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](#_Toc137269343)

[2. Introduction 5](#_Toc137269344)

[2.1. Description of the Problem 5](#_Toc137269345)

[2.2. Project Objectives 5](#_Toc137269346)

[2.2.1. Stealth AI 5](#_Toc137269347)

[2.2.2. Stealth UI 5](#_Toc137269348)

[2.2.3. Stealth Mechanics 6](#_Toc137269349)

[2.2.4. Movement/Traversal 6](#_Toc137269350)

[2.2.5. Balance 6](#_Toc137269351)

[2.2.6. Other Functionality 6](#_Toc137269352)

[2.3. Project Beneficiaries 6](#_Toc137269353)

[2.4. Work Performed 6](#_Toc137269354)

[2.5. Limited Scope 7](#_Toc137269355)

[3. Output Summary 8](#_Toc137269356)

[3.1. Guard Behaviour Tree 8](#_Toc137269357)

[3.2. Group Searching and Attacking Algorithm 8](#_Toc137269358)

[3.3. Detection System 8](#_Toc137269359)

[3.4. State Depiction System 9](#_Toc137269360)

[3.5. Hiding Mechanics 9](#_Toc137269361)

[3.6. Traversal 9](#_Toc137269362)

[4. Literature Review 10](#_Toc137269363)

[4.1. Introduction 10](#_Toc137269364)

[4.2. Artificial Intelligence 10](#_Toc137269365)

[4.3. User Interface 10](#_Toc137269366)

[4.4. Mechanics 11](#_Toc137269367)

[4.5. Traversal 11](#_Toc137269368)

[4.6. Balance 11](#_Toc137269369)

[4.7. Conclusion 11](#_Toc137269370)

[5. Method 12](#_Toc137269371)

[5.1. Stealth AI 13](#_Toc137269372)

[5.1.1. Guard Detection (Objective 1C) 13](#_Toc137269373)

[5.1.2. Guard Pathfinding Search (Objective 1B) 14](#_Toc137269374)

[5.1.3. Guard AI Behaviour Tree (Objective 1A) 14](#_Toc137269375)

[5.2. Stealth UI 15](#_Toc137269376)

[5.2.1. UI Depicting Enemy State (Objective 2A) 15](#_Toc137269377)

[5.2.2. UI Depicting Enemy Detection (Objective 2B) 15](#_Toc137269378)

[5.3. Stealth Mechanics 16](#_Toc137269379)

[5.3.1. Two Stealth Mechanics (Objective 3A) 16](#_Toc137269380)

[5.3.2. Limitng the Use of the Stealth Mechanics (Objective 3B) 16](#_Toc137269381)

[5.4. Traversal 17](#_Toc137269382)

# 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. Due to the strict timeframe for this project, the last two objectives could not be completed, so the build plan was split into four major sections.

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. This allowed me to be flexible and still complete the project to a high degree.

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.

Design 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. This design, implementation and testing method fit well with the Agile methodology. It allowed the implementation to be reworked if the testing criteria were unmet. It also allowed for the requirements to be redone if the output from the objective was not up to scratch in terms of design.

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, which was believed to lead to the most efficient development.

### 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 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.

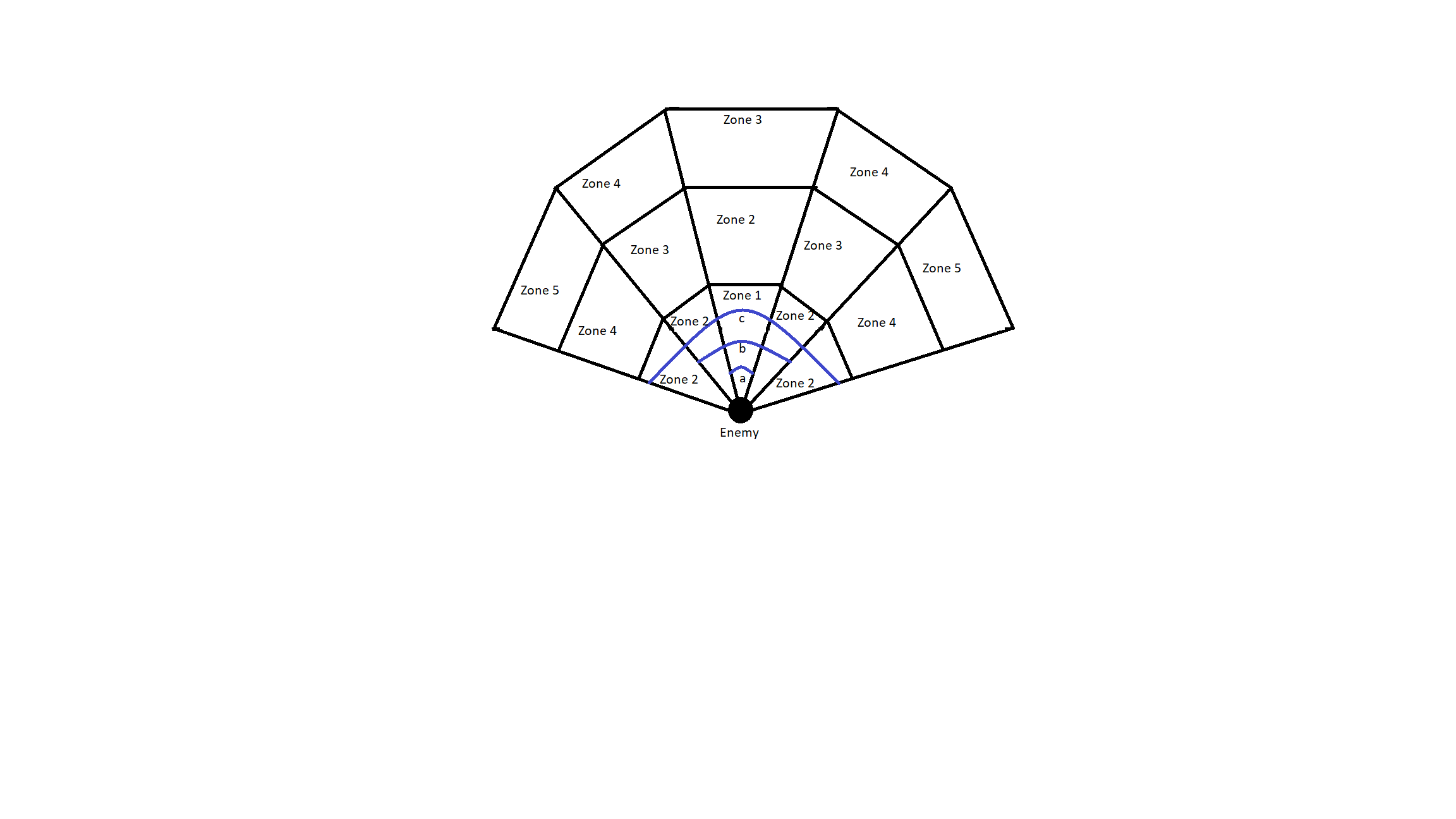


Figure 1

*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 the 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.

INSERT PSEUDO CODE IF LOW ON WORDS

### 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.

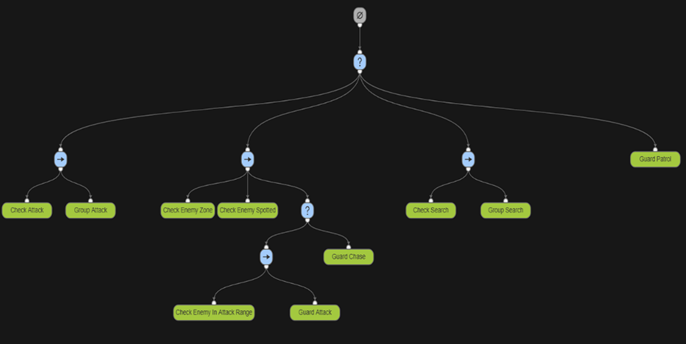


Figure 2

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 Enemy 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 Enemy 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. 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

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.