# 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. It was hoped that doing objectives 1C and 1B first would help create classes that could seamlessly fit into the behaviour tree in objective 1A.

### 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.

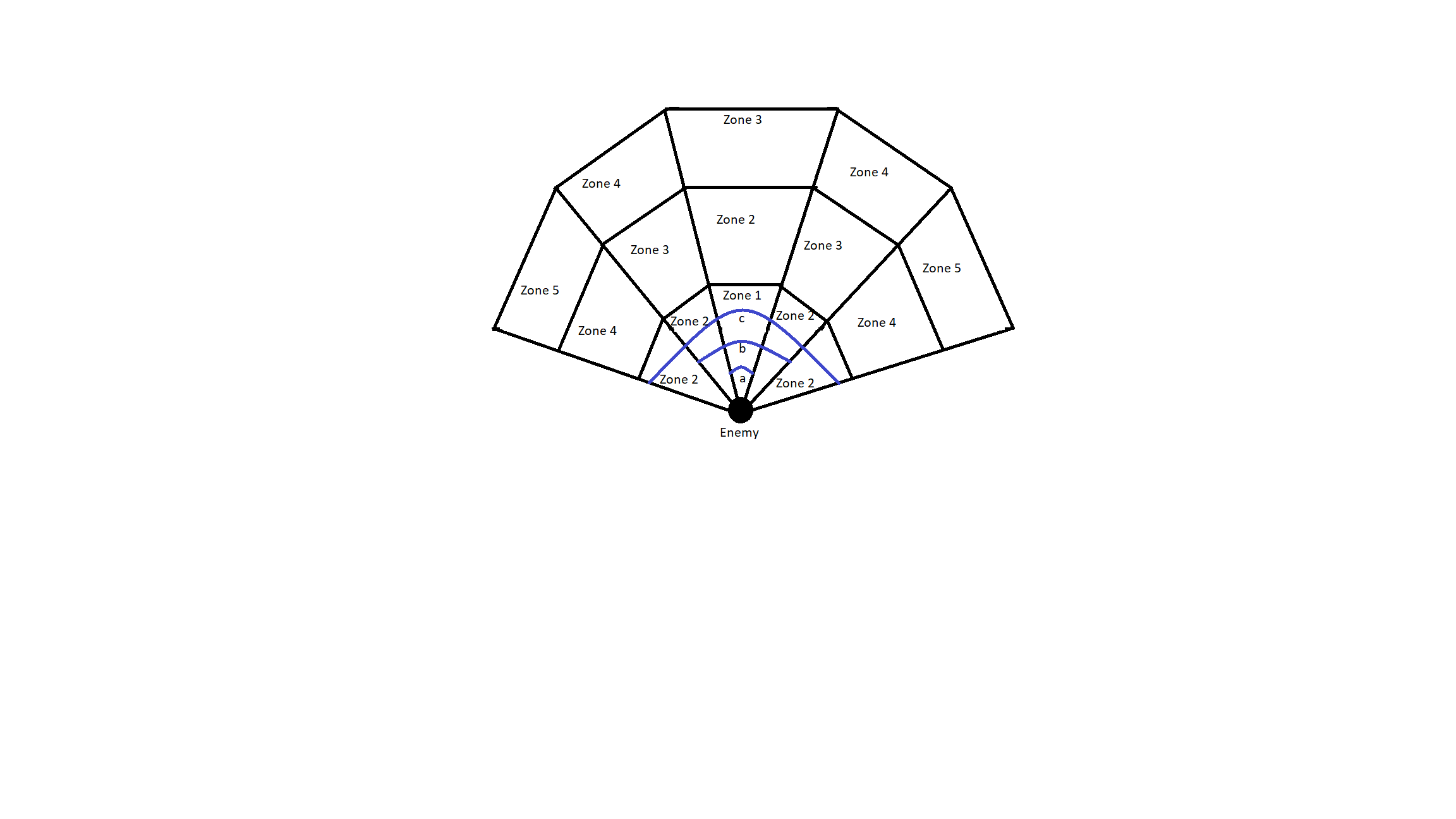


Figure 1

*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.

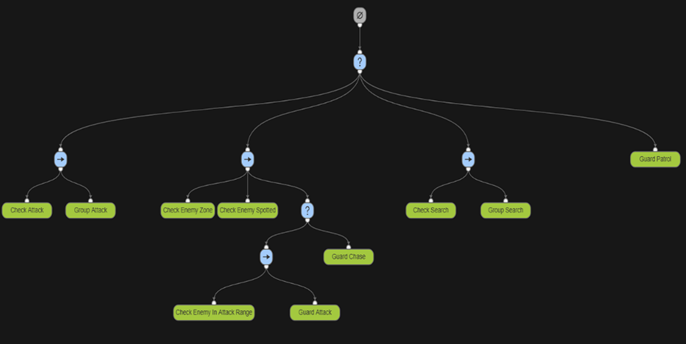


Figure 2

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.