START LOOP

Requirements

Design

Implementation

1. What algorithms/approach did I use
2. Programming lang/platform, why? And what effect did it have?
3. How did I plan and design
4. Stages of work?
5. Version control?
6. Code optimisations? Why or why not?
7. How did I test
8. How did I select inputs to test

Evaluation

Testing

END LOOP

NO RESULTS

State the tools used for each step (inc tutorials)

2500 words

Intro to chapter. Methodology used 1. Tools used 2. Plan and design 3. Build plan 4. Work plan 4. Version control 5. Code optimisations? Test and testing inputs? 500 words ish

Structure sub-chapters based on objective. Requirements. Design. Implementation. Evaluation. Testing. 200 WORDS MAX PER OBJECTIVE

Diagrams were drawn graphically preferably

This project was developed using an agile methodology. Specifically, each major objective was run in an iterative loop of design, implementation and testing. This ensured each objective was completed to a good quality before moving on. This lines up with the build plan.

The build plan was split into 6 major sections and was structured so that the most vital parts of the project were completed first. This ensured excellent quality of the most important 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, there was also a risk of not meeting all the objectives initially set out by the proposal. The build plan also works to help nullify this risk by placing the least important aspects of the prototype towards the end of the build.

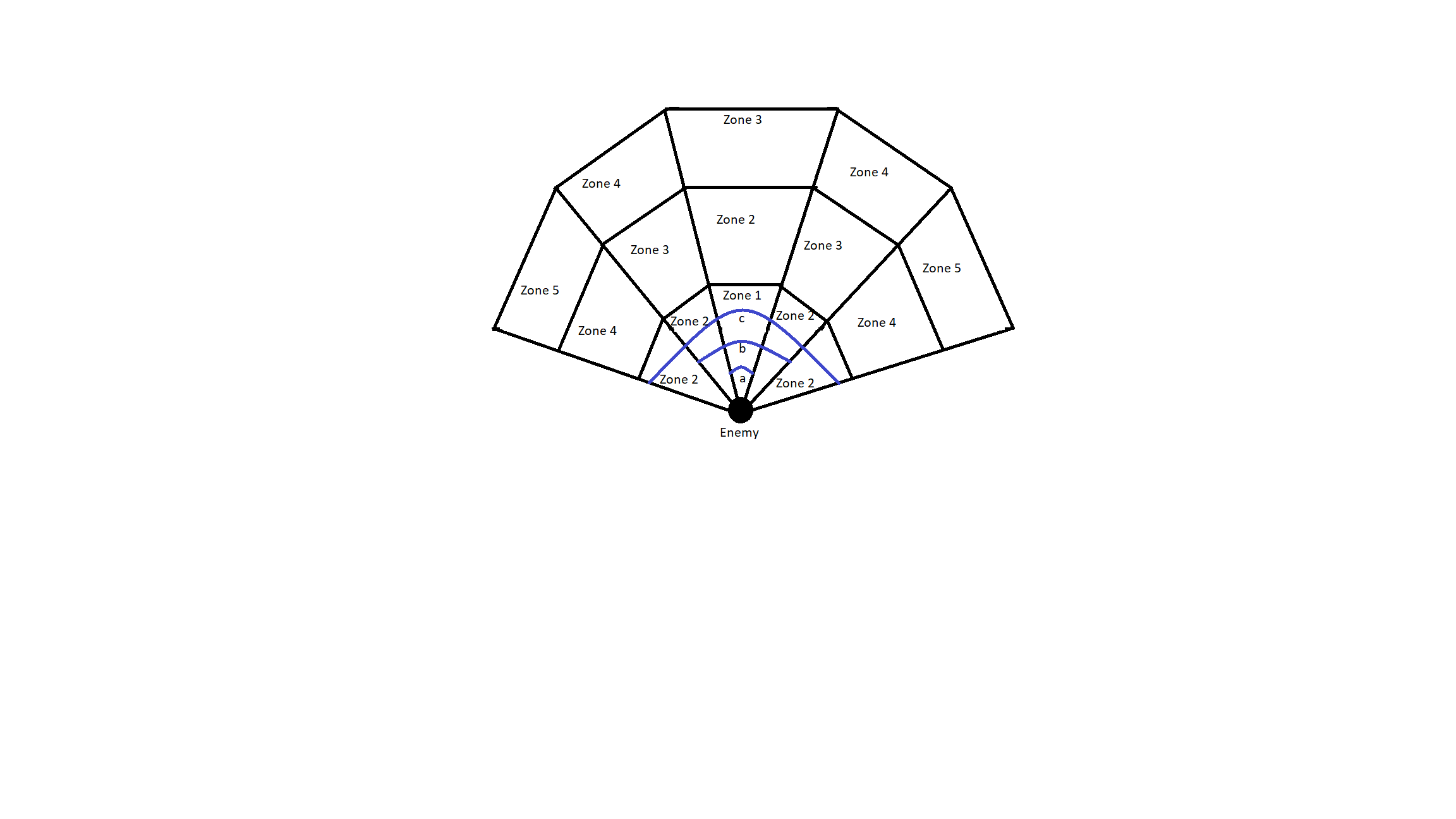
To track the progress of the project I designed a work plan in the form of a Gantt chart and this was updated regularly to reflect the progress made on the project. Coinciding with the agile methodology, there was a reflection every weekend to ensure I was on target with the project and if I wasn`t, 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 broken up into 10 different parts, one for each objective. These sub-chapters include at least the design and implementation required to complete that objective. They may also include the requirements, evaluation and testing of the objective/implementation.

Note, this project`s objectives are 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 will be GitHub. The project was backed up to GitHub at the end of every day in which any sort of work took place on the project. The reason GitHub was chosen was that I have used it a lot in the past to back up my other university work and found it to be quick and easy to use.

Enemy Detection (Objective 1C)



*An example of an Enemy Vision Cone. 15 vision zones, split into 5 zone types. The higher the zone type, the slower the detection. 'a', 'b' and 'c' are the close, medium and wide angles respectively, displayed with the blue lines.*

IN Figure 1

The first objective was to implement the enemy`s ability to slowly detect the player over time. This was done by first implementing some basic detection. This was achieved by performing a raycast from the enemy to the player and checking that the player was within a certain view angle relative to the enemy`s forward vector(ensuring the player was in front of the enemy). The next step was to check that the player was within a certain range of the enemy and that there were no obstacles in the way. To implement a more in-depth system, the enemy would have to slowly detect the player over time. To implement this, split the enemy`s view cone into different zones. This was done by first splitting the view cone into 3 different angles, wide, medium and close. The smaller the angle, the faster the player should be detected. The next step is to add 3 distinct ranges within the view cone. This allows for the player to be detected slower when they are further out from the enemy, with 15 distinct zones categorised into 5 zone types within the enemy`s view cone, all that is left to do is set a timer for each zone type to detect the player at a different speed depending on what zone the player occupies in the cone. 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 INSERT TITLE

Enemy Pathfinding Search (Objective 1B)

Objective 1B entailed the enemy being able to independently path find around the map to search for the player. The first component of this was to have the enemy follow a fixed patrol path around the map, if the player has been spotted by one of the guards but cannot be currently seen, the guards will go into a search state. In unity, one can bake a Nav Mesh and create a Nav Mesh Agent that can path find around a level. This Nav Mesh system implements an A\* pathfinding algorithm to find the shortest possible 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 class is used, containing the guards to coordinate their search. The search class randomly assigns each guard to a location on the map for them to search. The search is controlled by a timer, each guard has 2 search locations for 15 seconds each. After 30 seconds the search class calls an end to the search and tells the guards to resume their patrol as normal. In order to make the AI look like they are communicating in the game, once one guard spots a player, the search class tasks all other guards to path find to the player’s last known location.

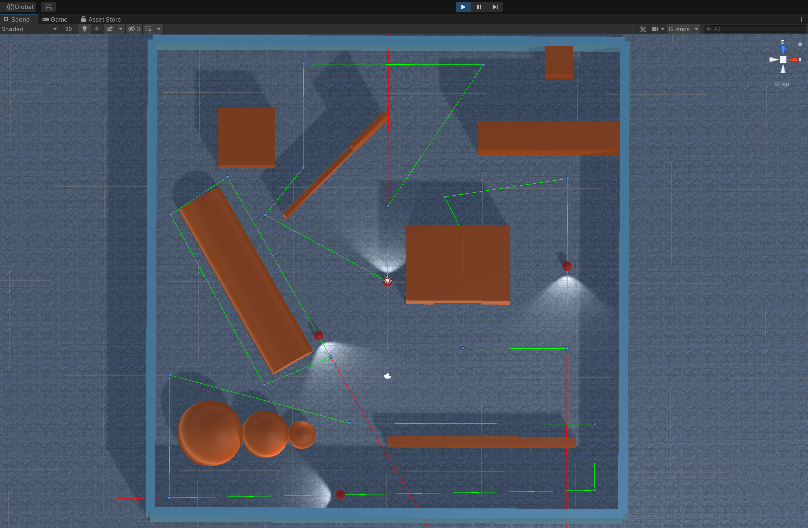
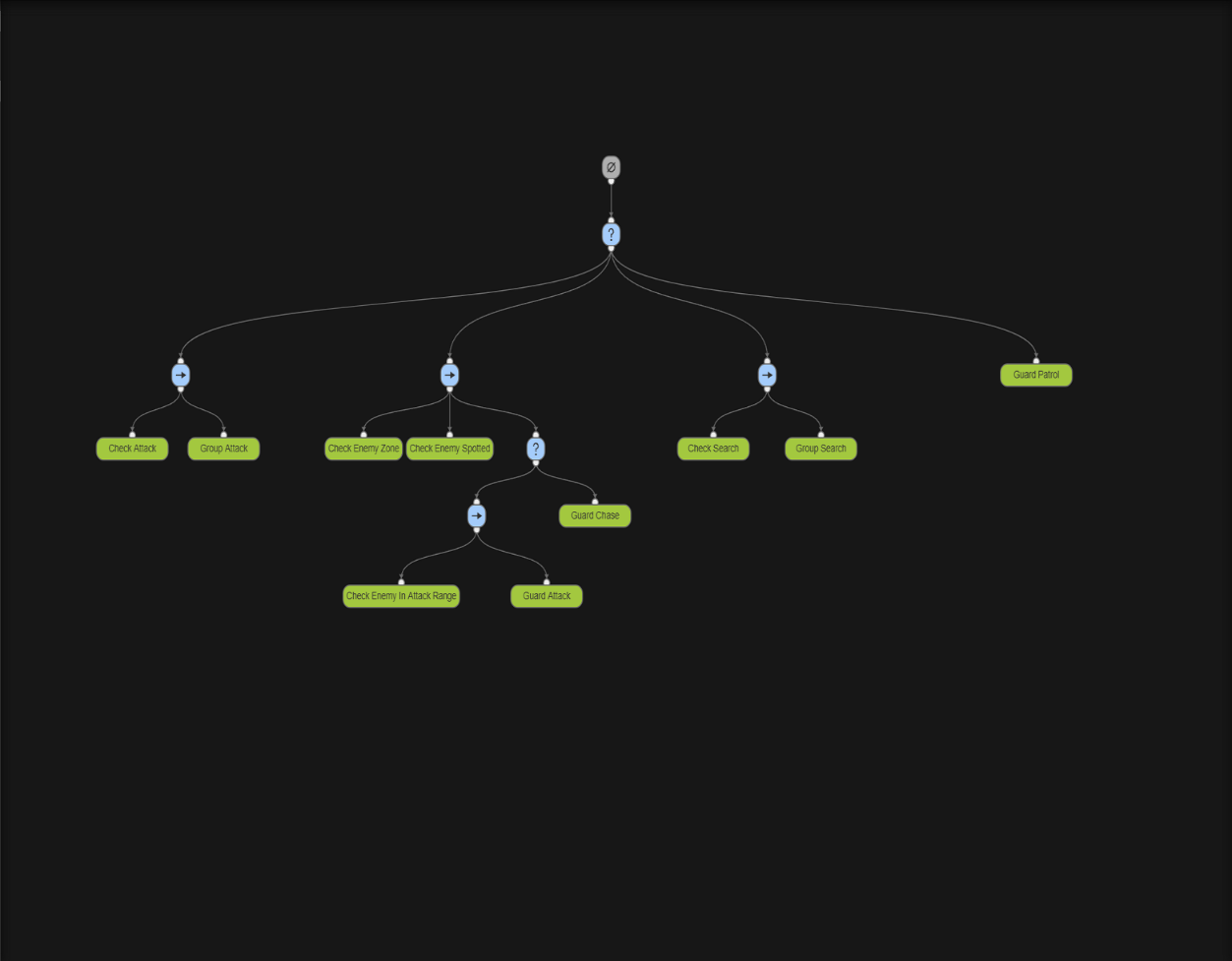


Diagram showing guards on patrol paths indicated by green lines and small blue dots.

Enemy AI Behaviour Tree (Objective 1A)

Objective 1A required the enemies 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. The behaviour tree had 4 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 of the 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 specific behaviour tree used for this project allows for the previously implemented methods from objectives 1C and 1b to work when called upon in the tree to make a coherent AI system.

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.)



UI Depicting Enemy State (Objective 2A)

Objective 2A involved implementing UI elements above the enemies` heads that would tell the player what state the enemy was currently in. The game had four main enemy states: Patrolling, Searching, Chasing and Attacking. The Patrolling state would have no UI element. The Searching state would have a question mark icon and the Chasing and Attacking states both had an exclamation mark UI symbol to depict an ‘Alerted’ enemy. The implementation involved binding these symbols to the enemies` heads and enabling and disabling them based on that enemy`s particular state. The symbols were simple sprites that could be designed by any 2d designing software. The specific software chosen for this project was Piskel(REFERENCE?). The sprites were then imported into Unity as assets and rendered as images under a Canvas UI game object.

UI Depicting Enemy Detection level (Objective 2B)

Objective 2B required the development of a UI element which would help the player understand how close the enemy 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. Then, the detection 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. The detection level was handled by a simple UI Image in Unity. 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.

Smoke Bomb Mechanic (Objective 3Ai)

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. Upon the player’s press of a button, the canister would be initialised into the level falling just in front of the player as if the player had let it go from their hand. The second component was the smoke, the smoke was an effect found on the Unity Asset Store 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 that Enemies could not see the player through the smoke. This implementation worked well but did not fully cover what was required in the Use Case Specification. The Results section will discuss the shortcomings and the reasons for those shortcomings.

Hiding Mechanic (Objective 3Aii)

The second mechanic to implement as part of Objective 3A was a hiding mechanic. This works largely 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. For the implementation at this stage, a bush was used. 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 enemies to walk through the bush. The last thing to add was a layer mask. The layer mask was set to ‘Obstacle’ which was the same for the smoke bomb and for other buildings around the scene. This ensures the enemies cannot see through the bushes and spot the player.

Limiting Use of Smoke Bomb Mechanic (Objective3Bi)

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 the deployment of 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 1 every time a smoke bomb was deployed. Once the counter reached 0, the player would not be allowed to drop any more smoke bombs.

Limiting the Use of the Hiding Mechanic (Objective 3Bii)

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. It is vital that each level has its own interpretation of this implementation to ensure the player cannot simply hide in the hiding areas for the majority of the level.