Results chapter

5400 – 6600 words

600 words per full section

160 per small objective

Introduction to Results chapter:

How was the methodology?

The build plan? Good time estimations? Delays?

For each major objective:

How did I get on?

Better ways of implementing?

What have I learned?

**Stealth AI**

The first major objective was Stealth AI. This was supposed to be completed over the course of two weeks. However, there was a misestimation of the time it would take to set up the basics of the project. This included making a basic level. Implementing a basic player controller and having them walk, run and jump around the map. And having guards who shoot at the enemy and take damage when they are shot at by the player. This caused some significant delays.

*Objective 1C*

The first part of the AI implementation involved having a Guard detecting the player slowly over time and having the detection be slower or faster based on where the player was in the Guard’s vision cone. The implementation was relatively simple. The detection was based on a timer that would start when the Player was within a vision zone. If the vision zone was closer to the Guard, the timer would run quicker. To define the vision zone, a simple algorithm was implemented. The algorithm would first check if there was an object in between the Player and the Guard. If there was not, the algorithm would check how close the Player was to the Guard. If the player was within one of three viewing distances, the algorithm would then check the angle between the Player and the Guard. There were three separate angles defined, one to represent the Guard`s periphery, one to represent a direct eye line (straight ahead) and one to represent the middle ground between the two angles. The combinations of the angle and distance created 15 vision zones that the Player could be in relative to a Guard. An improvement could be made, however. Currently, there are no visual elements/gizmos (the name for these kinds of visual help elements in Unity) to represent the different zones. This would be quite useful for debugging as it allows the developer to quickly see which zone the Player is in compared to which zone the Guard thinks the Player is in.

*Objective 1B*

Objective 1B was fraught with difficulties. Firstly, the objective suffered from some scope creep. The patrol paths were not strictly part of the initial definition of the project but before implementation, they looked quick and easy to implement. Even though they were easy to implement it soon became clear that they kept getting in the way of other parts of the code which were part of the definition, specifically the pathfinding algorithm. A workaround to this problem set the project back by a few days. The pathfinding algorithm itself however worked excellently. Unity uses the A\* algorithm, which is a common pathfinding algorithm. Implementing this was as simple as adding some pre-existing code into a C# script and using it for a game object, specifically a Guard. After implementing this, there was a realisation that the AI did not look very realistic as they would not work together. Thanks to the agile methodology, I slightly tweaked the design to have an overarching BTGuardGroup class which controlled when the Guards would search and when they would attack. The pseudo-code for this class is as follows:

This pseudo-code algorithm is for setting the guards into a search pattern

LOOP through all of the Guards

IF a Guard has seen the Player AND cannot currently see the Player

LOOP through all of the guards

Set all Guards into a Search Pattern

END LOOP

END IF

END LOOP

This pseudo-code algorithm defines the search pattern

Start a Timer

LOOP through all Guards

IF the Timer is between 10 and 15 seconds

Set the Guards to go to a random location on the map

ELSE IF the Timer is between 25 and 40 seconds

Set the Guards to go to another random location on the map

ELSE IF the Timer is above 40 seconds

Set the Guards to go back to patrolling

Set the timer to 0

END IF

END LOOP

This pseudo-code algorithm defines the attack coordination between the Guards

LOOP through all Guards

IF a Guard can currently see the Player

Set all Guards to move to the Player’s location

END IF

END LOOP

*Objective 1A*

The Behaviour Tree architecture was implemented successfully and quickly. However, I soon realised a major error in the layout of the project development when attempting to create a specific Behaviour Tree for the Guard AI. Before starting the project, it seemed wise to develop Objective 1C and 1B first as this would provide the base classes that could then be inserted straight into the Behaviour Tree for the Guard. However, this is not how it worked out. The implementations of the previous two objectives had to be adapted to fit within the Behaviour Tree architecture. This took a considerable portion of time. Added to that, I had to re-perform tests I had previously completed, to ensure that the classes were still working as they should but now within a Behaviour Tree architecture. If the project was to be done again, the start would entail making a simple Behaviour Tree with a single action and then adding to it slowly as more of the project was completed rather than making the classes first and trying to shoehorn the logic into a Behaviour Tree. Regardless of this, the implementation did end up working well.

To understand what the Guard Behaviour Tree does, some knowledge of Behaviour Trees, in general, is required. The Behaviour Tree architecture 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 node has 3 possible states. The first is FAILURE and when this is returned the tree continues traversing. The second is SUCCESS which either continues traversing the tree if it is a child of a Sequence node or simply returns SUCCESS if it is a child of a Selector node. This state will perform the action of the class that returns SUCCESS and the Behaviour Tree will loop from the start again on the next game loop. The last state is RUNNING. This state does not perform an action, but it does stop the tree from traversing further. What this is useful for is for a class which is still performing an action and does not want the tree to skip over it. The tree, therefore, stops traversing and goes back to the beginning and checks this node again to see if it has returned SUCCESS or FAILURE. The Behaviour Tree worked 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 specifically is set up as follows.

The first node after the root node is a Selector node (So the first action to return SUCCESS will be the one committed).

From this node, we go to its leftmost child, a Sequence node(all children must return SUCCESS for an action to be committed). This Sequence node has 2 children which are both Leaf nodes. The left-most child checks to see if the Guard should currently be attacking the player with the other Guards by checking a Boolean value in its definition. If the Guard should be attacking, the node returns SUCCESS.

If the node does return SUCCESS the tree looks at the next child. The Group Attack class simply tells the Guard to go to the Player’s current location and then return a FAILURE. To break this down, first, we know the Guard should be attacking, so we tell it to go to the Player’s location. However, we do not want to return SUCCESS at this point as that will cause the tree to keep telling the Guard to go to the Player`s location. What would be better, is for the guard to only be told once to go to the Player`s location but continue down the tree to allow the Guard to spot the Player when the Player is nearby and perform an attack when in range.

Since that last Sequence node returns a FAILURE to the top Selector node, we move on to the next left-most child. This is also a Sequence node. The left-most child of this Sequence node is the Check Enemy Zone class. This class checks to see if the Player is within any vision zone of the Guard. This class returns SUCCESS regardless of the zone the Player is in and even returns SUCCESS if the Player is not currently in a zone.

The next left-most child is the Check Enemy Spotted class. This class checks which zone the Player is in and either increments the timer if they are in the Guard`s vision zone or decrements the timer if they are outside of the vision zone. If the Player is within the Guard’s vision zone for long enough, the class will set the Guard to go to the Player’s position. This class is also responsible for setting the Guard`s attack Boolean to true or false based on how far the Guard is from the Player. This class will only return SUCCESS if the Player has been in the Guard’s vision zone long enough to be detected. The class will return FAILURE if the Player is not in the Guard`s vision zone and the timer to spot the Player is at zero.

The next left-most node is a Selector node. This Selector node has two children. The first child is a Sequence node which also has two children. The left-most child of this Sequence node is the class called Check Enemy in Attack Range. This class return SUCCESS if the player has been spotted nearby the guard and FAILURE otherwise.

The next child of the same Sequence node is a class called Guard Attack. This class simply gets the Guard to perform an attack on the Player and returns SUCCESS.

If the Guard is not close enough to the Player to attack but the Guard has spotted the Player, the tree goes back to the Selector node after Check Enemy Spotted and traverses down the right child which is the Guard Chase class. This class sets the Guard to go to the Player’s location, imitating the Guard chasing the Player to get within attack range. This class also returns SUCCESS every time it is called.

If none of these actions returns SUCCESS, the third child left-most of the top-most Selector node is then called. This child is another Sequence node. The left-most child of this Sequence node is a class called Check Search. This class queries the Guard`s Booleans to see if this particular Guard should be searching for the Player. This Boolean value is set to true or false in the overarching BTGuardGroup class. All that is important at this stage is to understand that if one of the other Guards starts searching for the player, this Guard will also go searching for the player. Making it seem as if the Guards have communicated and are working together.

If the Check Search class returns SUCCESS, then the next class to be checked is the Group Search class. This class performs no action but only returns SUCCESS. This is because every Guard’s searching pattern is being handled by an aforementioned overarching BTGuardGroup class. All that is required is that the Behaviour Tree does nothing else while this BTGuardGroup class is being executed to coordinate the Guards` search. Therefore, this class returns SUCCESS and performs no actions.

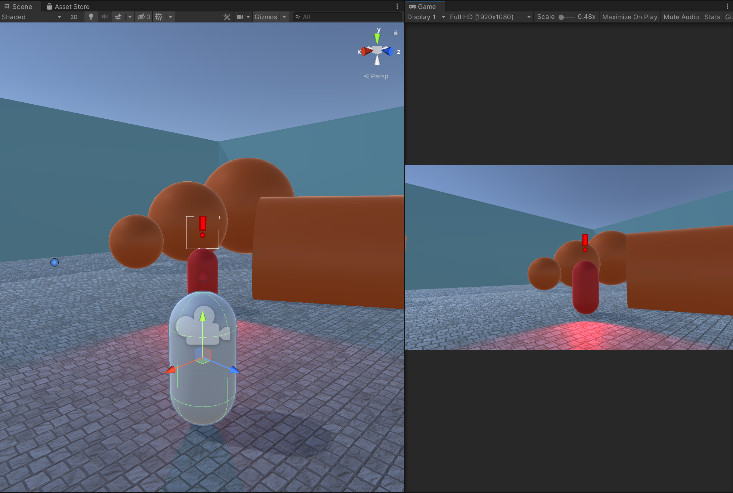
Lastly, if none of the first 3 children of the top-most Selector node returns a successful action, the Guard Patrol class is called. This class firstly ensures that the Guard should not be currently attacking. The reason for this is that in the Group Attack class at the beginning, the class returned failure, even if the Guard should be attacking the Player with all of the other Guards. This was done to allow for the rest of the tree to be run and for the Guards to still be able to perform individual actions such as actually attacking or chasing the Player. However, at this stage, the class ensures that the Guard isn’t coordinating an attack on the Player with the other Guards. And only sets the Guard to his patrolling route if this is the case. The patrolling class itself is relatively simple. It takes in a set of waypoints which are just empty game objects placed around the map. And sets the Guard to go to one of the waypoints and wait for a set amount of time. Then, after waiting, the Guard proceeds to the next waypoint and so on, continuously looping while the class is run. This class always returns RUNNING.

**Stealth UI**

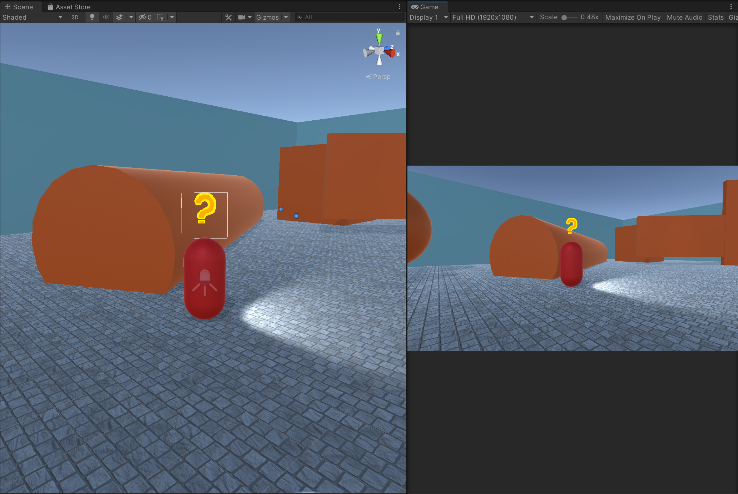
The Stealth UI section of the project went largely without a hitch. The work was completed ahead of schedule and both sub-objectives were completed to a good degree. There could be room to do more in terms of UI in the future. However, for the scope of this project, the current implementation worked fine.

*Objective 2A*

Objective 2A was implemented quickly and easily. The objective entailed having a UI element above the Guard`s head to indicate to the Player, the current state of the guard. There were two UI elements used, both of which were designed in Piskel(REFERENCE) and converted into sprites in Unity. The first was a Question mark sprite. This sprite signalled the Guard being in a ‘Searching’ state. This sprite would appear if the Player had previously been spotted but the player could not currently be seen. It would also appear when the Guards were organising a Search. The benefit of this is that if only one Guard saw the Player, the Player may assume that only that particular Guard would be weary of the Player’s presence, however with the UI, the Player is informed that the Guards are communicating and that all of the Guards have been made aware of the Player.



Screenshot showing Enemy ‘Alerted’ state sprite. (Red Capsule is the Enemy. The left view is the Scene view. The right view is the Game view/Player’s view)

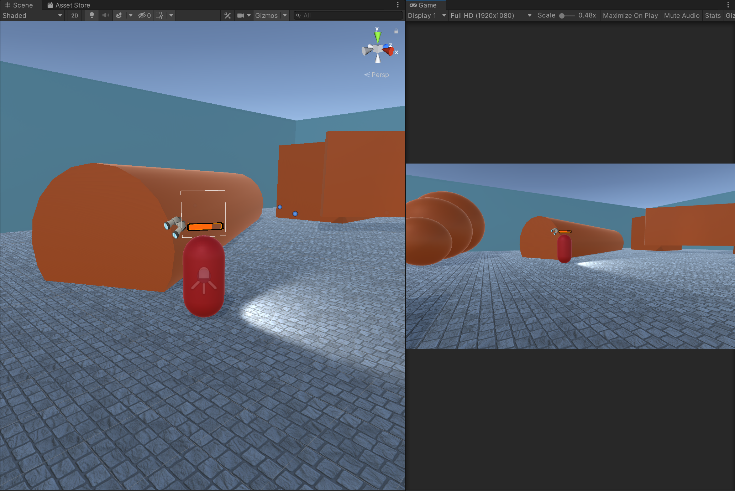


Screenshot showing Enemy ‘Searching’ state sprite. (Red Capsule is the Enemy. The left view is the Scene view. The right view is the Game view/Player’s view)

The second UI element used for this objective was an exclamation mark sprite. This sprite signalled to the Player that the Guards had spotted the Player and that they were either chasing or attacking the Player, this was referred to as an ‘Alerted’ state. This helped the playability a great deal. There was less ambiguity around situations such as if the Guard was simply patrolling around the map or if the Guard had spotted the Player and was chasing the Player down.

*Objective 2B*

There was a slight problem that arose when testing Objective 2B but overall the design and implementation were a success. Objective 2B involved having a UI element above the Guards` heads that would depict to the Player how close they were to being spotted by said Guard. The detection bar was made in Piskel(REFERENCE). The detection amount was set by a blank Image object inside Unity and was controlled by a Slider component built into Unity as well. The Slider component set the size of the blank Image based on inputs from the DetectionBarScript C# script. The script itself was being called by the CheckGuardSpotted class to set the detection level. The detection level was set by the time the Player was currently visible divided by the total time it would take to detect the Player in that particular zone. This implementation seemed to work fine until testing revealed a slight issue. To lower the detection level, the current time the player was visible was divided by the zone 5 timer(the furthest out zone from the Guard). This led to the detection amount of the bar jumping down when the player was not in zone 5 and the Guard lost sight of the Player. To fix this issue comprehensively, the detection bar lowering would have to be set relative to the zone the Player was in at the time of them being spotted. For example, if the player was caught in zone 2, the detection bar would be set by the current time the player was visible divided by the zone 2 timer. This change would fix this jumping but at this stage of the Project, it was not deemed necessary to take the time to redesign the solution for this implementation to fix this use.



Screenshot showing Detection Bar ¾ full. (Red Capsule is the Enemy. The left view is the Scene view. The right view is the Game view/Player’s view)

The last part of this objective was to change the colour of the blank Image based on how close the Player was to being detected, E.g. if the Player was very close to being detected and the detection bar was almost full the colour of the bar would be closer to red and if the player had just recently come into the Guards vision zone, the colour of the bar would be green. To achieve this, another built-in component of Unity was used. The Gradient component. Within the Gradient UI, the colours for the Gradient can be set. This was set to green at the start of the bar, orange for the middle and red at the end. The Gradient colour for the blank image could then be set in the same way the detection amount was set.

**Stealth Mechanics**

The stealth mechanics objective of the project entailed designing and implementing 2 different mechanics that would aid the player in being stealthy and limiting the use of these mechanics so that they aren’t overpowered.

*Objective 3Ai*

The first mechanic designed and implemented was a smoke bomb. The first thing that was produced in the making of the smoke bomb was a class that would spawn an object, in this case, the smoke bomb canister. The object was a physics object so once spawned the object would fall to the ground in front of the player. This was useful as it gave the player some feedback to let them know their input was registered. The spawn for the object could be modified easily for a future developer by changing the object that would be spawned and changing some parameters about it such as the throwing force. The current implementation could therefore easily be modified to implement a regular grenade or a flashbang instead.

Once the object was dropped, and the delay timer had run out, the object would be destroyed and a particle effect would take its place. In this implementation, it was the purple smoke which can be seen on the right(REFERENCE TO FIGURE). The smoke produced for this implementation was very simple. The smoke prefab(EXPLAIN PREFAB????) was defined to have the same layer mask as other obstacles in the game. In short, the enemies would see the smoke effect like a building or any other obstruction and therefore could not see through it.

However, there was more to the smoke bomb mechanic which was not successfully implemented. The requirements for the smoke bomb mechanic detailed the need for the enemies to be able to spot the smoke itself and react to it. For example, if an enemy spotted the smoke but never saw the player, the enemy would enter an alerted state and go over to investigate the smoke. This would’ve also prompted the other enemies to also investigate the smoke. After the smoke cleared, the enemies would’ve entered a search pattern. The solution to this problem should’ve produced an addition to the CheckEnemyZone class which would check if the smoke effect was within the enemy`s view cone and the enemy would`ve reacted accordingly. This was not successfully implemented.

*Objective 3Aii*

The second mechanic to implement was a Hiding mechanic. This implementation produced no code because Unity allows for efficient execution of the mechanic with the use of layers. The mechanic was simply designed as a mesh model that had a layer mask set to Obstacle. The same layer mask was used for buildings, obstructions and the smoke effect. This allowed for easy production of places for the player to hide in. Moreover, it is simple for future developers to take advantage of the simplicity of this feature and implement the feature easily into their own games.

*Objective 3Bi*

There were 2 methods produced to easily limit the usage of the smoke bomb mechanic. The first method was to limit the player’s ability to repeatedly deploy the smoke bomb in quick succession. A cooldown variable was added to the code which began running once a single smoke bomb had been deployed. Upon the completion of the cooldown timer, the player was again able to deploy a smoke bomb. This implementation is an effective way to slow down the pace at which the player deploys the smoke bombs. The implementation can also be easily adapted by future developers to change the cooldown timer.

The second method to limit the usage of the smoke bomb mechanic was to limit the total number of smoke bombs the player could deploy. The implementation produced an addition to the DropSmoke class which set only let the player deploy a smoke bomb if the smoke bomb count was above 0. Again, this could be easily modified in the Unity Inspector window by another developer to change how many smoke bombs the player should start with. Lastly, the first part of a player HUD was produced, which can be seen on the right(REFERENCE FIGURE). This was done so that the player could see how many smoke bombs they had left.

*Objective 3Bii*

A simple methodology was produced in order to reduce the usage of the Hiding mechanic. Since the hiding mechanic cannot be limited in the same way as a smoke bomb, the mechanic has to be limited by how and where the developer places the hiding locations on the map. In order to do this, the majority of the hiding areas were placed around the outskirts of the map and these areas were quite large. Whereas moving further into the main section of the level, the hiding areas were few and far between and much smaller in size. When developing the project further, this can be adjusted by play testing with a large degree of players and data analysis to see where the most common places players are hiding. This was beyond the scope of this project but future developers may seek to build upon this methodology for their own games.