is a fascinating subject in the context of game development. The potential for taking a niche computer science concept like 'minimum spanning trees' and using it to build entire game worlds dynamically is compelling. An obvious area for expansion would be adding a third dimension to the existing 2D level generation. This could take the form of stairways that connect multiple procedurally generated tiers. Another future development would be to add extra level biomes with environmental conditions.

### 6.2.3 A.I.

The A.I. developed for this project served as a perfect common enemy type. As it stands, the A.I. can patrol, pursue the player, attack and strafe around the player. These features do well for a common brute enemy type. However, it is a keen interest of ours to implement more specialised behaviours. One improvement we would make is to expand their awareness to interact with each other and the environment. Furthermore, we would like to implement a boss enemy type with more engaging combat. Their behaviours would be more reactive and simulate learning and countering the player's combat style.

## 6.3 Conclusion

The Applied Project and Minor Dissertation module has been a fantastic opportunity to take hands-on control of our academic career. Furthermore, it has been a fitting capstone of the last four years of college. The scope and expectations of this project, though a tremendous challenge, provided us with a positive environment in which to explore areas of computer science that interested us.

Overall, we are both delighted with the outcomes of this project and our work throughout the year. It was particularly enriching to immerse ourselves in game development. We are both of the opinion that game development provides a perfect medium for taking obscure computer science concepts and applying them to create interesting gameplay mechanics. A 'minimal spanning tree', when used to create an intricate series of connecting rooms and hallways, is an unreal way to see computer science in action.

Throughout this project we have made our fair share of mistakes. If we were to start this project again, a big change we would make would be to put in place a more concrete testing and validation framework. In addition to this, we would adhere to a more rigorous project management regime.

Finally, we both agree that we worked well together as a team. This is the second project we have collaborated together on for college. This has lead to a superb partnership in which open communication and honesty have been first and foremost. With our academic careers coming to a close and the start of our professional careers beginning, we look forward to the prospect of working together on personal game development projects in the future.

# 7 Appendix

Click For Project GitHub Repository: https://github.com/dnbutler64/FinalYearProject

# 8 Reference Images

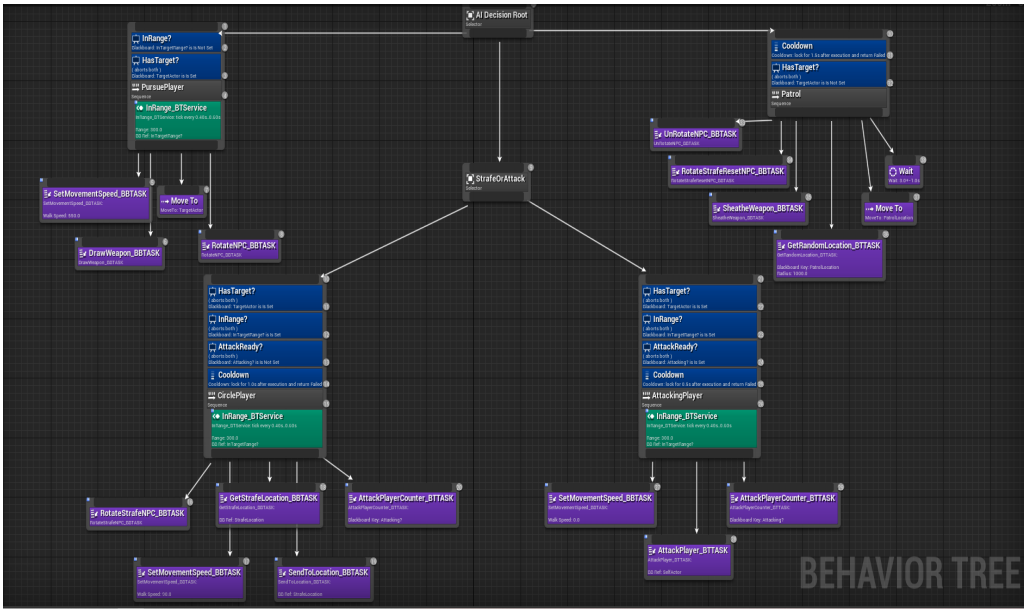


Figure 1: Full Behaviour Tree


Figure 2: Blackboard Key Info

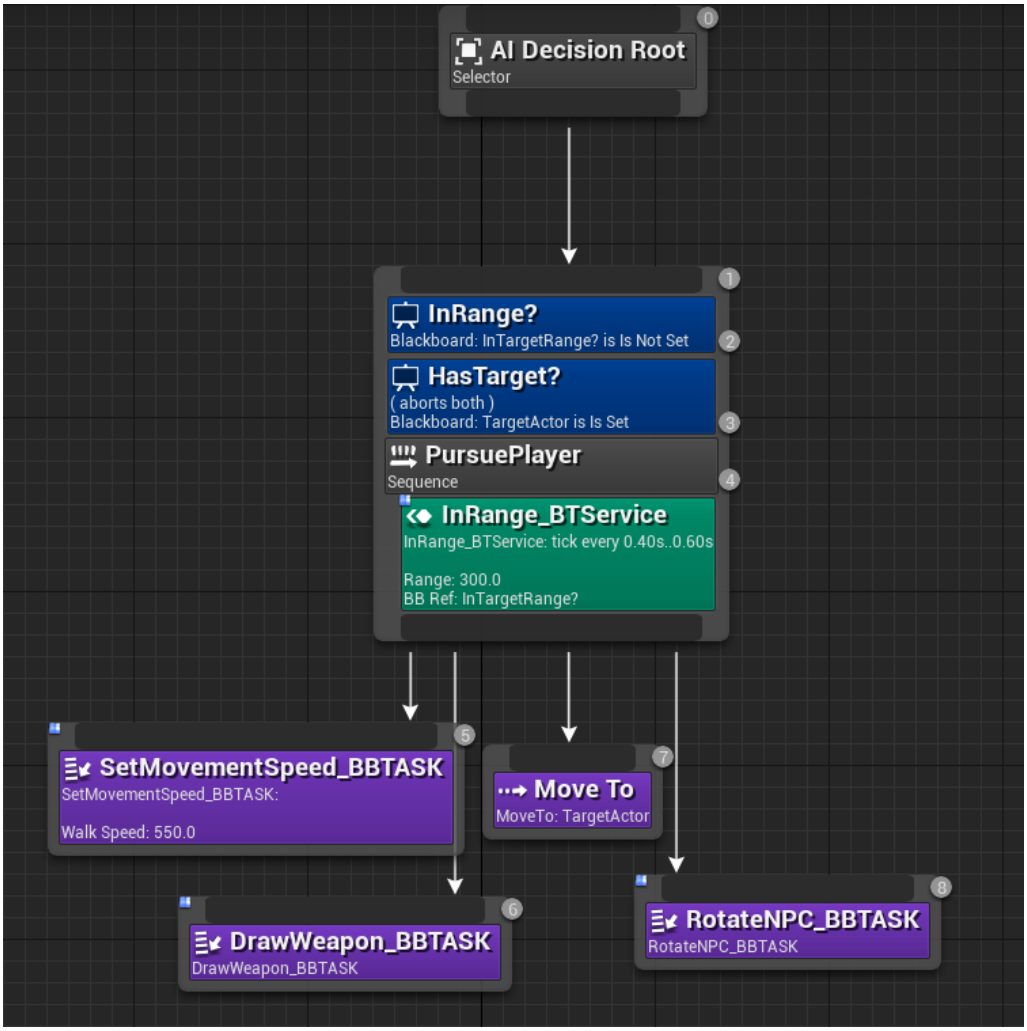


Figure 3: PursuePlayer Composite node

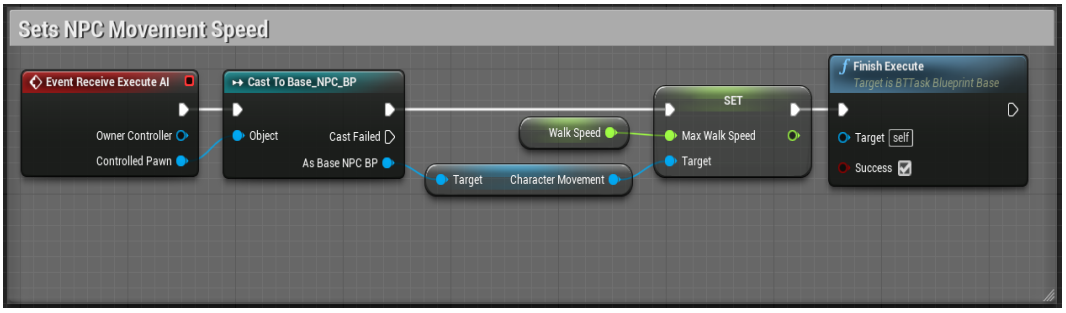


Figure 4: SetMovementSpeed

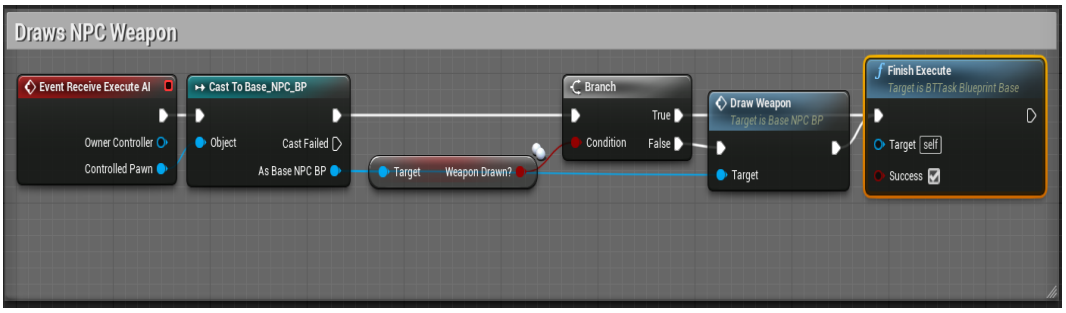


Figure 5: DrawWeapon

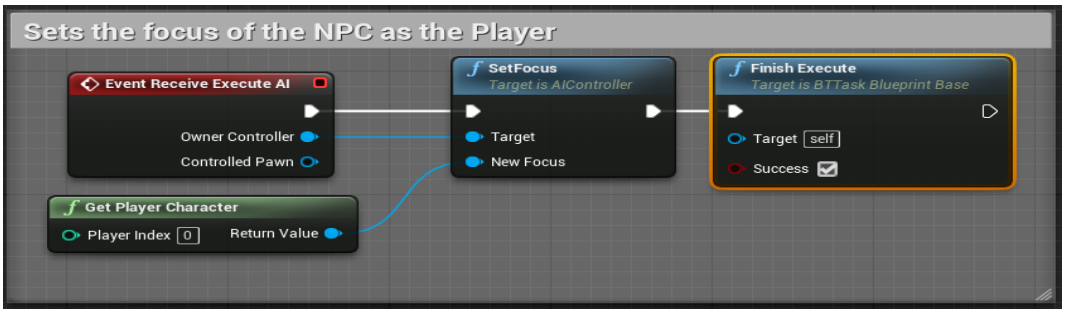


Figure 6: RotateNPC

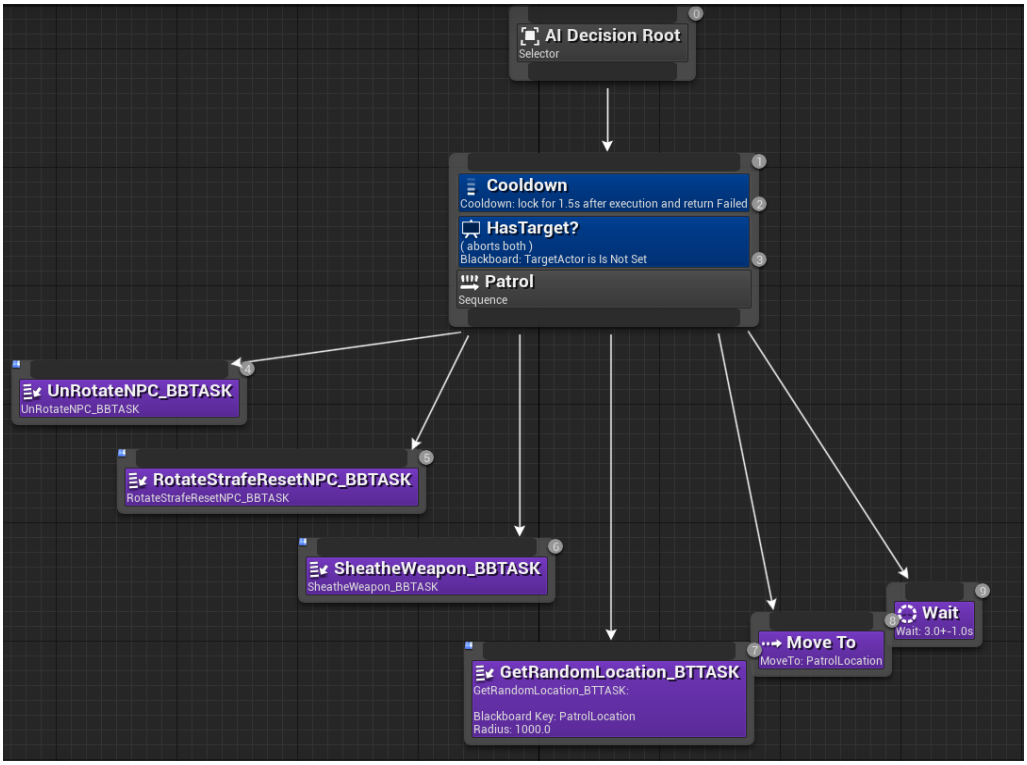


Figure 7: Patrol Composite node


Figure 8: UnRotateNPC


Figure 9: RotateStrafeReset

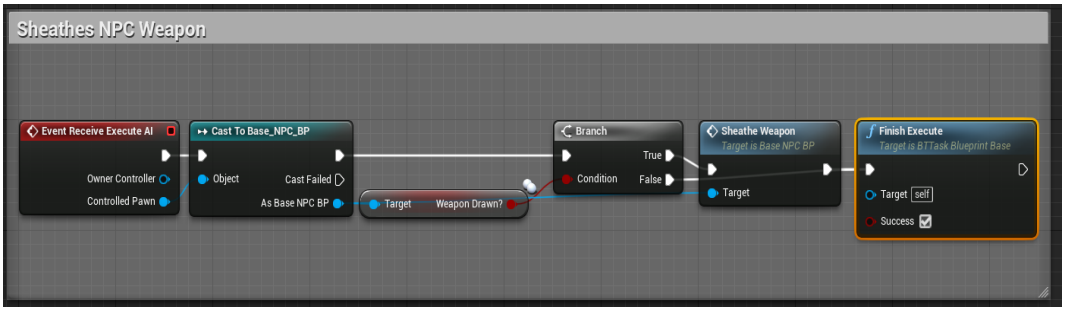


Figure 10: SheatheWeapon

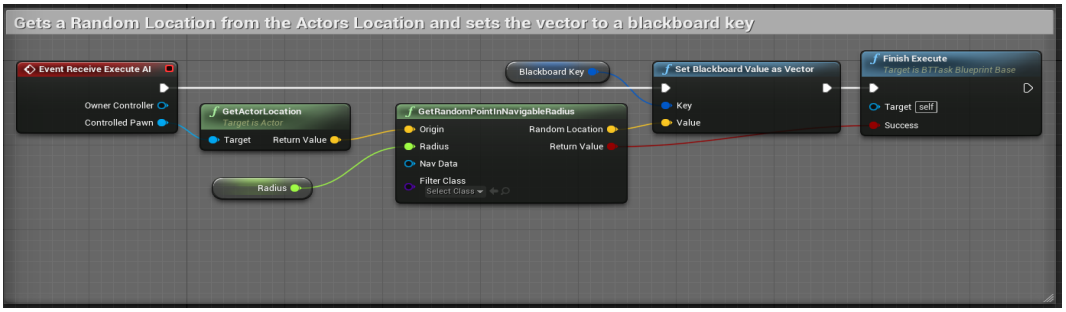


Figure 11: GetRandomLocation

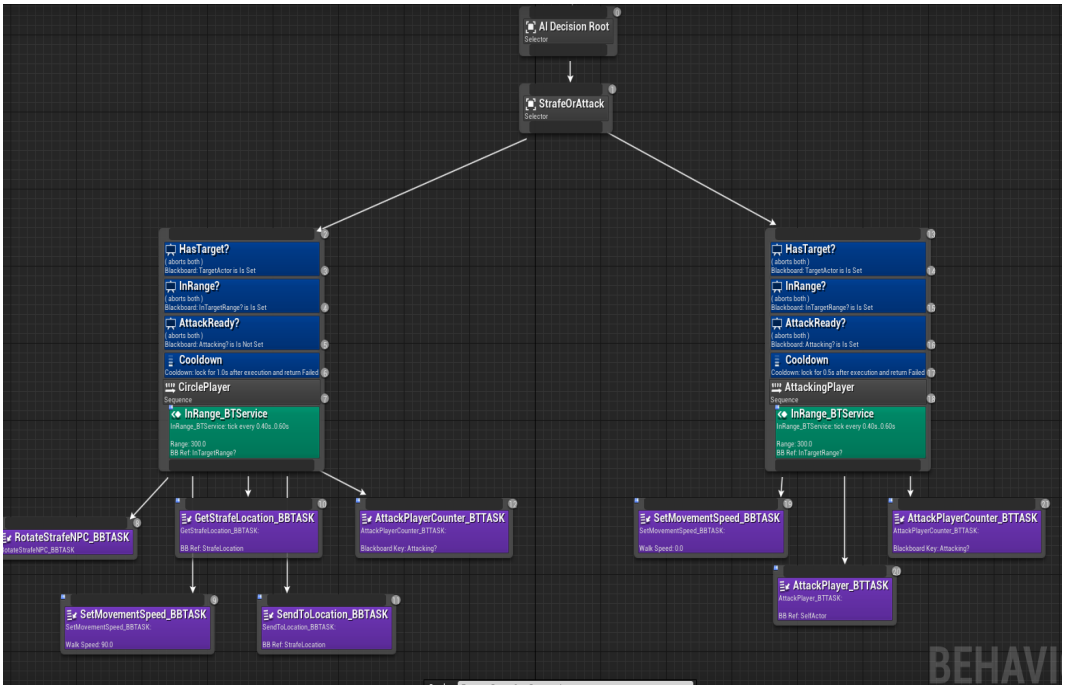


Figure 12: StrafeORAttack Composite nodes

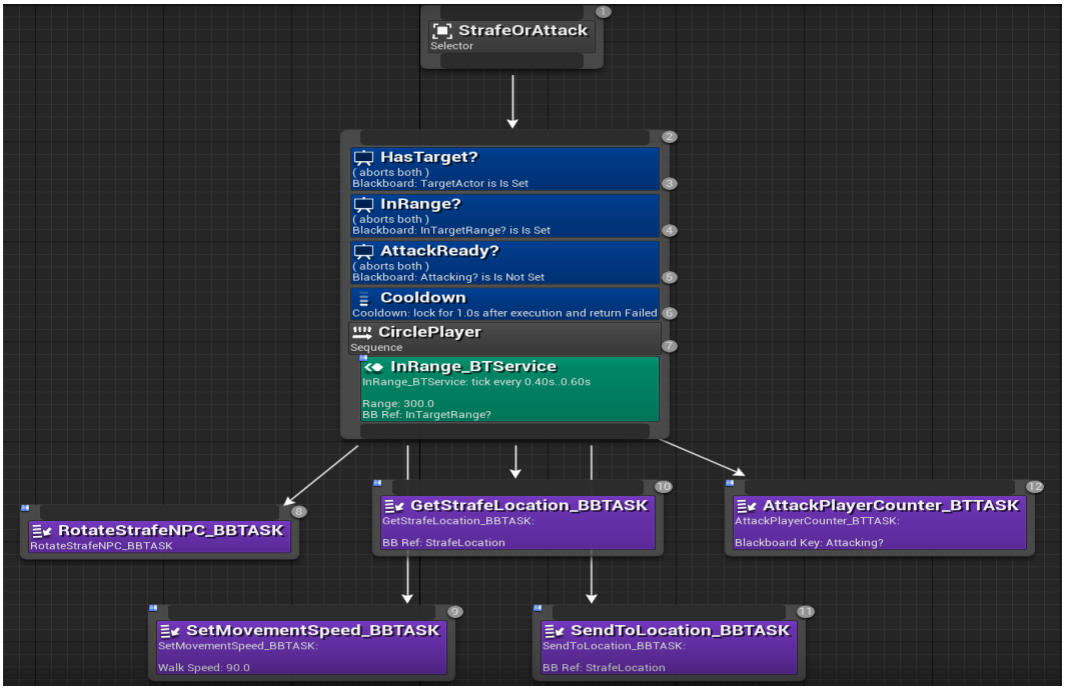


Figure 13: CirclePlayer Composite node

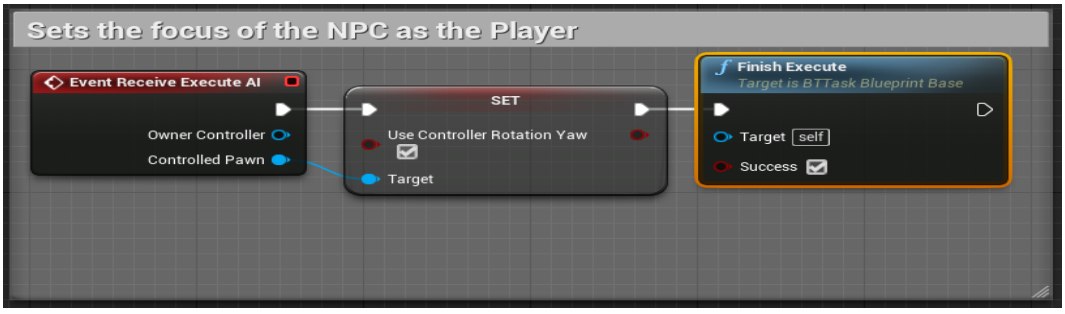


Figure 14: RotateStrafeNPC

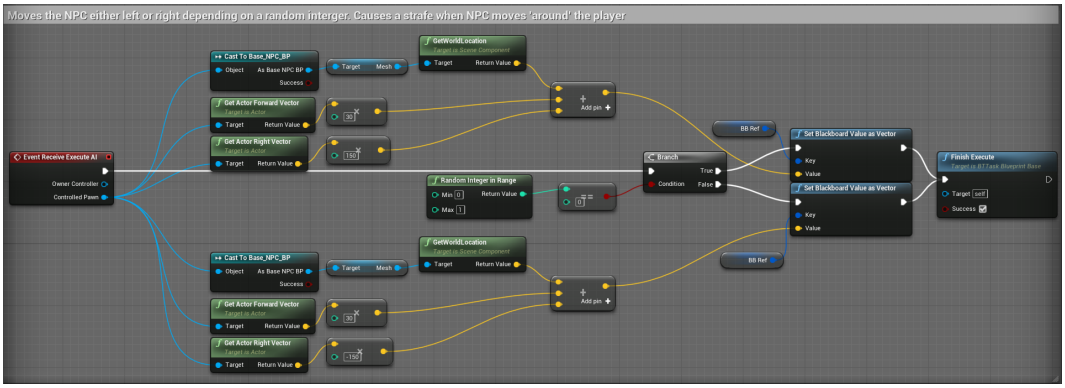


Figure 15: GetStrafeLocation

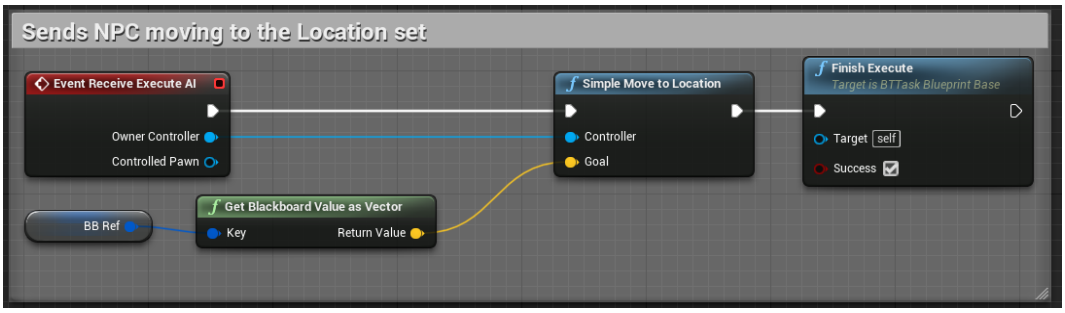


Figure 16: SendToLocation

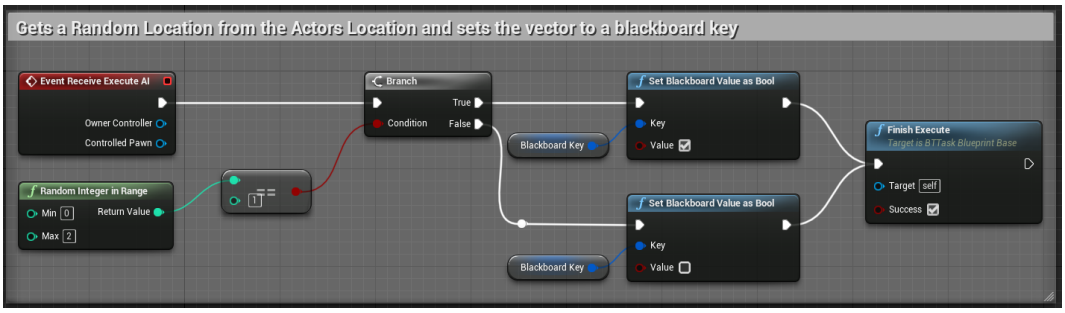


Figure 17: AttackPlayerCounter

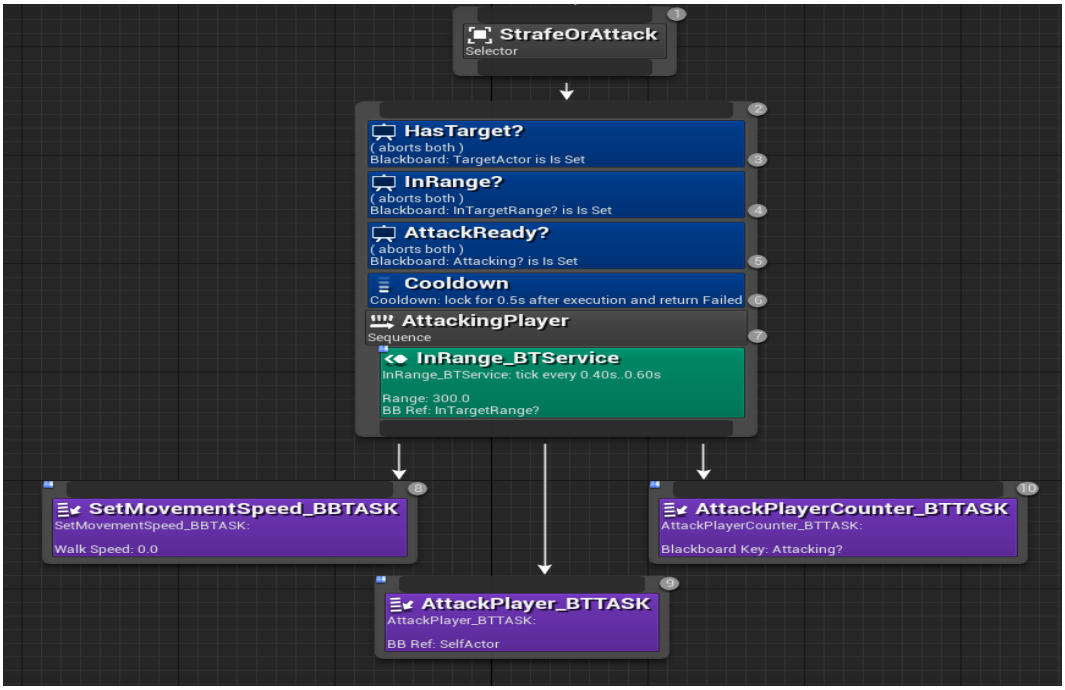


Figure 18: AttackingPlayer Composite node


Figure 19: AttackPlayerr

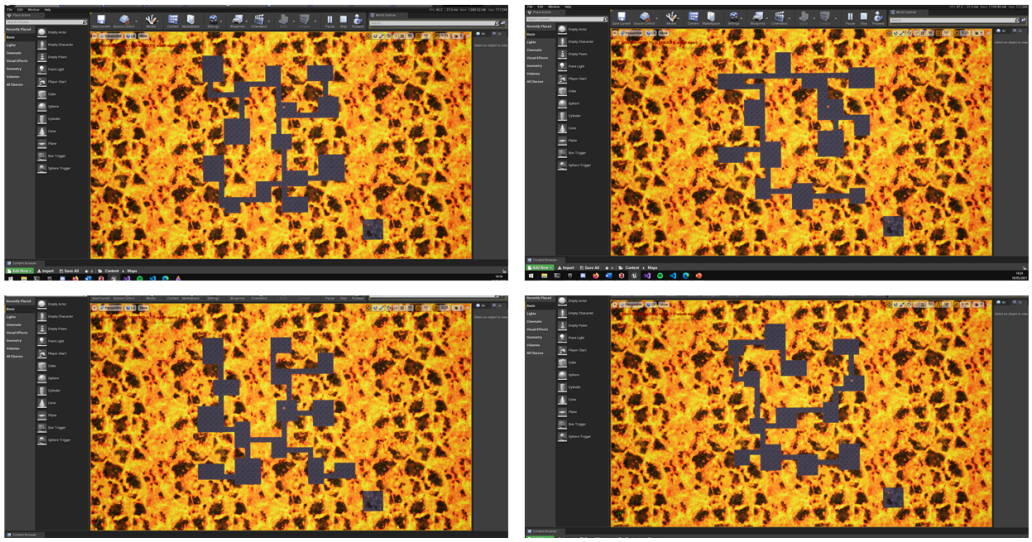

Figure 20: Multiple Instances of Procedural Level Generation Demonstrated

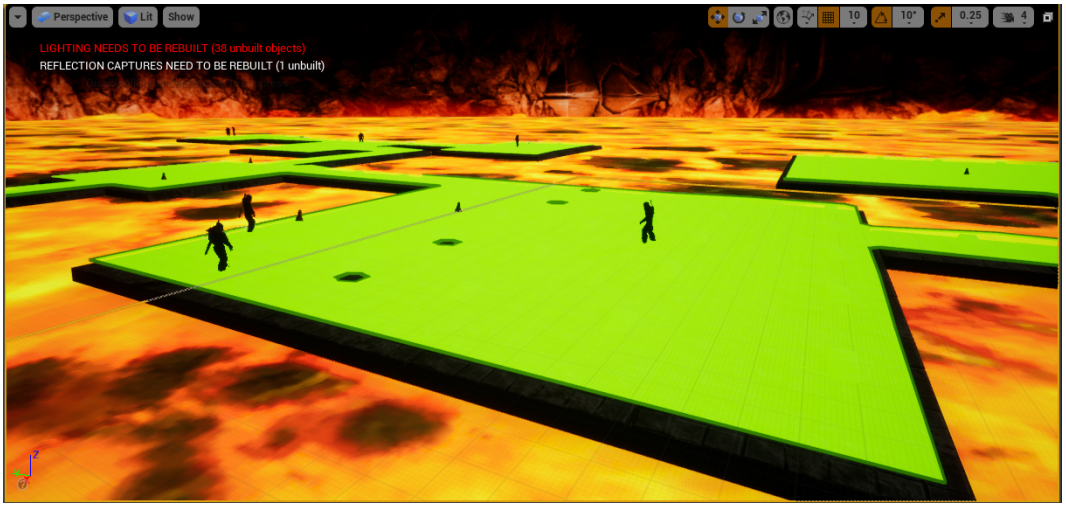


Figure 21: A.I. Navigation Mesh


Figure 22: A.I Controller

# References


[1] 7 Different Types of Game Testing Techniques - DZone Performance.

[2] Adhoc Testing - Tutorialspoint.

[3] Designing Secure, Flexible, and High Performance Game Network Architectures.

[4] Networking Overview | Unreal Engine Documentation.

[5] Online Subsystem Steam.

[6] OpenGL - The Industry Standard for High Performance Graphics.

[7] Steam Hardware & Software Survey.

[8] Topology Types | Geode Docs.

[9] Multiplayer video game, May 2021. Page Version ID: 1021928019.

[10] Prim's algorithm, April 2021. Page Version ID: 1020188658.

[11] Random walk, April 2021. Page Version ID: 1016769689.

[12] Dean James. Procedural Cave Generation with Random Walk, November 2020.

[13] Darius Kazemi. How to effectively use procedural generation in games.

[14] Chong-U Lim, Robin Baumgarten, and Simon Colton. Evolving behaviour trees for the commercial game defcon. In *European conference on the applications of evolutionary computation*, pages 100–110. Springer, 2010.

[15] Edmund Long. Enhanced npc behaviour using goal oriented action planning. *Master's Thesis, School of Computing and Advanced Technologies, University of Abertay Dundee, Dundee, UK*, 2007.

[16] Robert J. Shimonski Naomi J. Alpern. Peer to Peer Networks - an overview | ScienceDirect Topics.

[17] Diego Perez, Miguel Nicolau, Michael O'Neill, and Anthony Brabazon. Evolving behaviour trees for the mario ai competition using grammatical evolution. In *European Conference on the Applications of Evolutionary Computation*, pages 123–132. Springer, 2011.

[18] Jeff Orkin-Monolith Productions. Applying goal-oriented action planning to games.

[19] Penelope Sweetser and Janet Wiles. Current ai in games: A review. *Australian Journal of Intelligent Information Processing Systems*, 8(1):24–42, 2002.

[20] Unity Technologies. Unity Real-Time Development Platform | 3D, 2D VR & AR Engine.

[21] Arthur Zuckerman. 75 Steam Statistics: 2020/2021 Facts, Market Share & Data Analysis, May 2020.