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 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 enemy’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 is not, the algorithm would check how close the player is 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 guards’ periphery, one to represent a direct eye line (straight ahead) and one to represent the middle ground between the two angels. 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 visual 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

This objective was fraught with difficulties. Firstly, the project 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 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 class which controlled when the guards would search and when they would attack. The pseudo-code for this algorithm is as follows:

Fgdnbhiujgdfhisu;j

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 IC and 1A 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 actually 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 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 specifically is 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 have to return success for an action to be committed). This Sequence node has 2 children which are both leaves. 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 return success.

Therefore, 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 guard is nearby and actually perform an attack when the guard gets in range.

Since that last Sequence node return 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 return true regardless of the zone the player is in and even returns true 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 one of the guards` vision zone or decrements the timer if they are outside of the vision zone. If the guard 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.