City, University of London

BSc Computer Science with Games Technology

Final Year Project Report

First-Person Stealth Game Prototype

By

Tayyab Hussain

# Table of Contents

[1. Abstract 4](#_Toc137417457)

[2. Introduction 5](#_Toc137417458)

[2.1. Description of the Problem 5](#_Toc137417459)

[2.2. Project Objectives 5](#_Toc137417460)

[2.2.1. Stealth AI 5](#_Toc137417461)

[2.2.2. Stealth UI 6](#_Toc137417462)

[2.2.3. Stealth Mechanics 6](#_Toc137417463)

[2.2.4. Movement/Traversal 6](#_Toc137417464)

[2.2.5. Balance 6](#_Toc137417465)

[2.2.6. Other Functionality 6](#_Toc137417466)

[2.3. Project Beneficiaries 6](#_Toc137417467)

[2.4. Work Performed 6](#_Toc137417468)

[2.5. Limited Scope 7](#_Toc137417469)

[3. Output Summary 8](#_Toc137417470)

[3.1. Guard Behaviour Tree 8](#_Toc137417471)

[3.2. Group Searching and Attacking Algorithm 8](#_Toc137417472)

[3.3. Detection System 8](#_Toc137417473)

[3.4. State Depiction System 9](#_Toc137417474)

[3.5. Hiding Mechanics 9](#_Toc137417475)

[3.6. Traversal 9](#_Toc137417476)

[4. Literature Review 10](#_Toc137417477)

[4.1. Introduction 10](#_Toc137417478)

[4.2. Artificial Intelligence 10](#_Toc137417479)

[4.3. User Interface 10](#_Toc137417480)

[4.4. Mechanics 11](#_Toc137417481)

[4.5. Traversal 11](#_Toc137417482)

[4.6. Balance 11](#_Toc137417483)

[4.7. Conclusion 11](#_Toc137417484)

[5. Method 12](#_Toc137417485)

[5.1. Stealth AI 13](#_Toc137417486)

[5.1.1. Guard Detection (Objective 1C) 13](#_Toc137417487)

[5.1.2. Guard Pathfinding Search (Objective 1B) 14](#_Toc137417488)

[5.1.3. Guard AI Behaviour Tree (Objective 1A) 14](#_Toc137417489)

[5.2. Stealth UI 15](#_Toc137417490)

[5.2.1. UI Depicting Guard State (Objective 2A) 15](#_Toc137417491)

[5.2.2. UI Depicting Guard Detection (Objective 2B) 15](#_Toc137417492)

[5.3. Stealth Mechanics 16](#_Toc137417493)

[5.3.1. Two Stealth Mechanics (Objective 3A) 16](#_Toc137417494)

[5.3.2. Limitng the Use of the Stealth Mechanics (Objective 3B) 16](#_Toc137417495)

[5.4. Traversal (Objective 4A) 17](#_Toc137417496)

[6. Results 18](#_Toc137417497)

[6.1. Stealth AI 18](#_Toc137417498)

[6.1.1. Guard Detection (Objective 1C) 18](#_Toc137417499)

[6.1.2. Guard Pathfinding Search (Objective 1B) 19](#_Toc137417500)

[6.1.3. Guard AI Behaviour Tree (Objective 1A) 20](#_Toc137417501)

[6.2. Stealth UI 21](#_Toc137417502)

[6.2.1. UI Depicting Guard State (Objective 2A) 21](#_Toc137417503)

[6.2.2. UI Depicting Guard Detection (Objective 2B) 22](#_Toc137417504)

[6.3. Stealth Mechanics 23](#_Toc137417505)

# Abstract

Third-Person Stealth Games have been a staple in the Games industry for over a decade. Games such as Hitman 3(IO Interactive, 2021) and Splinter Cell: Blacklist(Ubisoft Toronto, 2013) are regarded by many as some of the best games ever made, not just in terms of stealth. Naturally, game studios attempted to replicate this success in the form of First-Person Stealth Games/Levels. This was largely unsuccessful. A market gap exists for an intriguing and tense First-Person Stealth Game/Level.

This project aimed to develop a First-Person Stealth Game Prototype that would implement some of the features which make blockbuster Third-Person Stealth Games great. To do this, the project was divided into five key areas to explore, Artificial Intelligence, User Interface, Game Mechanics, Movement and Traversal and Game Balance.

# Introduction

## Description of the Problem

Stealth games have been highly prevalent in the last decade. Games like Hitman 3(IO Interactive, 2021), Batman Arkham Knight(Rocksteady Studios, 2015), Metal Gear Solid V: The Phantom Pain(Kojima Productions, 2015) and Splinter Cell: Blacklist(Ubisoft Toronto, 2013) are all exceptional examples of good stealth games. However, these are all Third-Person games. There is a severe lack of First-Person Stealth games. Dishonoured 2(Arkane Studios, 2016) is the only relatively recent game that meets the First-Person Stealth criteria.

There have been many attempts by games such as Call of Duty Modern Warfare(Infinity Ward, 2019) and Battlefield 1(DICE, 2016) to include stealth levels within their FPS games. However, these often feel shoehorned in. A few areas in which these games need to improve compared to their Third-Person counterparts include a poor stealth AI, an unintuitive UI design for stealth, a lack of stealth-specific mechanics, limited movement/traversal options and a lack of balance between the player and the enemies.

This project aimed to create a prototype First-Person Stealth game that would incorporate a few of the features that make Third-Person Stealth games great. The specific areas the project looked at were the stealth AI, stealth UI, stealth mechanics, player movement/traversal and balancing the player and enemies.

## Project Objectives

The project’s main objective was to create a First-Person Stealth level using Unity. This included five main aspects, AI, UI, Mechanics, Movement and Balance. The exact objectives were defined as follows:

### Stealth AI

1. The stealth AI shall be designed with a behaviour tree with at least five states for the enemy. E.g., Cautious, Search, Attack, etc.
2. The AI shall use a pathfinding algorithm to find the player when in the searching state.
3. The AI shall not have a binary detection of the player. The AI should slowly detect the player over time and not instantly go into a state of alert when they spot the player.

### Stealth UI

1. There shall be a small icon to allow the player to determine the current state of an enemy. E.g., Red for an attack state and amber for a search state.
2. There shall be a UI element allowing the player to determine whether they are about to be spotted.

### Stealth Mechanics

1. There shall be two different stealth mechanics/gadgets that the player can use in the level. E.g., Binoculars to mark targets, Agent 47`s piano wire(IO Interactive, 2021), and Sam Fisher`s fibre optic cable(Ubisoft Toronto, 2013).
2. The player shall only be able to use the mechanic a finite number of times within the level.

### Movement/Traversal

1. The player shall have a unique and original way to traverse the level that differs from walking, running, crouching, and crawling prone. E.g., Batman`s Grapnel (Rocksteady, 2015) and Dishonoured`s Blink ability (Arkane Studios, 2016).

### Balance

1. Enemies shall be much stronger than the player in terms of health and damage they can do.
2. The weapons the player can use shall be very weak when taking on multiple enemies.
3. The player shall not regen health or have any way to replenish health in the level.

### Other Functionality

1. Main menu and pause menu.
2. Audio and Visual FX.
3. Saving and Loading.

## Project Beneficiaries

The primary beneficiary of this project was other developers specifically working on FPS games who wanted to implement a stealth level into their game. The project should give them a working prototype of a stealth level better implemented using some prominent features used successfully in Third-Person Stealth games.

Another beneficiary of the project was developers who may have wanted to build on top of the prototype to produce a full First-Person Stealth game.

## Work Performed

The project was developed using an Agile Development methodology. The development was split into stages based on the primary objectives, AI, UI, Mechanics, Traversal, Balance and Other Functionality. The stages were then sorted by their importance to the project’s success. Therefore, AI and UI were developed first, and Other Functionality was left until the end of development. This was done to ensure that if there were time overruns, the essential features of the prototype had been completed.

The features were designed, implemented and tested for each stage of the project. If the feature did not meet the objective, it was redesigned, implemented and tested again. Use case Requirements and Use Case testing were utilised to design and test the features. This worked effectively when paired with the Agile methodology as it allowed for a feature to be easily redesigned and retested if it did not meet the objectives or if the design was not up to scratch and the feature had to be improved.

## Limited Scope

The scope had to be limited in development due to time constraints. It was underestimated how much time it would take to implement parts of the prototype that were not vital to the project. Things such as a moderately comprehensive player controller and a much larger level design to incorporate some ziplines for the Traversal objective took time. Before starting the project, this time was not initially accounted for in the work plan. However, thanks to how the work plan was assembled, with the most vital stages of the prototype being completed at the beginning, the only stages removed from the development were the Balance and Other Functionality.

The scope of objective 1a regarding the Guard AI`s Behavior Tree was also limited. The objective entailed having a minimum of five key states for the Guard to be in. However, it soon became apparent that having four key states was comprehensive enough and that having another state for the Guard to be in would be unnecessary and likely be of a lesser quality than the previous four.

# Output Summary

## 3.1. Guard Behaviour Tree

The first output is the Behaviour Tree which controls the actions and states of each Guard in the prototype. It is software code coded in C# consisting of 15 classes and 904 lines of code(not including comments), of which I wrote 698 and 206 were re-used as detailed in the Results section. The intended recipients of the output are future developers. They can adapt the basic framework of the current Behaviour Tree for use in their games, or they can use this Guard Behaviour Tree as a base and build upon it.

## 3.2. Group Searching and Attacking Algorithm

Objective 1b entailed developing a system for the AI to use pathfinding to search for the player. For this, a searching algorithm was required to tell the AI guards where to pathfind to, to make it look like the Guards were actively searching for the player as a collective. Later this developed to also house functionality that would allow the Guards to all organise an attack together if one of the Guards spotted the player. This output was a single class I wrote in C#, consisting of 184 lines. The intended recipients of this output include future developers who can take this simple yet effective searching algorithm and make it more complex and more efficient for use in their games.

## 3.3. Detection System

The AI required functionality to spot the player slowly over time. To do this, a system was created to implement a vision cone for a Guard and then split that vision cone into five different vision zones. These vision zones would detect the player at varying speeds based on how close the player was to the Guard. The output was two classes I wrote in C# consisting of 280 lines of code.

Another aspect of the detection system was implemented in the UI stage of development. There was a UI element that would depict how close the player was to being detected based on the previously mentioned detection speed. The output consisted of two main parts. The first part was C# code consisting of 2 classes and 60 lines of code, of which I wrote 41 and 19 were reused as detailed in the Results section. The second part of the output was the sprites that would be used to display the detection amount. Two sprites were initially created as .png and then converted in Unity to sprites. I made one sprite using Piskel, and another was found online, and both were a combined size of 177KB. The sprites were detailed further in Appendix B.

The intended recipients of this output are the same as the previously mentioned future developers. With this output, they can easily modify the detection area and how quickly the player can be detected to suit their need for their specific games. The sprites can also be treated as placeholders and replaced by a more visually appealing design. The complete output is detailed in Appendix B.

## 3.4. State Depiction System

To inform the player of what state a Guard is in, a UI element was designed to depict the current state of a Guard. This output was simple and only depicted three of the four AI states. These included a UI element for the Searching state and a UI element for both the Chasing and Attacking states, called the Alerted state. The output was split into two parts. The first was C# code to enable and disable the UI elements. These were divided into two separate classes, along with a third class which handled turning the UI elements into a billboard. This last class was re-used, as mentioned in the Results section. From the three classes, 60 lines of code were written, 41 by me and 19 were re-used.

The second part of this output came in the form of sprites. These were used as the actual visual representation of a Guard’s state. As detailed in Appendix B, I designed the Searching and Alerted sprites using Piskel, and both sprites came to a total size of 201KB.

Future developers are the intended recipients of this output, and they can look to build upon this output by implementing more AI states and then complementing those states with more UI elements to depict those states. Future developers may also want to keep the current implementation of the Ui but alter the graphic design of the sprites to make them more visually appealing. The complete output is detailed in Appendix B.

## 3.5. Hiding Mechanics

A simple yet efficient and easy method of implementing hiding mechanics was implemented into the prototype. The output consists of two simple parts. The first is a layer mask called Obstacle. I added this to the layers section in the Unity Inspector Window. This allows future developers to choose an element they want the Guard not to see through, and it will enable the developers to quickly create a vast array of places the player can use to hide. The second component of this output consists of a single line of code embedded into a much larger class, coded in C#. The total amount of code in the class comprises 127 lines, and only one line is required for this output. I wrote the entire class. Future developers can also make changes here, they only have to change one line, and their AI will not be able to see through any other layer mask they choose. The complete output is detailed in the Results chapter.

## 3.6. Traversal

For objective 4a, a zipline was implemented into the prototype to allow the player to traverse the level more dynamically. This implementation consisted of 2 C# classes, both of which I wrote. The code amounted to a total of 98 lines. The intended recipients of this output are future developers who can build upon this feature and implement more complex zipline physics, animations, and a more polished output than the one in this prototype. The complete output is detailed in the Results chapter.

# 4. Literature Review

## 4.1. Introduction

Stealth games have been one of the most popular genres of video games since the turn of the millennium. Games such as Metal Gear Solid(Hideo Kojima, 1998), Tom Clancy`s Splinter Cell: Blacklist(Ubisoft Toronto, 2013) and Hitman(IO Interactive, 2016) defined the genre. An overwhelming majority of these games are in the Third-Person. The only notable recent exception is Arkane Studios` Dishonoured 2(Arcane Studios, 2016). There have been many attempts by FPS studios to include a stealth level within their action-packed games, but these often feel shoehorned in. For my project, I made a prototype First-Person Stealth game that sets out to achieve what most FPS Stealth games/levels are missing. In this literature review, I will compare and contrast five themes: AI, UI, Mechanics, Traversal and Balance in Third-Person Stealth and First-Person Stealth games/levels. This allowed me to find gaps in current FPS stealth levels that my prototype aimed to fill.

## 4.2. Artificial Intelligence

A stealth-based AI is a critical component of a stealth game. An AI that the user perceives to be unintelligent will make the game less intense and provide less of a challenge for the player. One essential part of a comprehensive AI is how the AI detects the player. In Splinter Cell: Blacklist(Ubisoft Toronto, 2013), the team used vision cones and vision zones(Walsh, M. 2014) to allow the player to be slowly detected over time based on the player`s location within the enemy’s vision cone. This is an example of great architecture for stealth AI and provides a realistic interpretation of how someone would spot a foreign entity in real life. In contrast, Metal Gear Solid(Hideo Kojima, 1998) has a binary detection system. This means that whenever the player is within the enemy’s vision cone, they are instantly detected. This could be very frustrating to the player as even if the player`s arm was barely visible for a split second, the enemy would go into an attack state. This makes AI seem unrealistic and unfair. However, unlike some games, Metal gear solid(Hideo Kojima, 1998) does allow the player to hide after being spotted and return to the stealth aspect of the game. This is evidence of an in-depth implementation of a behaviour tree or finite-state machine(FSM) for AI(Millington, I. 2019). To sum up, the AI needs a non-binary detection system and a behaviour tree or FSM that allows the player to escape after being spotted.

## 4.3. User Interface

Stealth games must have an intuitive UI design. A good UI can help the player understand what state the enemies are in, if the player is hidden and how close the enemies are to spot the player. Dishonoured 2(Arcane Studios, 2016) has an excellent UI that uses markers above the enemy’s head to depict the enemy`s current state. If we compare this to Call of Duty: Modern Warfare(Infinity Ward, 2019), we find that during the mission ‘Going Dark’, the UI gives the player no indication of what state the enemy is currently in. This does not give the player confidence and can make the player play safer and not explore the level to its fullest. Overall, it is crucial to have an intuitive UI that helps the player understand what state the enemy is in.

## 4.4. Mechanics

A game will not be fun if its mechanics are boring and unbalanced(Adams, E. and Joris Dormans, 2012). Games are made by how good their mechanics are, and stealth games are no different. This is most clearly seen during the Battlefield 1 mission ‘Fall from Grace’(DICE, 2016). During this mission, there are only two stealth mechanics; both are overused and unbalanced. The first is throwing an item to take the guard’s attention, and the second is destroying a communication box to stop reinforcements. Not only are they overused, but they are also unbalanced. The player can find things to throw to distract the guards all over the level, and disabling the communications boxes provides no real challenge. Therefore, mechanics in stealth games must be original and balanced to prevent the player from abusing the mechanic.

## 4.5. Traversal

Traversal is a vital component of any good game. The ability to traverse the level uniquely provides more replayability for a level. Stealth games have been using unique forms of traversal for a long time. Examples include Sam Fisher`s split jump in Splinter Cell: Chaos Theory(Ubisoft, 2005) and Batman`s grapnel in Batman: Arkham Asylum(Rocksteady Studios, 2009), which the player can use to climb above the guard’s line of sight or use ventilation shafts/grates to traverse the level below the guards’ feet. This kind of traversal is largely missing in First-Person Stealth games/levels. An example is Battlefield 1`s level ‘Fog of War’(DICE, 2016). In this level, the player has no unique ways to traverse the map. There is no way to change the verticality or manoeuvre around enemies. This leaves this level feeling a little flat. Ultimately, traversal is necessary for all games, but the lack of unique traversal in stealth games can leave the player with no opportunities to tackle a level in an original way.

## 4.6. Balance

A balance between the player and enemies is even more vital in stealth games than in regular games. In stealth games, if the player feels like they can win a shootout against a large number of enemies, it defeats the point of the player trying to be stealthy. The designers made a single enemy lethal in The Last of Us(Naughty Dog, 2013). This forced the player to play stealthily. However, this alone would not be balanced when the player was fighting a large group of enemies in a non-stealthy environment. To counterbalance this, when the player was fighting a large group, only one or maybe two enemies would shoot at the player simultaneously (McIntosh, T. 2014). This meant the player was still cautious of the enemy, but at least it gave them a fighting chance. Getting the balance right in a game is a tedious and lengthy process. Getting it right in a stealth game is just as hard. However, doing it correctly means the player plays the game in the way intended for them by the developer, thus resulting in a much more fun experience.

## 4.7. Conclusion

To conclude, there is a multitude of things that FPS stealth games/levels can learn from their Third-Person counterparts. Many FPS stealth levels incorporate one or maybe two good practices seen in Third-Person Stealth games, but the only First-Person game that encapsulates all these practices is Dishonoured 2(Arcane Studios, 2016). With the knowledge from this literature review, I focused my prototype on the specific areas that FPS stealth games/levels are in dire need of.

# 5. Method

This project was developed using an Agile methodology. Specifically, each primary objective was run in an iterative design, implementation and testing loop. This ensured that each objective was completed to a good quality before moving on.

The build plan was initially split into six major sections and was structured so that the most vital parts of the project were completed first. This ensured the excellent quality of the most critical aspects of the projects, which was crucial so that the prototype at the end of development highlighted the best bits to potential beneficiaries. However, the build plan was changed during development, and only four of the six objectives were completed. This meant the build plan was split into four major sections. This will be detailed further in the Results section.

To track the progress of the project, a work plan was designed in the form of a Gantt chart, which was updated regularly to reflect the progress made on the project. Coinciding with the agile methodology, there was a reflection every weekend to ensure I was on target with the project, and if I was not, I could move some of the objectives around and rethink my work plan.

The methodology chapter is divided into four parts, one for each primary objective. These chapters are then divided into sub-chapters, including the design and implementation required to complete that objective. They may also include the requirements, evaluation and testing of the objective/implementation.

Requirements and testing were done using use cases. Before development, use case requirements for an objective were laid out. The implementation then sought to achieve what the requirements had specified. At the end of the implementation, use case testing was performed to ensure the objective was met.

Note that this project aims to address areas in which first-person stealth games are lacking. Graphics and animations are not one of these areas. For my project, I will not use any fancy models for characters, etc, as it would take attention and time away from me implementing key features that are lacking in first-person stealth games.

The version control used for this project was GitHub. The project was backed up to GitHub at the end of every day in which any work took place. The reason GitHub was chosen was that I had used it a lot in the past to back up my other university work and found it to be quick and easy to use.

## 5.1. Stealth AI

The following three chapters detail the methods to design, implement and test the work for project objective 1, Stealth AI. The three objectives were completed back to front. The reasons for this will be detailed in the Results chapter.

### 5.1.1. Guard Detection (Objective 1C)

The first objective was to implement the Guard`s ability to slowly detect the player over time. This was done by first implementing some essential detection. This was achieved by performing a raycast from the Guard to the player and checking that the player was within a specific view angle relative to the Guard`s forward vector(ensuring the player was in front of the Guard). The raycast would also check for any obstacles between the player and Guard by utilising a layer mask check. The next step was to check that the player was within a specific range of the Guard and that there were no obstacles in the way. A more in-depth system involved the Guard slowly detecting the player over time. To implement this, the Guard`s view cone was split into different zones. This was done by splitting the view cone into three angles, wide, medium and close. The smaller the angle, the faster the player should be detected. The next step is to add three distinct ranges within the view cone. This allows the player to be detected slower when further out from the Guard. This created 15 distinct zones categorised into 5 zone types within the Guard`s view cone, as shown in Figure 1. The last step is to set a timer for each zone type to detect the player at a different speed. To help with testing at this stage, a gizmo or spotlight to check which zone the player is currently in and if they have been spotted would be beneficial.

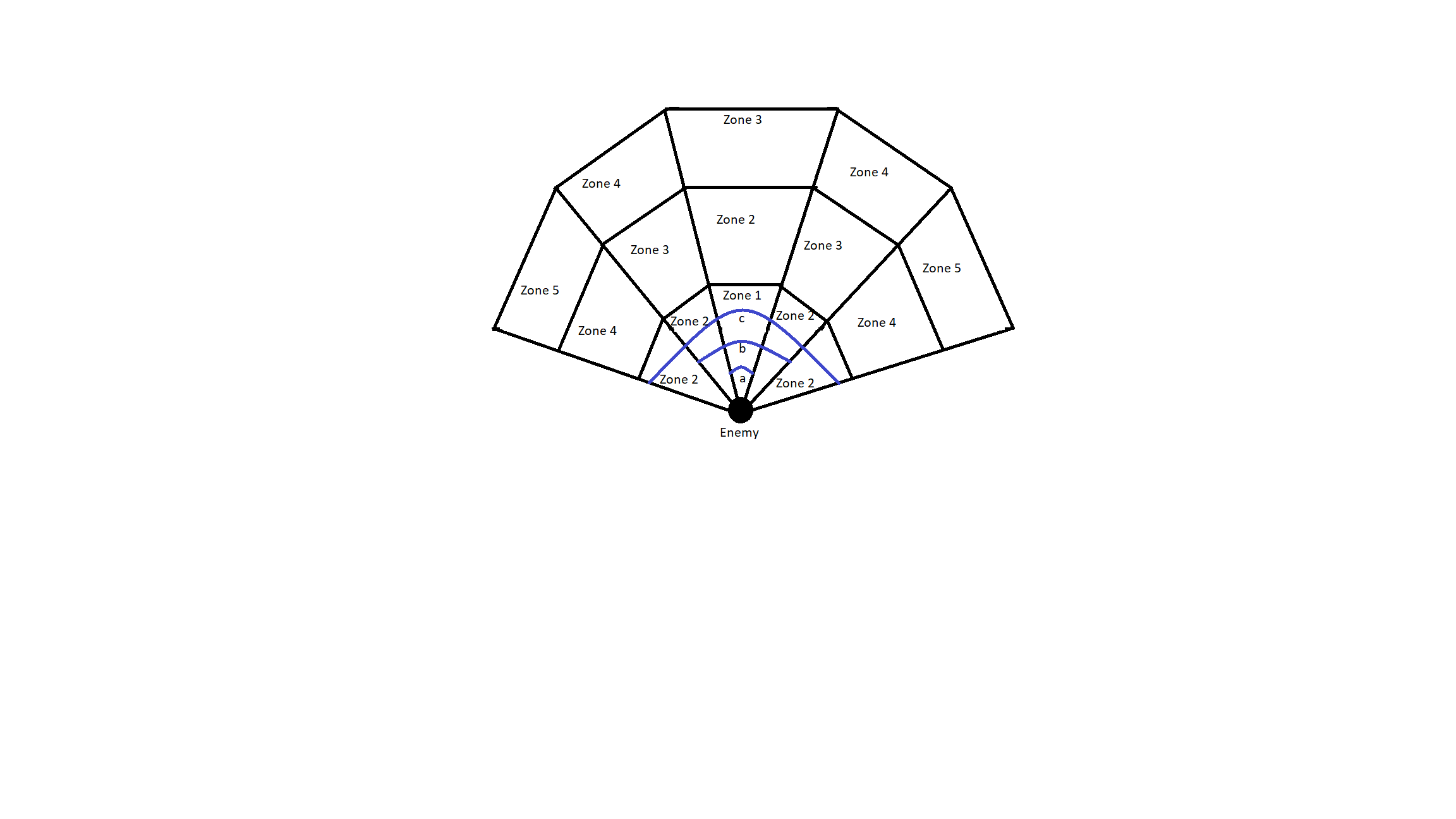


Figure 1

*An example of a Guard`s Vision Cone. Fifteen vision zones are split into five zone types. The closer a zone is to zone five, the slower the detection. 'a’, 'b' and 'c' are the close, medium and wide angles displayed with the blue lines.*

### 5.1.2. Guard Pathfinding Search (Objective 1B)

Objective 1B entailed a Guard being able to independently pathfind around the map to search for the player. The first component of this was to implement Unity`s NavMesh system. An online tutorial was used to obtain the necessary Unity package for the NavMesh and instructions on how to bake a NavMesh (Brackeys, 2018). Once a Nvamesh was baked onto the level, a Guard could be specified a location to travel to, and the Navmesh would help the Guard pathfind to that location. This Nav Mesh system implements an A\* pathfinding algorithm to find the shortest route to a given target. To implement a coherent searching algorithm using the pathfinding algorithm, all the Guards must work together to search the map. A separate search method is used, containing all Guards in a group to coordinate their search. This method is housed within the BTGuardGroup class. If one Guard spots the player, the BTGuardGroup class assigns all the other Guards in the group to convene at the player’s last known location. This makes it seem like the AI has communicated the player’s whereabouts. The search class randomly assigns each Guard to a location on the map for them to search. A timer controls the search. Each Guard has two search locations. The first search begins 5 seconds after losing the player. This was done to allow time for the Guards to convene on the player’s last known location. After searching for 20 seconds, the Guards will be assigned a new location on the map to search, and they pathfind to the new location on their own. After 15 seconds spent on the second search, the search class calls an end to the search. The Guards also use the NavMesh pathfinding system to chase the player if they spot them. The Guards will enter a patrol path if they are not chasing or searching. The patrol path utilises the NavMesh system to have a Guard constantly patrol around a fixed set of locations around the map. Each Guard has their own set of locations/patrol routes.

### 5.1.3. Guard AI Behaviour Tree (Objective 1A)

Objective 1A required the Guards to use a behaviour tree to dictate their actions. The specific behaviours include Patrol, Search, Chase and Attack. First, a generic behaviour tree architecture was implemented from a tutorial (Mina Pêcheux, 2021). The behaviour tree had four nodes. The first was a root node at the top of a behaviour tree. The second and third were Selector and Sequence nodes, respectively. A selector node works like an OR logic gate (If any child node returns success, then return success). A sequence node works like an AND logic gate (if all child nodes return success, then return success). The final node type is a leaf node where all classes/actions will be housed. A behaviour tree works in order from left to right. This is important as it allowed priority to be given to the left-most nodes since they would be the first to return. The Guard Behaviour Tree used in this project implemented four key states. These included Patrolling, Chasing, Attacking and Searching. The exact formation of this Behaviour Tree can be seen in Figure 2. The entire detailed workings of the Guard Behaviour Tree are in the Appendix.

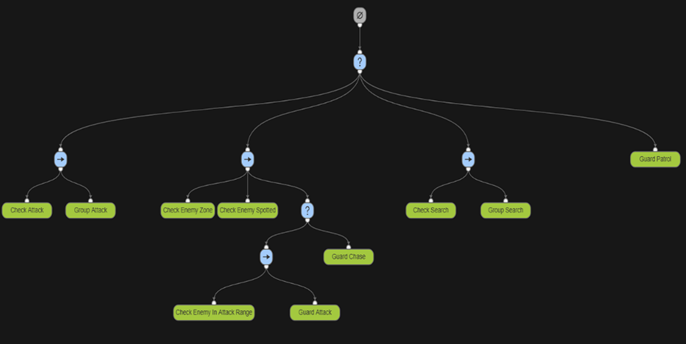


Figure 2

Behaviour Tree Diagram showing the Guard Behaviour Tree layout and classes. (The topmost node is the root node. The ‘?’ nodes are selector nodes. The ‘->’ nodes are sequence nodes. The green nodes are leaf nodes.) This diagram was designed using Adobe Behaviour Tree Visual Editor (opensource.adobe.com, n.d.)

## 5.2. Stealth UI

The following two chapters cover the methods used to design, implement and test the work for project objective 2, Stealth AI.

### 5.2.1. UI Depicting Guard State (Objective 2A)

Objective 2A involved implementing UI elements above the Guards` heads to tell the player what state the Guard was in. The game had four main Gurard states: Patrolling, Searching, Chasing and Attacking. The Patrolling state would have no UI element. The Searching state would have a question mark UI element, and the Chasing and Attacking states both had an exclamation mark UI element to depict an ‘Alerted’ enemy. The implementation involved binding these symbols to the Guards` heads and enabling and disabling them based on that enemy`s particular state. The symbols were simple sprites that could be designed in 2d design software. The specific software chosen for this project was Piskel(Descottes, 2019). The sprites were imported into Unity as assets and rendered as images under a Canvas UI game object. A tutorial was followed to add a Canvas game object and assign sprites to the Canvas (Brackeys, 2020). From there, whenever a Guard was in a state other than Patrolling, the corresponding UI element would be enabled and the others disabled.

### 5.2.2. UI Depicting Guard Detection (Objective 2B)

Objective 2B required the development of a UI element which would help the player understand how close a Guard was to spot them. The first part of the implementation involved designing and importing a detection bar into Unity and having that be another child of the Canvas UI game object. This was partly done in Piskel(Descottes, 2019), and the other part of the design was found online(creazilla.com, n.d.). Then, the detection UI element was set up to be enabled when the player was in the process of being detected and disabled if the enemies were alerted by the player or had not seen the player. A simple UI Image handled the detection level in Unity; the method to implement this was re-used from an online tutorial(Brackeys, 2020). Its position was set relative to the Canvas object, and its size was dictated by a built-in Unity component called a Slider. This component had built-in functionality, allowing the blank image to be scaled based on a pre-determined amount, in this case, the detection level. The detection level was set in the CheckGuardSpotted class by taking the time the player had been currently visible to the guard and dividing it by the total time it would take to spot the player in that particular zone.

## 5.3. Stealth Mechanics

The following two chapters cover the methods used to design, implement and test the work for project objective 3, Stealth Mechanics.

### 5.3.1. Two Stealth Mechanics (Objective 3A)

The first mechanic to implement as part of Objective 3A was a smoke bomb. The mechanic had two components. The first was a smoke bomb canister, which re-used a prefab found online (assetstore.unity.com, n.d.). Upon the player’s button press, the canister would be initialised into the level, falling just in front of the player as if they had let it go from their hand. This component was re-used from an online tutorial(Brackeys, 2017). The second component was the smoke, which was an effect found on the Unity Asset Store (assetstore.unity.com, n.d.) and had been manipulated using the Unity Particle System component to make the smoke look more appealing and do a better job of hiding the player. Once the canister had been dropped, it had a delay before the canister was culled from the game, and the smoke was instantiated. The smoke object had a box collider and the same layer mask as any obstacles in the scene. This was done so Guards could not see the player through the smoke. This implementation worked well but did not fully cover what was required in the Use Case Requirements Specification. The Results section will discuss the shortcomings and the reasons for those shortcomings.

The second mechanic to implement as part of Objective 3A was a hiding mechanic. This works predominantly the same way as the smoke bomb mechanic. The mechanic can utilise any mesh that seems realistic for a player to hide in, such as a bush or tall grass. A bush found online was used for the implementation at this stage (assetstore.unity.com, n.d.). The bush was rendered as a mesh and had a box collider component. The box collider component had the ‘Is Trigger’ boolean value set to true. This meant the physics engine ignored the box collider allowing the player and Guards to walk through the bush. The last thing to add was a layer mask. The layer mask was set to ‘Obstacle’, the same for the smoke bomb and other buildings around the scene. This ensures the Guards cannot see through the bushes and spot the player.

### 5.3.2. Limitng the Use of the Stealth Mechanics (Objective 3B)

Objective 3B involved limiting the use of the previously implemented mechanics so that they were not overpowered and the player could not solely rely on them when completing the level.

The first part of this was limiting the use of the smoke bomb mechanic. This was done in 2 aspects. The first was to implement a cooldown after deploying a smoke bomb to stop the player from repeatedly using the smoke bomb in quick succession. This was done by implementing a cooldown timer after a smoke bomb was dropped and not allowing the player to drop another smoke bomb until the timer had run down. The second aspect of limiting the use of the smoke bomb was to prevent the player from having an infinite number of smoke bombs at their disposal. This was done simply by adding a variable that would store the total number of smoke bombs and decrementing it by one every time a smoke bomb was deployed. Once the counter reached 0, the player could not drop any more smoke bombs.

The hiding mechanic cannot be limited to a finite number of uses or implement a cooldown such as the smoke bomb mechanic. In order to limit the effectiveness of this mechanic, the developer must place the hiding locations thoughtfully in the level. For this implementation, this was done by having the hiding areas at the edge of the map be large and plentiful, allowing the player to easily and safely recon the area. However, once the player moved closer to the main part of the level, the hiding areas would be smaller and fewer and far between. Finding a balance between where and how to place hiding locations will differ on a map-to-map/level-to-level basis. Each level must have its own interpretation of this implementation to ensure the player cannot simply hide in the hiding areas for most of the level.

## 5.4. Traversal (Objective 4A)

There was only one feature to be designed, implemented and tested as part of project objective 4, Traversal, in the form of a zipline.

Objective 4 entailed the design of a unique and dynamic way for the player to traverse the level. This was chosen to be a zipline. To implement a zipline, first, two locations had to be chosen in the level, one being a starting location for the zipline and one being the landing zone. In this prototype, they were stored as game objects with a rudimentary mesh attached to them designed bespoke in Unity`s ProBuilder. After this, linear interpolation was used to implement a zipline imitation of the player going between the two points. Linear interpolation in Unity can be used as “Vector3.Lerp” to transform the position of an object smoothly. It does this by taking three parameters, the start location, the end location and a number between 0 and 1. The number relates to how far along between the two points an object is. For example, if the number is closer to one, the object will be closer to the end location; if it is closer to 0, it will be closer to the start location. By setting the number of time elapsed while on the zipline over how much time the developer wants that zipline to take, a smooth interpolation can be achieved. Using this interpolation as the player’s position if they press a button can create a nice zipline effect. To ensure the zipline works accordingly, disabling the player’s movement functionality and gravity is advised, as it prevents any weird juddering or glitches. After the linear interpolation has finished, these can be reset when the player exits the zipline.

# 6. Results

The Results chapter will detail the outputs from each objective completed in the project. The chapter will be split into four main parts related to the four significant objectives completed during development.

The outputs will be categorised into two main groups, code outputs and Unity outputs. Code outputs relate to the specific code and algorithms used to implement a feature. Unity outputs relate to things such as game objects and prefabs contained within the Unity project. Prefabs are templates that a developer can use to create new instances of that prefab quickly.

The Agile methodology adopted for the project worked well. The iterative loop for objectives allowed for features to be continually improved upon until the tests were successful. The requirements and design for each project objective did not need to be changed during development.

Due to the strict timeframe for this project, the last two objectives could not be completed. Initially, the build plan was split into six major sections, but due to the timeframe, the build plan was split into four major sections. Thankfully, this did not significantly impact the project as a whole, thanks to how the build plan was laid out. Since the most vital parts of the prototype were completed first, Objective 5 and Objective 6 played a minor role in how well the project would turn out.

The use of a Gantt chart did help tremendously, and it allowed the project to better adapt to the strict timeframe. A week or so before development would begin on Objective 4, it could be seen on the Gantt chart that plans to complete all six objectives were not entirely possible. With this information, the build plan was re-organised to put less emphasis on trying to complete all six objectives and more effort was put in to ensure the first four objectives were completed to a high degree.

Use case requirements, and testing worked well in conjunction with Agile. If an implementation did not meet the criteria of the use case tests, then the implementation could be quickly reworked. It also allowed for requirements to be redone if the output from the objective was not up to scratch in terms of design, but this was never required in this project.

## 6.1. Stealth AI

The following three chapters detail the results from objective 1, Stealth AI. The three objectives were completed back to front. This was because the objective was designed with the behaviour tree in mind. Thus during the Project Definition stage, it was assumed that the best course of action would be to develop a behaviour tree first and then add functionality to that with classes from other objectives. However, before development began, this decision was reversed, and instead, it was decided that the classes should be made first and then implemented as part of the behaviour tree. The benefit of doing this was that if the states were correctly implemented beforehand, the behaviour tree could be tested quickly, as the expected results from the classes within the behaviour tree would be known. However, what was wrongly assumed at this stage was that there would be little or no changes to the classes when they would go from being implemented as part of a stand-alone class to one that had to work as part of a behaviour tree and work together with many classes. Refactoring these classes to fit within the behaviour tree architecture took considerable development time.

### 6.1.1. Guard Detection (Objective 1C)

The Guard Detection objective created outputs used by the AI to spot the player slowly over time.

The requirements of this objective entailed a Guard not having a binary detection of the player but instead slowly detecting the player over a few seconds. The design for the solution to this problem was simple and elegant. The Guard would have a view cone, which was split into different sections, and each of these sections would detect the player slower or faster based on how far the player was from the Guard.

The first code output produced by this objective was the algorithm used to detect the player. The algorithm created vision zones within a Guard`s vision cone, and the player would be spotted at different rates depending on the zone they were occupying.

Part of the algorithm related to checking if the player is within close proximity to the player(nearViewingDist) can be seen below. The algorithm can be modified to include more zones by adding more angles in the Guard`s vision cone and adding more viewing distances for the Guard.

*// Checks if there is an obstacle between the guard and player*

*if (!Physics.Linecast(\_transform.position, \_player.position, \_obstacleMask))*

*{*

*// Checks to see the distance between the guard and player*

*if (Vector3.Distance(\_transform.position, \_player.position) < nearViewingDist)*

*{*

*// Checks to see the angle between the guard and player*

*if (playerGuardAngle < angleA / 2f)*

*{*

*\_guard.zone = GuardBehaviourTree.ZoneState.zone1;*

*}*

*else if (playerGuardAngle < angleC / 2f)*

*{*

*\_guard.zone = GuardBehaviourTree.ZoneState.zone2;*

*}*

*else*

*{*

*\_guard.zone = GuardBehaviourTree.ZoneState.emptyZone;*

*}*

*}*

*}*

Another part of this algorithm is that a timer is started once a player is found in a zone. Once the timer ends, the player will be spotted by the Guard. The timers can be modified depending on the zone to create a specific detection timer per zone for any future implementation of this algorithm.

The implementation also has a raycast check. This stops the Guard from being able to see the player through specific layers. An output in Unity was a layer mask related to this called Obstacle Mask. This was placed on any objects, such as buildings or trees, to prevent the Gurad from being able to see through the game object. With the Unity IDE, developers can easily set a new object as part of this Obstacle layer, which will easily prevent the Guard from being able to see the player through that object. Within the Guard Behaviour Tree script for the Guard prefab, there is an option to change the Obstacle layer mask to any other layer mask in the game. This is an efficient way to stop the Guard from spotting the player through other layers, not just the Obstacle mask.

The use case tests for this implementation passed. The Guard could successfully detect the player at different rates based on what zone they were in. The use case testing is further detailed in the Appendix.

### 6.1.2. Guard Pathfinding Search (Objective 1B)

The Guard Pathfinding Search objective created outputs to allow a Guard to navigate the game world independently and search for the player semi-realistically.

The requirements for this objective entailed a Guard pathfinding to the player’s last known location, coordinating with their team to pathfind to a location on the map to look for the player and then return to their patrol paths.

The design for this implementation required the Guards to stay patrolling until a Guard spotted the player. This Guard would then communicate to the others about the player’s whereabouts, and they would all converge on that location. If they could not find the player after having spotted them, they would pathfind to random locations on the map, and then after their search had ended, they would return to their patrols. The design was changed during development; the locations to search would be selected randomly, but the actual search locations would be pre-determined by the developer. This allows future developers to have some say in where the Guards search for the player. Also, it can make the AI seem more intelligent as they could be made to look in places the player is likely to hide in instead of random locations.

The first output of this objective is the NavMesh system. This project package was imported, and a tutorial was followed to allow a NavMesh Agent to be created on the level(Brackeys, 2018). The output from this was the ability to add a NavMesh script onto the Guard prefab and set the Guard`s pathfinding parameters. These parameters can be modified to support different types of Enemies in the game. For example, a new NavMesh agent type can be quickly created for a larger Enemy. This Enemy could be slower than a regular Guard and not be able to fit through tight spaces. These parameters can be easily set and adjusted with the NavMesh script.

The NavMesh system requires code to tell a NavMesh agent, in this prototype, this was the Guard, where to go. The second output, therefore, naturally followed on from Objective 1C and told the Guard to set its destination to the player`s location upon spotting the player. This modified the previous algorithm from just spotting the player and was now a Chase class. Of course, when the Guard lost sight of the player at this stage, they would not go anywhere. The following output changed this.

The BTGuardGroup script encapsulates all of the Guards in a group. At this stage, it was only responsible for checking if one of the Guards had spotted the player and then telling the other Guard to go to the player’s location. This was used as a rudimentary attempt at communication between the Guards, making the AI look more intelligent. However, once all of the Guards had lost sight of the player, the BTGuardGroup script would then organise a search for the player. It did this by randomly assigning the Guard to go to a specific location on the map and stay there for a few seconds before checking another location and returning. This could be manipulated easily to fit any other search pattern. The search locations can be stored as game objects in the scene, and the position of these objects can be used to set the search locations. This allows easy changes to the search locations to adapt to new environments. The search locations for a Guard group in the prototype can be seen in Figure 3.



Figure 3

*Scene view of prototype. Small blue dots on the ground show the search locations for the two Guards in this area.*

The final output of the Guard Pathfinding Search answered what the Guards would do when they are not Chasing or Searching for the player. The answer for this was Patrolling. The same NavMesh system works with game objects in the scene to create a Patrol Path for the Guard. The Guard prefab comes with a child game object called Patrol Paths, which then has three empty child objects that can be used to store the Patrol Path. The Guard Behaviour Tree script under the Guard prefab has built-in components to allow developers to select Patrol Points for a particular Guard. The developer simply has to drag and drop the Parol Path game objects into the Patrol Points component to create a Guard`s Patrol Path.

During testing, the expected results of the test were observed. However, if there had been more time to develop this project, the requirements and design could have been improved to accommodate a better search pattern. Currently, the Guards stand stationary when they are at a search location. A better implementation would have a more realistic interpretation of what the Guards would do when they have reached their search location, such as moving around the search location or maybe turning a few degrees to make it look like they are looking over their shoulder to look for the player. For this project, the current implementation did the job. It created an efficient pathfinding system and a searching pattern with Guards who look like they are communicating. However, there is room for improvement.

### 6.1.3. Guard AI Behaviour Tree (Objective 1A)

The requirements for the behaviour tree involve having four main states that can control the Guard`s behaviour. These would primarily be based on the previously implemented classes from objectives 1C and 1B.

The design of the behaviour tree proved to be very complicated. This was due to two main reasons. The first was detailed at the start of the Results chapter. Due to the order of operations for this objective, the classes previously developed for the AI had to be heavily adapted to accommodate the Guard Behaviour Tree. The second reason was due to the implementation of the BTGuardGroup class having to have access to the Guard components such as its NavMesh Agent; often, the behaviour tree and the BTGuardGroup class would clash, and both try to do different things at the same time. To solve this, four new classes were added, Check Attack, Group Attack, Check Search and Group Search. These classes were essentially used to stop the behaviour tree from doing anything when the BTGuardGroup class coordinated a search or an attack.

The architecture for the Behaviour Tree was re-used from a tutorial(Mina Pêcheux, 2021). The architecture included four node types, as detailed in the Method chapter. Therefore, the first output for this objective is the ability for any developer to quickly and easily modify and change the structure of the Behaviour Tree to suit their requirements. They can efficiently add or remove states and design a new Behaviour Tree with their classes with the architecture provided.

The Guard AI Behaviour Tree implements an efficient method of selecting the correct state for a Guard to be in based on their current situation in the game. In this implementation, four main states were created. These include Patrolling, Chasing, Searching and Attacking. The first three were implemented as part of the last two objectives and modified to fit within the architecture of the Guard Behaviour Tree. On the other hand, the Attacking state had to be developed by splitting the Chasing state into two parts. This was done by changing the Chasing state to only chase the player if they were in zones 2-5. If the player were in zone 1, the Attacking state would instead be run. The four different states are one of the outputs of this objective. However, the Attacking state is currently just a placeholder. The actual attacking functionality from the Guards was not in this project’s scope as it was deemed that First-Person games already do this exceptionally well. A Guard can correctly execute the Attacking state, but in its current form, the Guard performs no attacks when in the state.

Alongside the placeholder attacking functionality, the BTGuardGroup class was modified. This was done so that if one Guard attacked the player, all other Guards would be sent to the player’s location. This was done to mimic how a team of operatives would communicate in real life and converge on an opponent. The BTGuardGroup class is another output in the prototype. By assigning the script to a game object, a developer can assign Guards to it. This will, in turn, ensure that these Guards work together in the game.

The Guard Behaviour Tree script not only housed functionality for when a Guard should be in a specific state, but it also had additional parameters that could be used to modify the Guard`s behaviour slightly or to debug the Guard`s current state. For example, a developer could change the Obstacle mask the Guard has to a different layer in the scene. The script has many boolean values that the developer can use to check specific parameters and understand what the guard is currently doing. The specific script for the Behaviour tree is the final output for this objective.

Despite the long-winded development of the behaviour tree, the use case testing passed without a hitch.

## 6.2. Stealth UI

The first output from this objective was a billboard script. This was re-used from an online tutorial (Brackeys, 2020). The billboard script would orient the UI elements above the head of the Guard to always face the player. The billboard script could be placed onto any Canvas (an object which holds UI elements).

### 6.2.1. UI Depicting Guard State (Objective 2A)

The requirements for this objective involved multiple UI elements above the Guard`s heads depicting what state they were currently in. Three of the four states within the Guard Behaviour Tree had UI elements attached to them. The search state had one UI element attached to it, and the Chasing and Attacking states were combined to have an Alerted UI element attached to them.

The design for the UI elements depicting the Guard`s state was simple. It required each UI element to be displayed or hidden based on the current state of the Guard. Since this is a prototype, the visual for the UI elements did not have to be magnificent, they just had to be clear in what they represented, which was achieved.

The first output for this objective was the scripts to load and unload the sprites. The Guard prefab has a child object called a Canvas. The Canvas object is used to implement 2d sprites into a scene. By binding it to the Guard and utilising the afore-mentioned billboard script, the Canvas object would float above the Guard`s head and always face the player. The sprites were stored as Image components under this Canvas game object. To load and unload them, a script was implemented for each sprite. The script had the functionality to enable and disable the Image component. The Guard prefab already had the Searching and Alerted sprite stored in the Guard Behaviour Tree script and, therefore, could enable and disable them by simply calling one of two functions, “display” or “disable”. This is a straightforward output, but it is easy to reproduce for any new sprites added to the game.

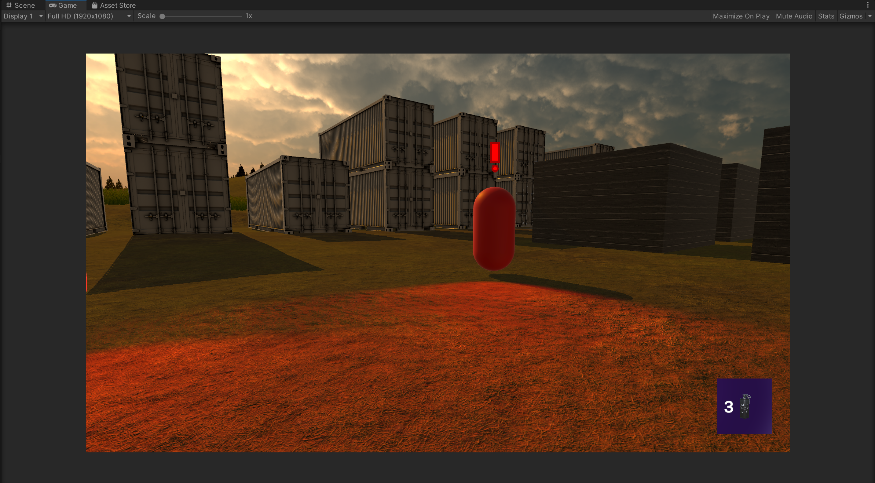


Figure 4

*Screenshot of game view showing a Guard`s Alerted UI element.*

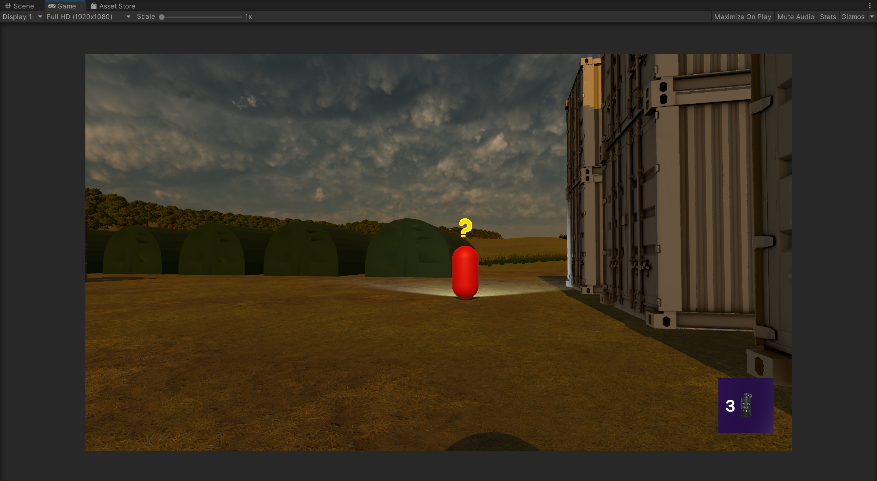


Figure 5

*Screenshot of game view showing a Guard`s Searching UI element.*

The second output for this objective was the actual sprites. Two sprites were designed in the free online sprite editor, Piskel (Descottes, 2019). There was one for the Searching sprite, resembling a question mark symbol and one for the Alerted sprite resembling an exclamation mark. These can be seen in Figure 4 and Figure 5, respectively.

### 6.2.2. UI Depicting Guard Detection (Objective 2B)

This objective required implementing a detection bar UI element that would tell the player how close a Guard was to spotting them.

The design for this objective was split into two parts. The first part entailed displaying the detection bar at the desired time. This was done by enabling the bar when the player was in the process of getting spotted and when the player was actually spotted. The second part of the design involved how to depict the detection level. How this could be done in Unity during the design stage was unclear. However, the general principle for how to work out the detection level was easy to design. It would be based on the time the player had currently been visible for over the time it would take to fully spot the player.

Similarly to objective 2A, this objective required the design of a class that would load and unload the Detection Bar UI element. In this case, it was done when the Guard could see the player, but the player had not been visible long enough for the Guard to have spotted them. The Detection Bar script re-used functionality from an online tutorial to smoothly increase or decrease the bar(Brackeys, 2020). This was done by having a blank image scaled using a Unity Slider component. The Detection Bar also had a Gradient component. This allowed the blank image to be coloured differently based on its size (a larger image would mean the player was closer to being detected). The Slider and Gradient components and the associated code were also re-used from the online tutorial(Brackeys, 2020).

The Guard Behaviour Tree script sets the Detection Bar level for the Guard. It sets the level of detection based on the timer used to spot the player. For example, if the player had been seen for one second and the Guard would fully spot the player after five seconds, then the Detection Bar level would be at 1/5th capacity. This was done by passing a value to the Detection Bar script from the Guard Behaviour Tree script based on the detection level.

The final output of this objective was the sprites associated with the Detection Bar. There were three sprites used. The first was the blank Image component, scaled up and down to show the detection level. The second was a border for this Image component, designed in Piskel(Descottes, 2019). The last sprite was a re-used asset found online(creazilla.com, n.d.). It was used to give the player some indication of what the bar was showing. The visual for the Detection Bar can be seen in Figure 6.

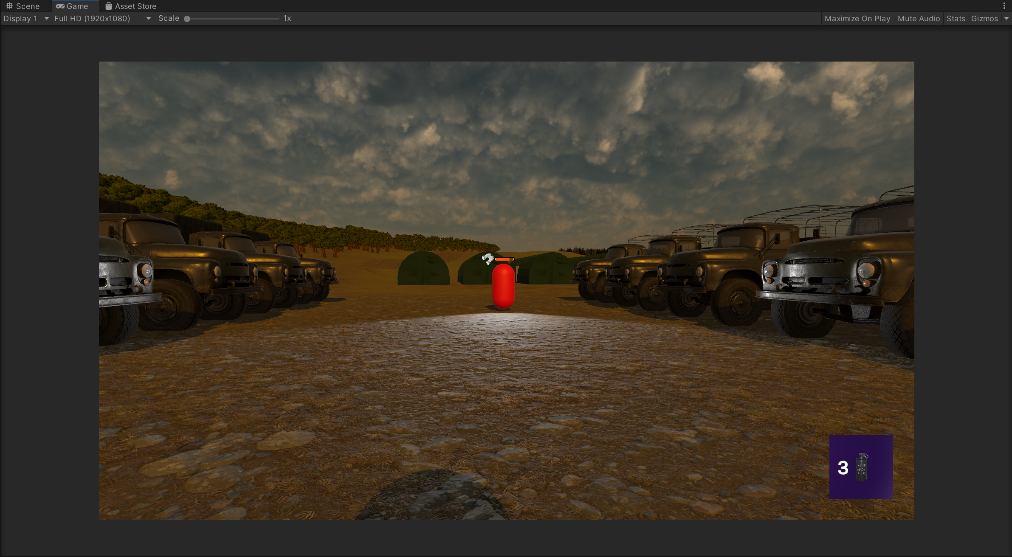


Figure 6

*Screenshot showing the Guard`s Detection Bar UI element.*

During the use case testing of this feature, two problems occurred. The first was fixed relatively quickly. It was to do with when the detection bar would be enabled and disabled. The test found that the detection bar would disappear when the player was out of the Guard`s line of sight. This was because the design of the detection bar stated that the UI element must only be displayed when the player is in the process of being spotted. Since the Guard is not currently spotting the player, the detection bar will disappear when the player moves out of the Guard`s line of sight. A quick fix was implemented, ensuring that the detection bar would only be disabled if the detection level was at 0. This fixed this issue.

The second issue that arose after this one had been fixed was that the detection bar would jump down when the player left the Guard`s vision zone. This occurred because of the different zone timers in the Guard’s vision cone and how the detection bar lowered after the player left the Guard`s vision cone. The detection level is set relative to every zones timer. For example, it was set to five seconds for zone five and one second for zone one. If the player jumped between zones or jumped out of a zone entirely, then the timer needed to fully spot the player would change. This led to the detection level jumping up and down based on where the player moved to in the detection zone. To explain this clearly, imagine if the player was in zone 1 and had 1 second before being spotted. If the player was halfway to being spotted, their visible timer would be 0.5 seconds, and the Guards bar would be half full. If the player instantly teleported to zone 5, where the timer is 5 seconds, then the time the player has been spotted for has not changed, it is still 0.5 seconds, but now crucially, the time for the player to be spotted has gone up. So now the bar would not be half full, but a tenth full. These kinds of situations were causing the jumping in the detection bar.

An elegant redesign for this solution may involve storing the current detection level based on the percentage of the bar that has been filled and not solely on the timer. That way, if the player changes zones, the percentage would remain the same, and the timer can decrease steadily.

## 6.3. Stealth Mechanics

### 6.3.1 Two Stealth Mechanics (Objective 3A)

Objective 3A required the development of two mechanics that would aid the player in playing stealthily.

The smoke bomb mechanic specifically required the smoke bomb effect not to be seen through by Guards and that if the Guards were caught inside the smoke bomb radius, they would lose sight of the player. Lastly, if a Guard spots the smoke but does not spot the player, they will communicate with the other Guards in their group and convene at the location of the smoke and then enter a search pattern to look for the player.

The design for this implementation seemed simple at first. The design involved having a smoke bomb canister drop and then be culled from the game world while simultaneously the smoke effect would be instantiated. The smoke effect would have the obstacle layer mask to prevent Guards from being able to see through it. Then a simple collision check will be used whenever a smoke effect has been instantiated to check whether a Gaurd was within the smoke effect. If they were, their movement would be disabled temporarily. Lastly, to get the Guard to spot the smoke, the Guard Behaviour Tree would have another class implemented into it, returning success if the smoke was within the Guard`s vision cone. This would, in turn, set a boolean variable for the Guard to true, and the BTGuardGroup script would read this variable to be true and then set all of the other Guards in the group to converge to this location.

The smoke bomb mechanic was implemented relatively unsuccessfully. The drop smoke script comes as a component on the Player prefab. This script handles the deployment of a smoke bomb canister. The mesh of the canister was found online (assetstore.unity.com, n.d.). The script also has built-in parameters that allow a developer to alter things such as the throwing force of the canister, the exact mesh thrown or the angle it is thrown. This script was re-used from an online tutorial(Brackeys, 2017). After a canister was instantiated, another script would begin running. The smoke bomb script was attached to the smoke bomb prefab. Upon deployment of the canister and the running of the smoke bomb script, there would be a short delay before the canister was culled. For example, this delay is adjustable by future developers who may want a longer fuse time. After the canister was culled, the smoke particle effect would be instantiated. Again, this effect can be altered in whatever way a developer sees fit due to its accessibility in the inspector window under the smoke bomb prefab. The smoke bomb script was also re-used from an online tutorial (Brackeys, 2017). The smoke bomb particle effect originated from a Unity Store Asset (assetstore.unity.com, n.d.). However, this effect did not suit the requirements in its original form. The effect had to be modified using the built-in particle system component to make the smoke look more realistic. The change from the original smoke effect to the modified one can be seen in Figure 7. The smoke effect particle prefab had a box collider and a layer mask to make the Guards unable to see through the smoke. The Guards could not see through the smoke by applying the Obstacle layer mask, providing a realistic interpretation of a smoke bomb. A final script was also created. This one was placed directly on the modified smoke particle effect prefab. It was called destroy smoke and culled the game object from the world after the smoke effect had run out. These can be categorised into five main outputs at this stage. The first three are the scripts, including the smoke bomb script, the drop smoke script and the destroy smoke script. The other two outputs are the smoke bomb canister prefab and the smoke particle effect prefab.

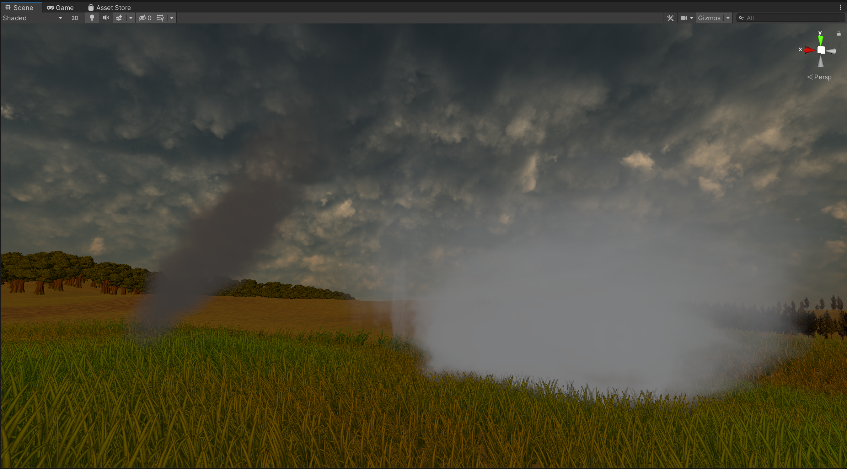


Figure 7

*The screenshot shows the original smoke effect on the left and the modified smoke effect on the right.*

Use case testing for this mechanic failed, and there should have been more outputs from this mechanic. The smoke bomb requirements detailed that the smoke bomb should be able to be spotted by a Guard who would then go over to investigate the smoke. They should have also communicated with the other Guards in the group to tell them to go over to where the smoke was spotted. Lastly, if a Guard were caught inside the smoke bomb, they would stop moving until the smoke had cleared, which was meant to represent the Guard's confusion and increase the mechanic's power. These features could not be implemented. Due to a lack of understanding of Unity, it was not possible within the time frame to make the Guards able to spot the player and the smoke. The attempted solution involved modifying how the Guard Detection algorithm worked to use it to allow a Guard to spot the smoke, but this was not possible.

The second mechanic to implement was a hiding mechanic. The requirements for this objective entailed allowing the player to take cover within a game object, such as a bush or tall grass. In this prototype, this was in the form of some bushes which were found online (assetstore.unity.com, n.d.). Due to the implementation in Objective 1C relating to creating a layer mask called an Obstacle mask, the implementation for this mechanic was simple. By placing a box collider around a mesh with the IsTrigger component selected and applying the obstacle mask to the object, any mesh could be used as a hiding place for the player. This mechanic in its design is only limited by the developer’s creativity. The obstacle mask has been previously discussed as an output. The other output specific to this objective is the creation of three bush prefabs which can be dragged and dropped into the map to provide an easy hiding spot. They come with the box collider and Obstacle mask, as detailed. The use case testing for this objective passed.

### 6.3.2. Limiting the use of Stealth Mechanics (Objective 3B)

Objective 3b required that the player could not overly rely on the game’s mechanics.



Figure 8

*The screenshot shows the HUD graphic for the smoke bomb counter.*

The design for limiting the effectiveness of the smoke bomb mechanic was simple. The smoke bomb would have to be limited in its number of uses and how much time the player would have to wait between deploying smoke bombs so that they could not drop multiple smoke bombs repeatedly in quick succession.

The drop smoke script was modified to include a parameter for the developer to define how many smoke bombs the player should start with. This would then be decremented as the player used them, and once the player had run out, they could not drop any more smoke bombs. Another output, a simple HUD, accompanied this. The HUD was only in place to inform the player how many smoke bombs they had left. Again the drop smoke script was modified to access the number of remaining smoke bombs and print that number onto the screen. An online tutorial was used to implement this (GameDevTraum in English, 2022). Alongside this counter, a small graphic was designed to show the player exactly what this number meant at the corner of the screen. The graphic also re-uses the smoke bomb canister asset (assetstore.unity.com, n.d.). The full graphic can be seen in Figure 8. The HUD graphic is contained within the Player prefab.

The smoke bomb mechanic was also limited in another way. The drop smoke script was modified again to stop the player from dropping the smoke bombs in quick succession. This was done by a timer which would set the cooldown time between each deployment of a smoke bomb.

The limitation of the hiding mechanic was not easy to design. The outputs from the hiding mechanic offer nothing to limit the uses or prevent the mechanic from being used in quick succession. However, due to the nature of the outputs from the hiding mechanic and how easy they are to drag and drop into the level, the developer can easily manipulate where the player can hide. Thus, they can regulate the effectiveness of the mechanic by how frequently the hiding objects are placed in a level.

Both mechanics passed their use case tests.

## 6.4. Traversal (Objective 4A)

The project's Traversal component requires the player to have a unique and dynamic method of moving through the level. In the prototype, a zipline was chosen as the solution.

The design of the zipline entailed having two points between which the player would travel. At this stage it was assumed that a simple cahnge to the players tranfomr component over time would be the simplest solution but upon furtehr reserch in the Unity and C# documenatation, a simpler solution using linear interpolation was found.

The first output from this objective was the zipline script. This was stored under the player prefab. The script implemented linear interpolation to move the player down the zipline. It also disabled the player’s move script and their gravity to ensure a smooth ride down before re-enabling these when the player dismounted at the bottom of the zipline. The script could be easily modified with the parameters in the inspector window to increase or decrease the time the player spends on the zipline.

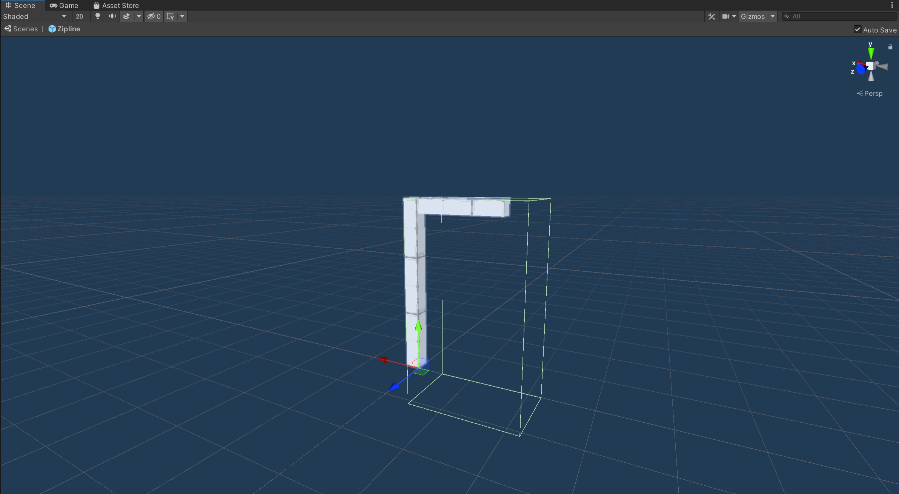


Figure 9

*A screenshot showing the zipline hook point prefab, built using Unity ProBuilder.*

A zipline prefab was also created as an output for this objective. The zipline prefab had the zipline script built in. Developers could use this to set the zipline's starting location dn ending location. This was vital as it prevented the zipline from being used backwards. Of course, the option is there for the developers to change this and have the zipline be usable in both directions if their implementation requires it.

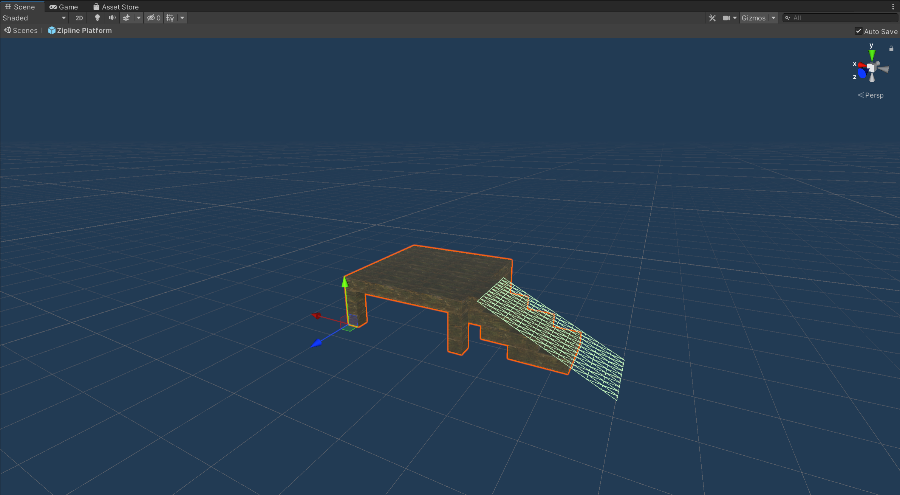


Figure 10

*A screenshot showing the zipline platform prefab, built using Unity ProBuilder.*

The zipline prefab also has a simple zipline hook point object attached to it. This was designed from scratch in Unity using the built-in ProBuilder tool, as shown in Figure 9. Also, a simple zipline platform was designed from scratch again in ProBuilder, which can be seen in Figure 10. The two components combined could be used to have a comprehensive zipline area in-game, as seen in Figure 11.

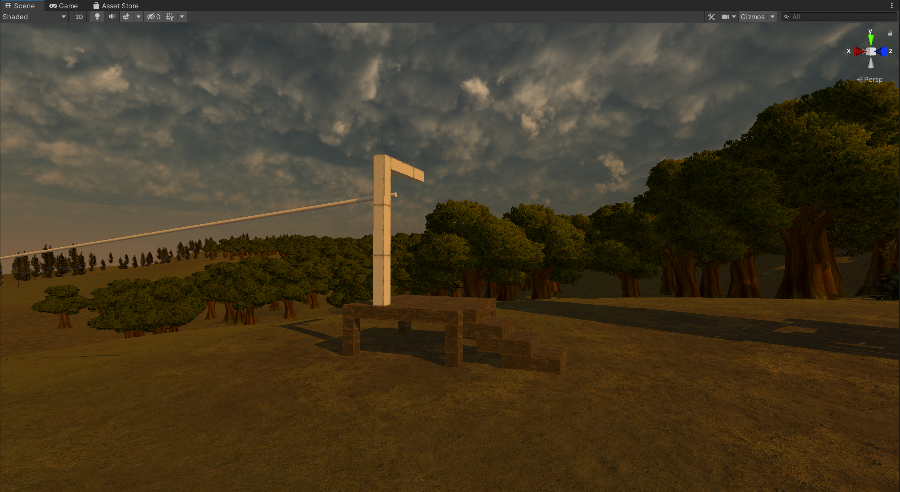


Figure 11

*A screenshot showing the zipline hook point prefab and the zipline platform prefab together in the scene.*

Use case did pass for this feature, however it was determined that the feature in general neede to be more polished. Further details can be found in the Appendix.

## 6.5. Additional Results