[Abstract]

Computer Project

COmputer Science for Games

Jacob Pascoe - 17828121

2021

Contents

[2 Introduction 3](#_Toc69478255)

[3 Planning 4](#_Toc69478256)

[3.1 Gantt Chart 5](#_Toc69478257)

[3.2 Talking about Planning 6](#_Toc69478258)

[4 Literature review of Pathfinding Algorithms 6](#_Toc69478259)

[4.1 A\* Algorithm 6](#_Toc69478260)

[4.1.1 Heuristic Value 7](#_Toc69478261)

[4.1.2 Weighting 8](#_Toc69478262)

[4.1.3 Box Blur Algorithm 8](#_Toc69478263)

[4.1.4 A\* Pseudo Code 10](#_Toc69478264)

[4.2 Dijkstra’s Algorithm 10](#_Toc69478265)

[4.2.1 Heap Optimisation 11](#_Toc69478266)

[4.2.2 Dijkstra’s Algorithm Pseudo Code 12](#_Toc69478267)

[5 Methodology 12](#_Toc69478268)

[5.1 Chosen Pathfinding 12](#_Toc69478269)

[5.1.1 Algorithm Speed 12](#_Toc69478270)

[5.1.2 Algorithm Flexibly 13](#_Toc69478271)

[5.1.3 Performance 13](#_Toc69478272)

[5.1.4 Selected Pathfinding 13](#_Toc69478273)

[5.2 Game Engine 14](#_Toc69478274)

[5.2.1 Unity 14](#_Toc69478275)

[5.2.2 Unreal 14](#_Toc69478276)

[5.2.3 Selected Engine 14](#_Toc69478277)

[5.3 Nav Mesh VS Pathfinding 14](#_Toc69478278)

[5.4 Implementation 14](#_Toc69478279)

[5.4.1 Grid 14](#_Toc69478280)

[5.4.2 A\* Pathfinding 15](#_Toc69478281)

[5.4.3 Units 16](#_Toc69478282)

[5.4.4 Binary Heap 16](#_Toc69478283)

[5.4.5 Nodes 17](#_Toc69478284)

[6 Testing 17](#_Toc69478285)

[6.1 Pathfinding Testing 17](#_Toc69478286)

[6.2 Amount of Agents 18](#_Toc69478287)

[6.3 Optimisation Testing 18](#_Toc69478288)

[6.3.1 Binary Heap 18](#_Toc69478289)

[6.3.2 Path Request Manager 18](#_Toc69478290)

[6.4 Path Smoothing Testing 19](#_Toc69478291)

[7 Risk Assessment 20](#_Toc69478292)

[8 Review 20](#_Toc69478293)

[8.1 Pathfinding Review 20](#_Toc69478294)

[8.1.1 Local Avoidance 20](#_Toc69478295)

[8.1.2 Weighting System 22](#_Toc69478296)

[8.1.3 Path Smoothing 22](#_Toc69478297)

[8.1.4 Decision making 23](#_Toc69478298)

[8.1.5 Final Pathfinding Review 24](#_Toc69478299)

[8.2 Game Review 24](#_Toc69478300)

[8.3 Report Review 24](#_Toc69478301)

[8.4 Review Summary 24](#_Toc69478302)

[9 Conclusion 24](#_Toc69478303)

[10 References 24](#_Toc69478304)

# Introduction

* Talk about the goals of the project.
  + Apply pathfinding to a game in unity.
  + Optimization the pathfinding to work well.
  + Make a 3D space for the pathfinding to work around.
  + Have the AI to be able to make choices in where to go.
  + Have a player for the Pathfinding to follow.
  + Have the AI be able to shoot the player.
  + Have Models of the AI and weapons
* Talk about what has been done within these goals.

This project focus on pathfinding in a first-person game

The goals for this project are to get a pathfinding algorithm to work and run a number of agents in a game. The first goal of this project is find a system that will fine the short path on a given while also sticking to the easier path on this map. This will be done by adding a pathfinding algorithm which need to get form one part of the map to another part of a map while avoiding any obstacles on this map. This is the map part of this project as the main part is finding the shortest path. Another goal for this project is to have the agent to be able to move around of this map using the pathfinding algorithm. This is so it does not just find the shortest path on a map it can move around the map and update to the environment around it. while this is happening, the pathfinding has to update while the game is running so it can find a path towards a moving target this is so the pathfinding acts as an AI in this game. the next part of this is to make sure the agents can change their path depending on what is around them as more objects are added to the map the agents job becomes more than just walking towards the player it will have to make decisions to either grabs an object on the ground or to chase the player this will depend on how well this is developed. Another goal of this is to make a demonstration of a wave base game where the AI move towards the player and shoot at them. The main part of this part is so in the further this can show my skills as a programmer and can be shown on my portfolio as an achievement of what has been done in my time in university.

The reason for making a pathfinding system for a game is because I have wanted to add real agents to a game for as long as I have been on a coding course. As this was a goal that was do able but not easy, I found this would have been a good final year project as it takes a lot of research to find the correct system for this game. this also take a fair amount of coding to get it set up and working how I planned for the system to work. The goal for agent is to have agents that can make decisions while the game is running and change the path as they move around the map. These agent are designed to be able to shoot and find a weapon around the map. this little decision making is made so the agents do try to find a weapon when the player is closer to the agent then a weapon is. They are also designed to find the shortest lowest cost path.

# Planning

## Gantt Chart

Figure 1:Gantt Chart

## Talking about Planning

* Talk about the planning of the project and how I am going to keep up with the plan.

# Literature review of Pathfinding Algorithms

## A\* Algorithm

The A\* algorithm was proposed by Hart, Nilsson, and Raphael in the year 1968 (Hart et al., 1968) in this paper the writes suggest a new method of working out the shortest part to any node on a graph. This new method is the heuristic value of each node this is an improvement on Dijkstra’s algorithm. The lower the heuristic value is the close A\* will follow Dijkstra’s algorithm but as the value increases get close to the true distance of the path A\* will become more optimal and processed to find the shorter path (Millington and Funge, 2009). The heuristic value h(n) is the true value of an optimal path from the target node(*t*) to the current node(*n*) (Hart et al., 1968). As this is only part of the equation to work out the shortest path from *n* to *t* the other factor in this equation is the actual cost from start node(*s*) to current node (*n*) this is defined as g(n). Each node is set a movement value as the pathfinding moves through the node this will increase g(n).

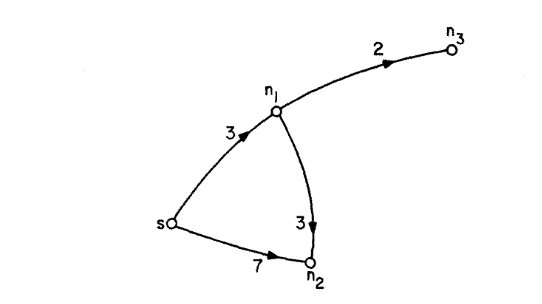


Figure 2:Subgraph For A\*

The graph in Figure 2 show four nodes as the pathfinding will start at *s* this means g(s) = 0 but as A\* moves through the graph it will increase. With this g(*n*3) = g(*s*) + g(*n*1) + g(*n*3) = 0 + 3 + 2 = 5. The finial part of this equation is the evaluation function f(n) this is the two values before added together.

This equation proves that as the heuristic value lower this algorithm acts more like Dijkstra as this only uses g(n) so the closer the heuristic is to zero the more the algorithm is acting on g(n). with this the algorithm change chose which path takes it closer to the target goal.

With this we can set out how the algorithm can search:

1. Mark as “open” and calculate f(s).
2. Select the next open node n whose f value is the smallest resolve ties arbitrarily, but in favour of the best node.
3. If not, mark n “closed” and terminate the algorithm.
4. Otherwise, mark *n* “closed” and apply the successor operator τ to *n*. Calculate f for each successor node of *n* and mark as “open” each successor not already marked “closed”. Let f(*n*) be the actual cost of an optimal path which got through n, from s to a preferred goal node n. determination of f(*n*) is the primary interest. Remark as “open” any “closed” node n which is a successor of n and for which f(*n*) is smaller now than it was when n was marked as closed.

(Hart et al., 1968) with this A\* can search for the shortest path from the start node and the target node and always find the quickest path.

### Heuristic Value

A\* behaviour will change depending on the heuristic value and how the user decides to calculate it, it the terms of games this will be the difference between speed and accuracy. With most game the user wants the algorithm to run fast than it is accurate as the user want a distance that is close to the correct distance but not exact for this trade it will make the algorithm run faster. There are a few main way of calculating this on a grid-based map these are The Manhattan distance and Euclidean Distance (Stanford.edu, 2021). All of these has uses on different types of grids.

#### Manhattan Distance

The Manhattan distance was designed for working out the distance between to point on a graph but was used in the city of Manhattan as well (Han et al., 2012). This distance is defined as

This only works if the two point are connected by a right-angle triangle otherwise this equation cannot be used the reason for this is because it will follow the opposite and adjacent rather than the hypotenuse (RedBlodGames, 2010). This is best for a grid that only allows movement on a square grid in 4 directions this is because of what is state above about the right-angle triangles (Szabo, 2015). Below is an example of the code that is used to implement this.



#### Euclidean Distance

The other way of calculating the distance between two points is the Euclidean Distance this is based of Pythagoras theorem and will calculate the distance of the hypotenuse of the triangle made from the two points on the grid (Han et al., 2012). This is defined as

This will only work on a 2D grid as it need a right-angle triangle form the equations to work. This is best on a pathfinding system that allows to pathfinding to move in any direction (Stanford.edu, 2021). Below is an example of the code use to implement this.



### Weighting

Weighting in A\* pathfinding is a way to change how the pathfinding makes decisions while running. this can make the pathfinding avoid area the user does not want the pathfinding to go such as grass and water area on a map. This can be done by changing the equation the is used to pick the next node (Algfoor et al., 2017).

With this equation above the pathfinding will get the heuristic but then multiple that number by the weighting system that has been put in. below are to images from the testing of this system. on the left the pathfinding system without the weighting this shows the pathfinding ignores any path and road that has been added to the map, the right shows the pathfinding with the weighting system added to the game. With this system the uses can make pathfinding think the shortest path is along the roads and paths not across the grass.



Figure 3a, b: Example of path weighting

### Box Blur Algorithm

The box blur algorithm is normal used for photos and making the pixels more smooths when the picture is being processed but as this works with pixels the method can be changed to work with the nodes on the grid in a game. this system work by taking 9 nodes and getting the average of all of these nodes(Elye, 2020) (Ngo, 2020). Below is the basic matrix that this system uses.

This works with the grid when the weighting needs to be smoother, so the agent walks along the middle of the path. In this is will take all neighbouring node/pixels and get the mean of those pixel. This makes it so when the are 2 pixels/nodes that have very different values in can make it more gradual as the values change(Elye, 2020). This system can be used on any size of matrix’s depending on what the user needs. As the figure 4a below show that when the box blur is not active the path and the grass are just black and white, figure 4b shows when it is active the paths blended towards the centre of the path.

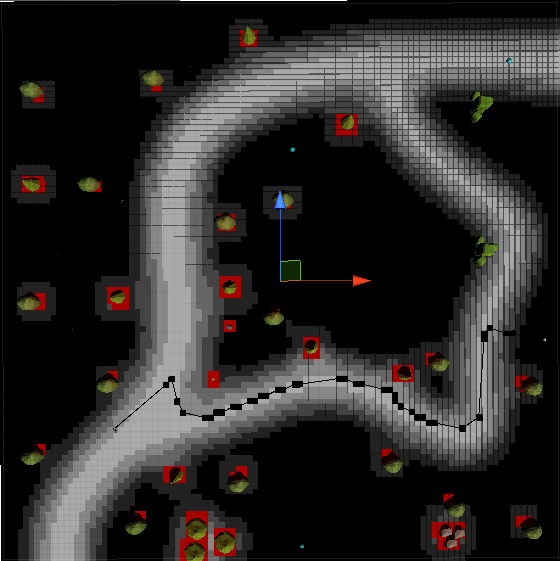
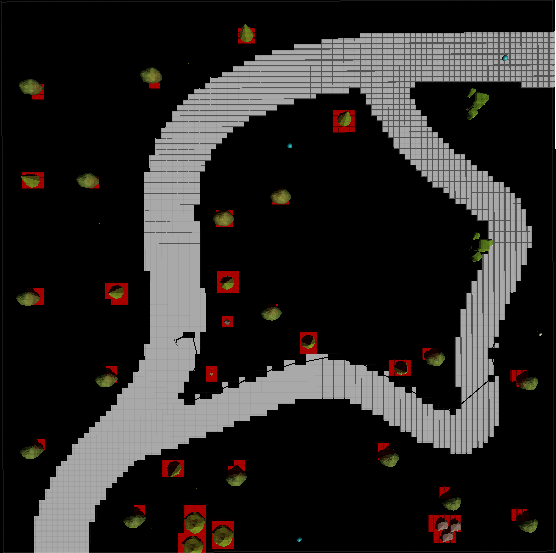


Figure 4a, b: Example of Box Blur Algorithm

### A\* Pseudo Code



## Dijkstra’s Algorithm

Dijkstra’s algorithm proposed in 1959 by E.W Dijkstra and is a way to find the shortest path on any graph using weighting systems (Dijkstra, 1959). The weighting system works by setting a value for all edges on a graph these edges go from on vertices or node. This pathfinding system works by checking most nodes in a graph until it finds the target node and retraces its steps back to the starting node.

Diagram

Description automatically generated

Figure 5: Dijkstra Subgraph

Above in figure 5 is a simple view of a graph that Dijkstra could work through in this example the algorithm would want to find a path from node A and node G. for this algorithm to work all nodes and edges are added are add to the unexplored list, once the algorithm starts node A or starting node is added to checking list. Then these two steps are follow until the shortest path is found (Dijkstra, 1959).

1. Consider all Edges that are connected to the node that have just been added to the checking list, then check if there is a shorter path to any node connect to these edges that are in the explored list and save that path if the path is not shorter reject that path. If the node is in the unexplored list add that node to checking list.
2. Move to the next node with the lowest path cost from the starting node. If current node equals target nodes the algorithm has found the shortest path if not go back to step one and repeat this until target node is found.

With this system the algorithm would have found the path A, B, D, G with a cost of 5. this is the system that A\* builds of (See 4.1) as A\* uses the heuristic value (See 4.1.1). this system uses just the g cost in A\* so it will take longer to find the path as it is not moving towards the target node, it just follows the lowest cost so far.

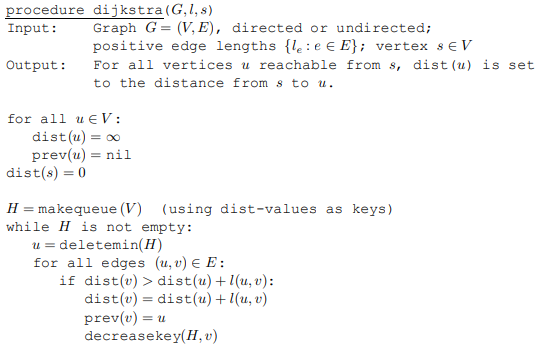
### Heap Optimisation

Heap Optimisation is a way of speeding up Dijkstra’s algorithm this uses a system to change where the values are in the checking list, so the lowest value is always checked next there are a few ways of doing this. The simplest way of doing this is with an array this does not sort any of the data that is put into it as it is last in first out, this method can work for smaller graphs but as more nodes are made in the graph this method becomes slower. Below is the equation for this where O is the time complexity of the algorithm and V is the number of nodes. (Dasgupta et al., 2006)

The other method is the binary heap this changes the order in the array, so the lowest value node is always first to leave the array. The time complexity of this method is as follows. This V and O are the same, E is the number of edges in a graph. (Dasgupta et al., 2006)

These two methods are uses depending on the graph if the graph has less edges than node2 then the array method works faster but if the edges are more than that binary heap should be used.(Dasgupta et al., 2006)

### Dijkstra’s Algorithm Pseudo Code



(Dasgupta et al., 2006)

# Methodology

## Chosen Pathfinding

This section will summarise the literature review and by the end will have explain why the chosen algorithm has been selected. This will explain the speeds of the algorithm, flexible of the algorithms, and performance of both.

### Algorithm Speed

Both algorithms are good and can be used in their own way, but Dijkstra algorithm falls short when game map becomes too large as there are too many node and edges. In the chosen game there is 1002 nodes on the map with each node having four edges. A\* time complexity is where b is the branching factor and d is the death of the target node in this tree. This is affected by the heuristic value as the closer this is to the true value the smaller the branching factor will be as the user want the branching factor to be as low as possible as d increases it takes roughly 10 time longer for it to search. Whereas Dijkstra does not have this if the user is using the binary heap the time complexity will be as this algorithm searches more node and edges than A\* it will take longer. This is also a factor when the graph is getting bigger and increasing the node and edges. With this in mind A\* will be a better pathfinding with time taken to find the shortest path. Another reason why A\* is faster than Dijkstra algorithm is because A\* searches in the direction of the target node seen is section 4.1.1 with this the algorithm does not need to search as many node making it faster to find the target.

### Algorithm Flexibly

The A\* algorithm can be manipulate as the user need for different maps and different sizes. This is shown by the heuristic and weighting of this algorithm this makes the algorithm take different path that are not the shortest part but can be when paths and water are added to the map. This means A\* can be made to make paths that are more realistic and truer to AI. Where Dijkstra algorithm does not have these features and is harder to find these paths that are need for games that are not mazes. As this system only use one cost that is cost so far this make it hard for the user have the pathfinding go on any other than the shortest path. This would be good if the map that was made was a maze but as there is different routes to the player this pathfinding does not fit with this category. With this section A\* preforms better with flexibly.

### Performance

On this section I found a website ( [https://qiao.github.io/PathFinding.js/visual/](https://qiao.github.io/PathFinding.js/visual/%20) ) that has pathfinding algorithms add on to it so the user can test the speed, accuracy, and number of operations. On this website I tested A\* with Manhattan and Euclidean distance and Dijkstra’s and recorded to results in the table below.

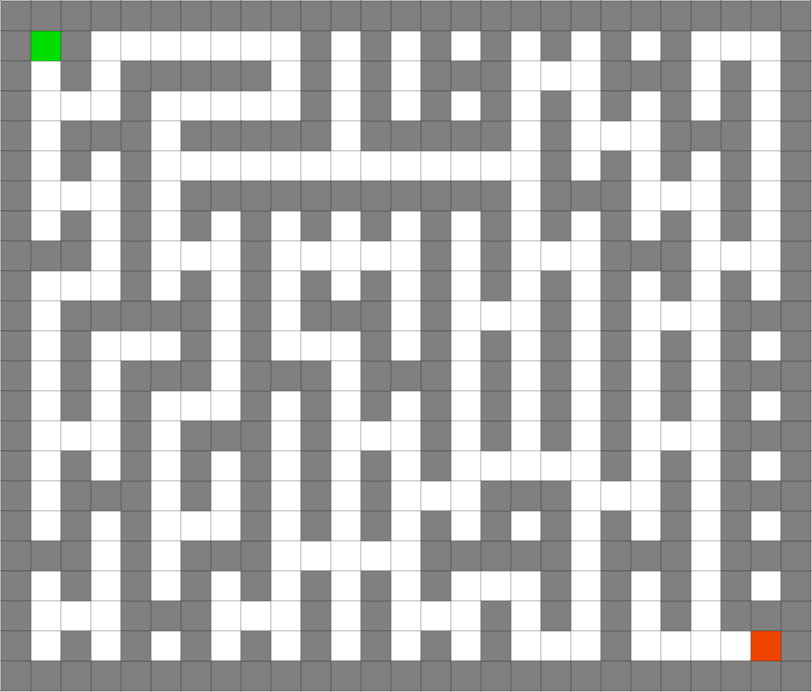


Figure 6: Test Maze

Above shows the results from the test of the website. this shows that the Manhattan distance works the best with this maze that has been made. As this pathfinding has the shortest time and lowest nodes check this makes it a good choice for a pathfinding that has to be run multiple time a second during the run time of the game. In this Dijkstra’s still finds the shorts path but takes longer than the Manhattan distances and check more nodes. The reason why how many nodes are checked is because all of this takes memory, and this will affect how the game runs and how quick the AI can changed directions.

### Selected Pathfinding

The Pathfinding picked for this is A\* as this works a lot better than Dijkstra is all categories this mainly comes down to the fact A\* is an improvement on Dijkstra algorithm and has added features which makes the algorithm perform better. As is this pathfinding works faster, works better on bigger maps is checks less node and moves towards the target node. With these test it makes A\* the obviously chose out of the two pathfinding algorithms.

## Game Engine

This section will explain why one of the game engines was chosen over another engine.

### Unity

### Unreal

### Selected Engine

Both of these game engines are valid options for doing this project. in this case neither of the two engines are better than the other. With the engine that was selected was unity, the main reason for this game engine to be selected was that I have more experience was unity than I do with unreal engine as I have made serval games on unity and have only made one game on unreal. With this is mind unity was the clear option as I know the engine works a lot better. another factor for picking unity is because it does not have blueprints, I found this was I good thing because if I use unreal, I would have used the blueprints and I find that is messy and find it hard to understand what the blueprints are doing this is partly to the fact that unity is more intuitive to learn on.

## Nav Mesh VS Pathfinding

These two system work similar as they do the same pathfinding. The main different it that the nav mesh is built into unity and can auto generate a grid that avoids any obstacles. This is in place as away to have agents in the game without having to add a pathfinding algorithm. This make making a game easier to do. this does have a weighting and cost system where the agent can move along paths. My goal going into this project was to make an agent that uses a pathfinding algorithm, and this was looked into but does not fit the requirements for a finial year project as unity hands all of the systems. This also cannot be optimised to use as little memory as possible. Both are viable options when making agents in a game but as I am doing a coding course, I want to make code and add a system to find the shortest path without a systems in unity.

## Implementation

This will explain how the code was done and how this was added to the game. The part this will go over will be the grid creation, pathfinding, moving the agents, and binary heap optimisation. Add more as they get added.

### Grid

The grid script is designed to map out where the nodes are placed on the map and to set the values of the node on the map. The first part of the script is creating the grid on the map this is set to make the grid to the size of the map and can be changed depending on how the developer needs. This part also set the size of each node this is done by setting the value of a node radius this is done in the unity inspector window. Along with this in the awake function the terrain type is set this is part of the weighting system, this uses a ray cast to check each node and see what layer mask it is overlapping with. In the inspector the developer can change what layer is set and what the value is for this layer mask. This data is stored in a dictionary the main reason for the dictionary is because 2 bit of data can be added and only one need to be called to get both bit of data. When the ray cast checks each node in the gird it will try to get the value from the dictionary and output the data of the movement penalty that is set in the inspector this value is then set to that node.

When the create grid, function is called the function will find the bottom left conner of the map check if the point of the grid is not a wall by checking around the node to see if the unwalkable mask is in the radius then if that node is walkable, it will check what the movement penalty is using the method above. Then this set the node using the node class with the data of; if its walkable, the X and Y positions and the movement penalty of this node. This runs until all points of the grid has been don. This then gets passed to the box blur algorithm (See 4.1.3).

The Box blur algorithm is a way to get the average value of 9 bit of data in a matrix. The reason for this in this game is because of how the argents moves before this point. The agents would just move alone the side of the path as this was the shortest path with the lowest weight value. This algorithm combines all values from 9 node adjacent to this node because of this the middle of the path becomes the lowest weighting on the path. With this the agents move along the middle of the path and the path becomes smoother.

### A\* Pathfinding

This script handles the A\* and where most of the calculations for the pathfinding is done. the main part of the script is the coroutine that uses the algorithm to find the path for the agents in the game. This take two vector3 positions on the map these are the agents position and the player’s position. When this is call these positions are converted from vector3 to nodes on the grid. With this data the algorithm checks if both nodes are walkable the reason for this is so the algorithm does not run either the target or start cannot be reach as it would waste memory and would not need to run. This then opens a heap list[[1]](#footnote-1) for sorting the nodes and for any open node and opens a hash set for any nodes that have been closed and are not needed anymore. This then adds the start node to the open heap. This then only runs the algorithm while the open heap has more than zero item in it as if there is nothing in the open list there will be nothing for the algorithm to search. This will also check if the target and current node are in the same place, so the algorithm does not run when it is not needed to save on memory. This then checks all the neighbours of the current node while also checking if any of the neighbour’s nodes are in the closed list and walkable as there is no need to check node that have been checked and are not walkable. This then gets the G cost of that neighbour node and the H cost multiplied by the movement penalty of that node and add the two values together to get the F cost of that node. It will then check is the neighbours G cost is lower than the current F cost if this is true then the pathfinding will move to that node and run this again. This also sets the neighbour node G cost to the current F cost and set the H cost again, it will also set the neighbour node to a parent of the current node this is so the pathfinding can retrace the path and find the shortest path.

This script also handles the simplify path, this was added for testing so when developing I could see if the pathfinding were working correctly. This just makes it so the on-screen gizmos can be seen. This checks if the path has changed direction and will draw and new line to the next node this makes it easier to see the path. This converts the last node and the new node and checks the direction it is going in if it changes add this to the finial path array for the retrace path.

The retrace path function just calls the simplify functions and then reverse this path that has been given back from the simplify path function this is so the units script can know what path to take the agents on.

### Units

The units class handle all agents add request the path for it to follow from the pathfinding script. This script first tried to request the path in the update function this would request a path sixty time a second, I found this cause the game to lag and it would not be able to handle more than two agents with a push. This was changed to use the invoke repeat function works like the update class but the developer can change how often the function is called this was tested with a few values the best was found to be 0.5 seconds this made it so the agents could get close to the end of the path before a new path was requested. This then checks for visible weapons and will compare the distance of each weapon in the agent’s field of view and find the closest one to the agent. This then calls the function choose path this takes the distance of the player and weapons in the range of the agent and works out which path to go on, if the weapon is closer it will go towards the weapon if there are now weapons it will always go to the player. Once a path has been found it will follow that path by moving the transform of the agent towards the next node until it has reached the target node then the pathfinding will be terminated until a new target node is made.

### Binary Heap

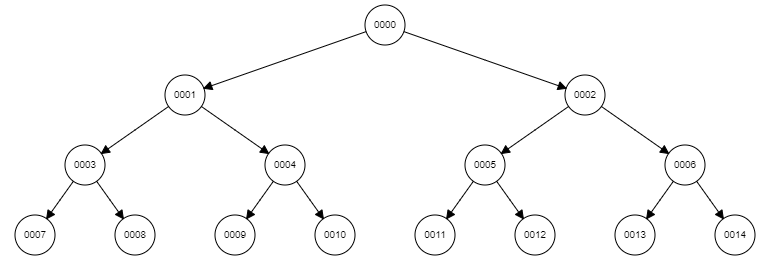
The binary heap is one of the most important bit of cade. As this sort where were each of the node are stored in the list as this make the node with the lowest F cost be at the top of the list. The reason this is important is because when the pathfinding is searching for a path it will always want to check the node with the lowest F cost first as this will bring it closer to the target node. This is done by having each node have two child at a point. The rule of this is that each child has to be less than the parent node. The reason this is importance is so the node with the lowest value is at the top. This is done by adding a new node to the list if the node has a value bigger than the parent nothing happens. Id the parent has a higher value than the child there are swapped until the node has found a point where it is less than its than is child node and more than its parent node. If there is no node that is less than this node then it must be the lowest node in the list. Because of the way this is done less node have to be checked to find the lowest node in the binary heap. If this were not in place and an unsorted array was used the code would have to compere every node. this is fine when there are only 5 node in the array as it will only have to check 4 different node but when the amount in the array increases the time will increase. 

Figure 7:Binary Heap Tree



Figure 8: Binary Heap in Array

Once the first node in the heap has been removed the code will take the last value in the binary tree this will be the bottom right-hand item and move it to the top of the tree and check each of this node children and swap it with its lowest valued child. Below is how a binary tree is visualized but it is really stored as an array. To find each point in the tree the program only needs to find the children of itself, for the left child its and for the right its this will always find where in the array the child is. The way to find the parent is . This words because when the program divides an integer and get .5 it will round down to the closest whole value.

### Nodes

The node class handle all the node data. This is what is used in all of the code explain above, this code set the value of each node. this is mainly done in the grid script, but the pathfinding and units scripts use this data. This is where all the data for is the node walkable, movement penalty, position, and gird position. This also holds the data for the G, H and F cost it is also used to work out the F cost by getting the cost of the G cost and H cost and adding them together. This script also has function to compare nodes together to see which node is more promising to go to next the pathfinding script used this function. The main used of the script is to be a constructor class for the grid. This also always the node to know which node is its parent so the path can be retraced, and the shortest path can be found.

# Testing

## Pathfinding Testing

Testing the pathfinding was to see if the A\* algorithm could find a path from point A to point B with no obstacles on the map this was just to see if the system can find a path from one point to another. This was done by making the nodes in the grid visible and once the path was found these node would turn red showing the route of the path from A to B this would prove that the system can find a path.

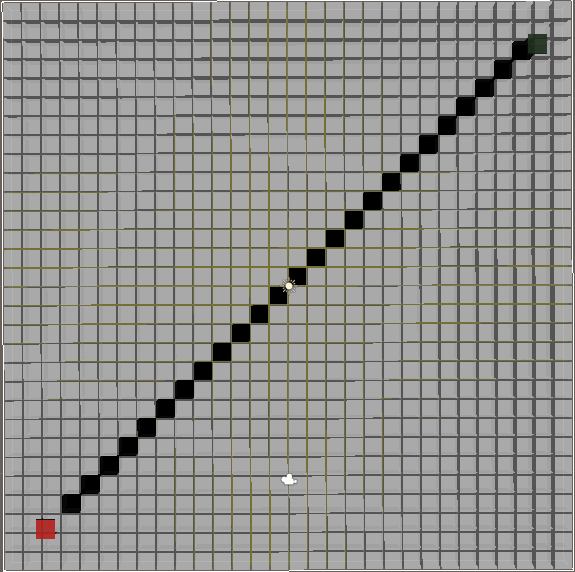


Figure 9: Pathfinding Testing

This then moved to adding object to the map to see if the pathfinding can find a path with object around it. this was still done by showing the grid and checking if the node will check the layer mash of the object as walls this was done by making the gird show any nodes will a wall in it as blue this would just check if the node would detect the will in the area of the node. This was then testing with the pathfinding with the same method as above showing the path as red. This was to make sure the pathfinding could find a path around object in the map.

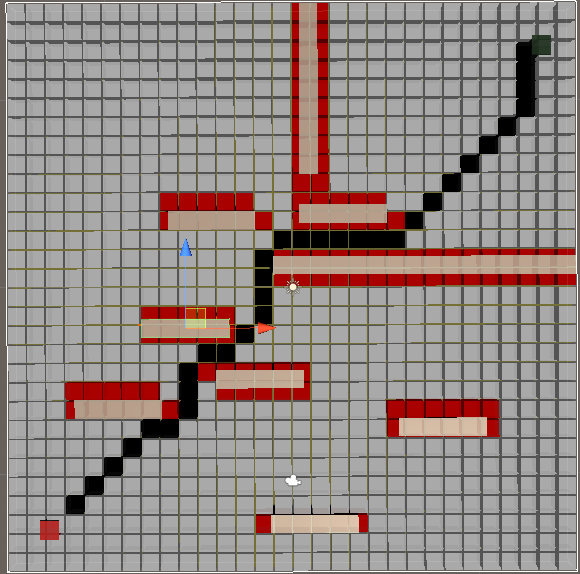
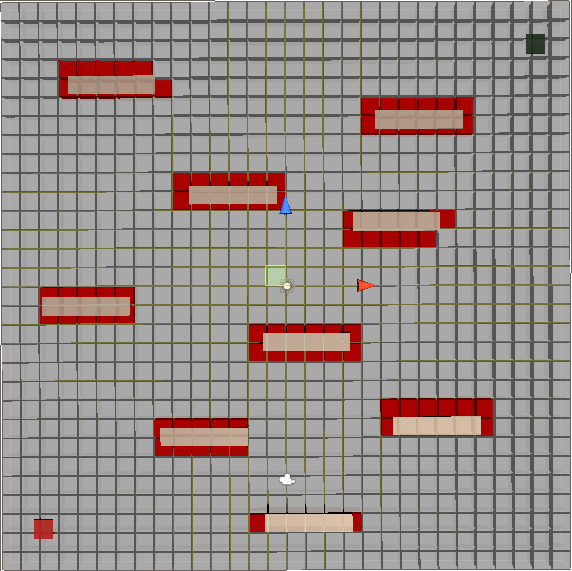


Figure 10a, b: Pathfinding Wall Detection Test

Finial the pathfinding was tested to see if the pathfinding could find a path of an object that is moving. This was done by changing the pathfinding code to update while the game was running and move the target and seeing if the path change as the target moved. This work finds with just one pathfinding running.

## Amount of Agents

This was a stress test for my game and what has been made. For this test I add a new agent every time to see how many agent the game could run at one time until the game starting lagging, or the pathfinding stopped working. During this test I found that the max amount of agent in this game was 15 anything more that this would start to lower the frame rate and lower the performance of the pathfinding as the engine could not handle the amount of request for a path that where being sent.

With the original code I found as soon as 3 agent where added the pathfinding would not run as well this is where I found that requesting a path sixty time a second was to much and had to lower the amount of time the path was requested per second.

## Optimisation Testing

### Binary Heap

This was this to test if adding binary heap will improve the speed the that the pathfinding will find the shortest path this was tested by checking how long the pathfinding took to find the path with out binary heap and test how long the pathfinding took with binary heap. With this test the path took 25ms with out binary heap and took 5ms to find the path with binary heap, so this was a success and was adding to the code.

### Path Request Manager

This system was added so more agents could be running at once. This was tested by seeing how many agents can be added before path request was add which was 5 agent before lagging and once this was added the game could handle 15 agent this system proved to be a success and was kept in the game as more agents could be handled at runtime.

## Path Smoothing Testing

This is a system to make sure the pathfinding goes along the paths I have created in the game and not along the grass. When this was first add the test was to find the best values for the grass, path, road, and water. This was done by set the road and path as zero as this is where I want the algorithm to follow this route and only leave it when it has no other choice. The values I had to figure out is how to keep the algorithm off of the grass and the water. This is because in the map there is a point in the map where the path can be cut across below you can see this part inside of the red circle.

|  |  |
| --- | --- |
| Weight of Grass | Does the pathfinding cut across |
| 0 | Yes |
| 1 | Yes |
| 2 | Yes |
| 3 | Yes |
| 4 | Yes |
| 5 | No |
| 6 | No |

AS this table above shows that when the weighting value for the grass is above 5 the pathfinding no longer moves across the grass as this is what I want the this is the weighting the grass will stay at as it will no longer cut across the grass.

Another test that had to be done while designing this code is the box blur algorithm this makes the agent walk along the middle of the path this had to compare against the pathfinding when this was not in place if the agent did not walk along the middle of the path, then the code had to be changed until the agent would walk along the middle of the path.

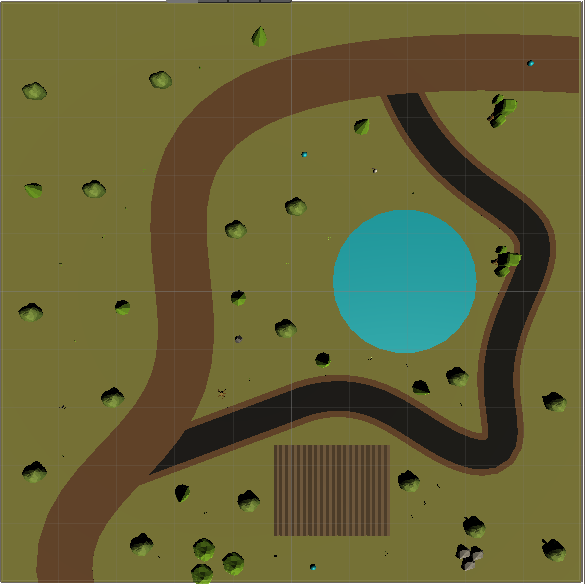


Figure 11: Map Design and Weighting Test

# Risk Assessment

Move Risk with working on a computer for long periods of time are covered by the health and safety (Display Screen Equipment) regulation act this covers any risk for a user who uses there any DSE for an hour or more.

|  |  |
| --- | --- |
| Risks | How to avoid Risk |
| Eye Strain for look at a screen for to long of a time period | Make sure the user takes regular break in this case a break every one hour, adjust screen brightness so the glare does not injury eyes. |
| Back strain for not have the correct posture | Make sure the user is following the DSE recommended sitting posture. This can also be fixed by making sure the computer monitor is at the correct height |
| Repetitive strain injury | Maintain good posture, take regular breaks |
| Stress | Make sure a little work is done every day, so the workload does not get too big. Also take regular break of something the user enjoys taking their mind off of the university work. Also make sure the user is having enough sleep, so they are not tried. |
| Mind fatigue | Take breaks while work and also plan rest day where the users do not have to think about the university work. |

All of these risk is accounted for over the run time of this project and have been in forced so the user does not encounter any of the risked in this list.

# Review

This section will cover all points of the review that could have been improved and done better this will cover the main part of the project such as the pathfinding, the game itself, and the project wright up.

## Pathfinding Review

In this section I will talk about how in the future this system could be improved on and making the pathfinding system better this will cover local avoidance, weighting system, path smoothing, and decision making.

### Local Avoidance

This is an improvement that would have made the pathfinding a lot better than it is as this would make it so the agents in the game would not enter to same node as another agent this would make it so when the game is being played if the agents are not remove of the map, they end up bunching up in to one section and all follow the same path. After doing some research on this I found a paper that goes over what I would have need if I need about this problem soon in the project. this paper explains how collision can be avoided by using the velocity of each agent.

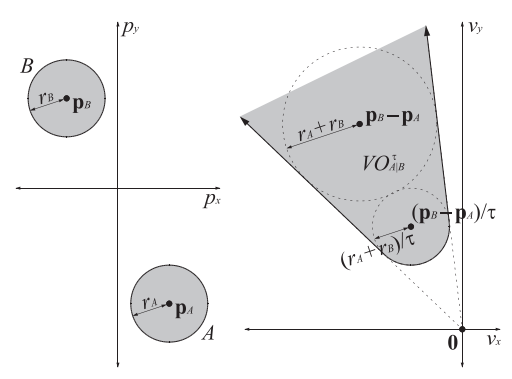


Figure 12: Velocity Avoidance

This figure shows how agent A can avoid agent B this is done by making a truncated cone from point 0 also where agent A is. This cone is made by having the legs tangent to the disc of the radius of rA + rB at a point if pb – pa and the disc of the radius (ra+rb)/t this means as the time constant increase agent A will not hit agent B (van den Berg et al., 2011). This means that if the agent is traveling in a direction that enters this cone the two agents will collider with each other. This algorithm used the velocity of each agent to work out what direction it can go to avoid a collision with another agents. With this system the user can make a system where all agents can got from point A to point B with out hitting another agent. When this is done for agent A and presumes agent B is not moving and will check this angle again at the next time constant. Below is the graph for a moving target where agent B has selected its velocity this time the algorithm has to check each point of the area agent B could end up in (Dark Grey) and perfume the same calculation as before in Figure 12. This will equal an area (Light Grey) where agent A cannot move in (van den Berg et al., 2011).

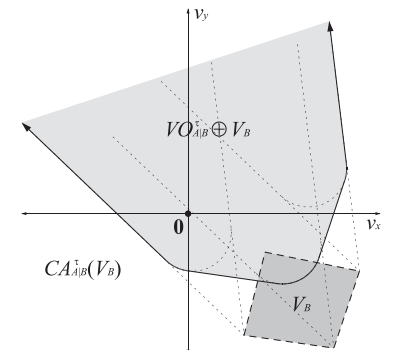


Figure 13:Velocity Avoidance on Moving Object

With this system in place in would make my game agent move in more realistic movement and not go on the same path as each other. This system was found to later into the project and seem very advanced to add into this project this is why it was not added to the project. Unfortunately, this was not add as it would of improve my agent and made the game a lot better. while researching I did find library that could have added this to make game, but I wanted everything about the agents to be code that was created by myself.

### Weighting System

Not sure what to put here.

### Path Smoothing

The path soothing in this game works quiet well, but this can be improved on as this method just has the agent moving from one point to another in a straight line. With this system the AI can look quite unnatural as it is following each waypoint and does not do any gradual turn. Figure 14 shows how the agents in the game move as they just move in one direction for this to be improved the agents would have to start turning before, they hit the node.

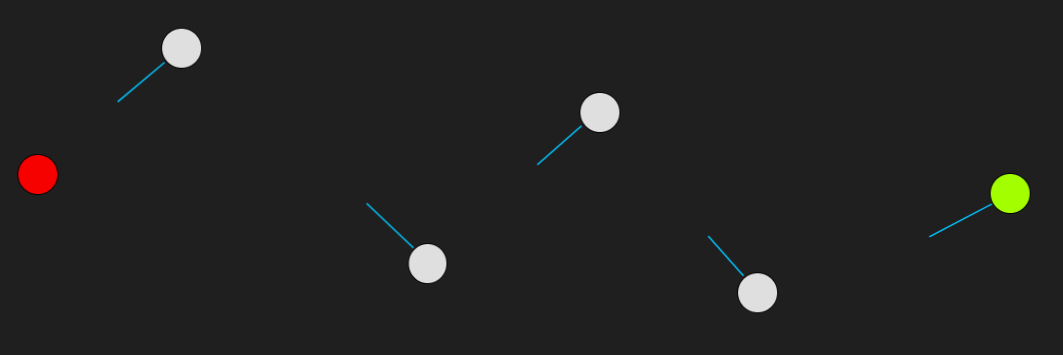
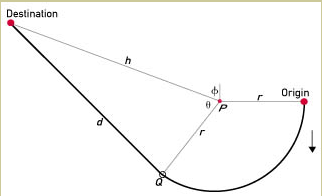


Figure 14:Showing How Pathfinding Move in Game

this system works by calculating a turning radius for the agent in the game. This works by finding the centre of the turning radius which would be the radius of the turning circle. This can be done by setting a distance from the node when the agent will start its turn. Once this is done the agent being turn with the turn radius that has been set by the used all it needs to do now it work out how long to keep turning for. They way this is done is by having the code work out the distance form the centre point of the turn radius and the next location this will be defined as h this will make a right-angle triangle with the radius of the circle and to work out the length of the straight line, we use Pythagoras theorem to work that out. The only part left is to work out is the angle form the point the circle is made to when the agent needs to start on the straight line. 

This feature could not be added to the game as I do not know enough about circle theory to have this in my game as the website this information explained it a little, but the user needs good knowledge of how circle theory worked to get this in the game. this could have been done if I had found this reason sooner and gave myself more time to do the research on this so I could fully understand how this system works and included it into my game.

### Decision making

The agents in this game have good decision making where they will choose a path depending on what is in their line of sight and what is closer. This could have been improved by have a weighting system on the guns so the agents could be up better guns and decide which gun is the best. This could be done by having a weighting system on each gun and making the agent look for the gun with the highest weighting system. with this the agents chose change what path they are going on not only on what is closer but what is consider the best gun on the map. This would make the agents react more to the world around them. This could not be added to the game as the game had been finished and the report had to be finished and the coding was done for this project.

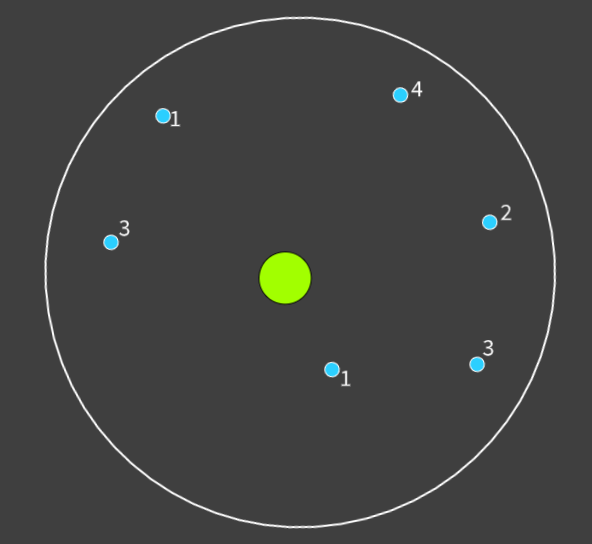


Figure : Decision Making Design

In figure 15 it shows the design of this system where all the weapons have a value, but the agent no longer look for the closest weapon it will look for the weapon that is in its field of view but have the highest value. While this is happening, it will also look for the gun with the highest value that is closest to it. in the example the closest weapon has a value of one, but the agent will get the weapon with the value of four, so it has the best weapon around it. this system would stay in place so if the agent found a weapon with a higher value on the floor it would swap that weapon out for a better one.

### Final Pathfinding Review

Overall I found that the pathfinding in this game was a success as the agents do what I set out for them to do each point above was a way to improve this system further, I found that the A\* pathfinding once in the game worked well and did all task in a quick manner this mean that the system could be used at run time, this system could also make decision on what why path to take and what target to follow. I found that the path smoothing works very well ever thought it can be improved on as shown in 8.1.3 but for the game these agents did not need this as they do follow the path and road in the game in the middle of the path which was a goal of the project. Each agent will take different paths and will not follow the same path when they are further away for the player.

* if weighting could be improved, HOW?

o if shooting could be improved, HOW?

## Game Review

What I felt like has gone well in the game and how any of the part oof the game could been improved

## Report Review

Not sure if this is needed.

## Review Summary

# Conclusion

(Bourg and Seemann, 2004, Rabin, 2008, Dasgupta et al., 2006, Sedgewick, 1983, Millington and Funge, 2009, Han et al., 2012, Hart et al., 1968, Ngo, 2020, Dechter and Pearl, 1985, RedBlodGames, 2010, Tree, 2020, Elye, 2020, Cormen, 2009, Szabo, 2015, ScienceDirect, Algfoor et al., 2017, Dijkstra, 1959, van den Berg et al., 2011)

# References

ALGFOOR, Z. A., SUNAR, M. S. & ABDULLAH, A. 2017. A new weighted pathfinding algorithms to reduce the search time on grid maps. *Expert Systems with Applications,* 71**,** 319-331.

BOURG, D. M. & SEEMANN, G. 2004. *AI for game developers,* Sebastopol, Calif, O'Reilly.

CORMEN, T. H. 2009. *Introduction to algorithms,* London;Cambridge, Mass;, MIT Press.

DASGUPTA, S., PAPADIMITRIOU, C. H. & VAZIRANI, U. 2006. *Algorithms*, McGraw-Hill, Inc.

DECHTER, R. & PEARL, J. 1985. Generalized best-first search strategies and the optimality of A\*. *J. ACM,* 32**,** 505–536.

DIJKSTRA, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik,* 1**,** 269-271.

ELYE. 2020. *How to Blur an Image on Android* [Online]. Available: <https://medium.com/mobile-app-development-publication/blurring-image-algorithm-example-in-android-cec81911cd5e> [Accessed 02/03/2021].

HAN, J., KAMBER, M. & PEI, J. 2012. *Data mining: concepts and techniques,* London;Amsterdam;, Morgan Kaufmann.

HART, P. E., NILSSON, N. J. & RAPHAEL, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics,* 4**,** 100-107.

MILLINGTON, I. & FUNGE, J. D. 2009. *Artificial intelligence for games,* Burlington, MA, Elsevier Morgan Kaufmann.

NGO, P. 2020. Game Application Using A\* Pathfinding Algorithm to Help Improving Dementia.

RABIN, S. 2008. *AI game programming wisdom 4,* Boston, Mass, Charles River Media.

REDBLODGAMES. 2010. *Heuristics* [Online]. Available: [http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#:~:text=A\*'s%20Use%20of%20the%20Heuristic%23&text=If%20h(n)%20is%20always,\*%20expands%2C%20making%20it%20slower](http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#:~:text=A*'s%20Use%20of%20the%20Heuristic%23&text=If%20h(n)%20is%20always,*%20expands%2C%20making%20it%20slower). [Accessed].

SCIENCEDIRECT Manhattan Distance - an overview | ScienceDirect Topics.

SEDGEWICK, R. 1983. *Algorithms,* London;Reading, Mass;, Addison-Wesley.

SZABO, F. E. 2015. M. *In:* SZABO, F. E. (ed.) *The Linear Algebra Survival Guide.* Boston: Academic Press.

TREE, S. 2020. *How Dijkstra's Algorithm Works* [Online]. Available: <https://www.youtube.com/watch?v=EFg3u_E6eHU> [Accessed].

VAN DEN BERG, J., GUY, S. J., LIN, M. & MANOCHA, D. Reciprocal n-Body Collision Avoidance. *In:* PRADALIER, C., SIEGWART, R. & HIRZINGER, G., eds. Robotics Research, 2011// 2011 Berlin, Heidelberg. Springer Berlin Heidelberg, 3-19.

[https://github.com/Pascoe007/ScuffedHot](%20https:/github.com/Pascoe007/ScuffedHot)

‌

1. A heap list is a way to store data where each item has two children and a parent, this will be sorted as an array with the order of the binary tree set out left to right. [↑](#footnote-ref-1)