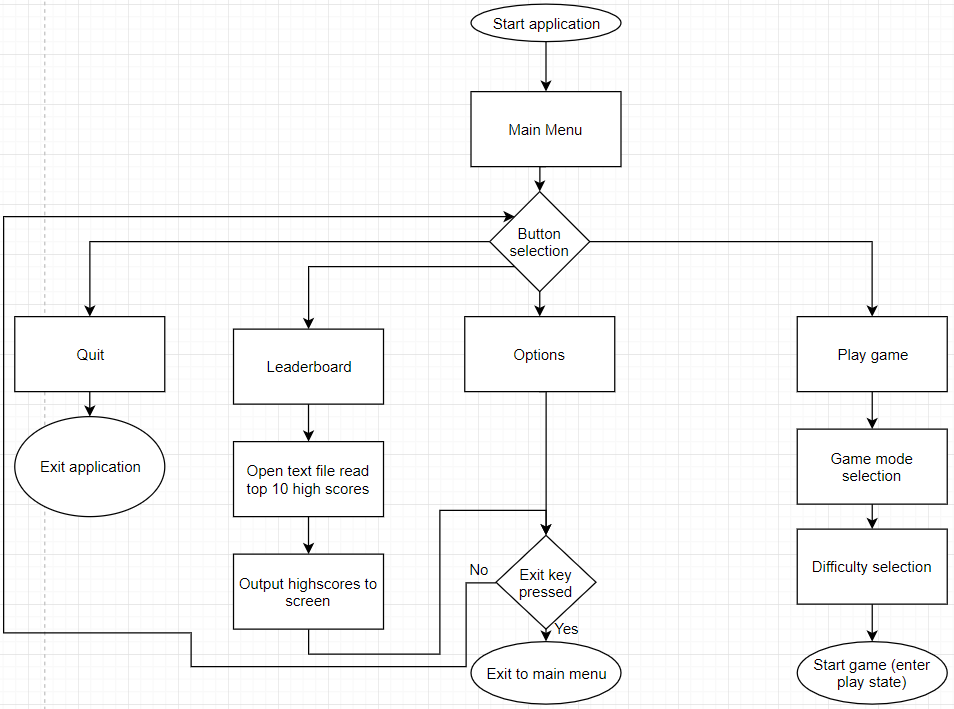
# 2 - Design

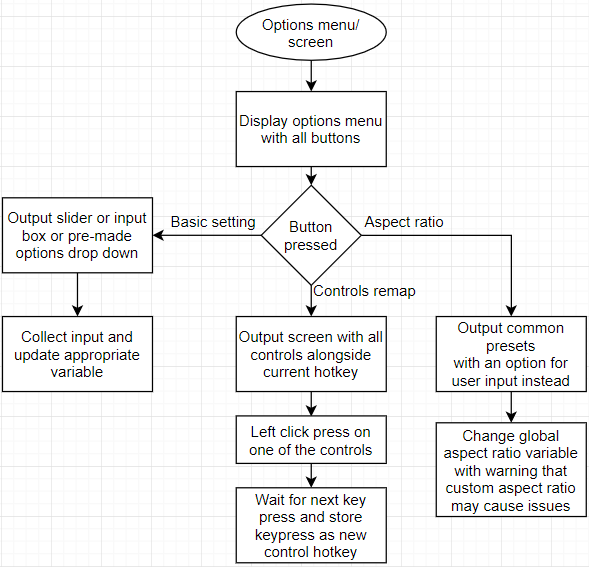
## 2.1 High level overview

Flowcharts shown here describe an overview of the way my project will work. As with most games and applications, when you launch the game you are met with a home screen. The home screen will have buttons that send them to different parts of the game. These locations may be options configuration, play game or quit. Each of these (bar quit) will be their own menus.

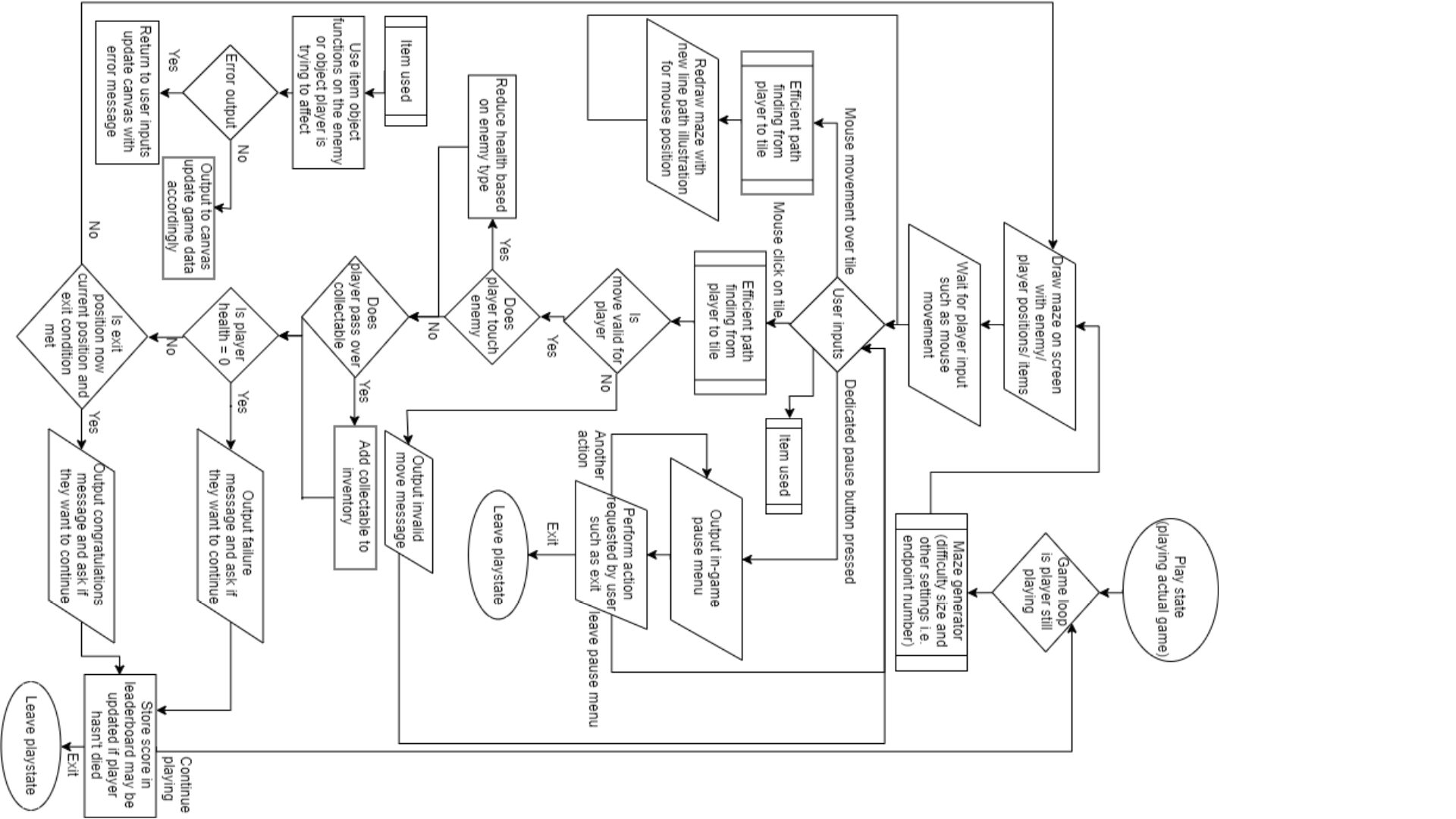
**Menu:**

This is the flowchart design for the main menu screen. Players will be greeted by this screen when they start the application. It needs to be simple in nature so that users don’t become confused when trying to start the game.

**Options:**



Due to the nature of Pygame as a library aspect ratio must be dealt with separately from other settings. Custom aspect ratios may cause issues since pixel positions are the main way most graphics are dealt with. Furthermore, controls remapping must be done in a specific so that any key on any keyboard can be mapped as a hotkey.

**Play state:**

## 2.2 Description of game layout

This is a representation of the different screens that will exist within my game:

Main menu

Play

Game setup

Game screen

In game options overlay

Leaderboard

Full leaderboard

Options

In the game setup screen users are given a choice of difficulty level size enemy quantity and number of objectives. After the user confirms their choices the application will enter the play state in which the user is playing the game as shown in the play state diagram. Once the game is over the player can choose to either continue/ retry or leave. This decision will choose whether to remain in the play state or move back to the main menu state. In game options will be separate from normal options since users won’t want to be sent back to the main menu version in game. This means the game will be paused and a smaller “window” of sorts will be displayed above the game within the same surface. Once the user finishes with the in-game pause menu the game will no longer be paused and users can carry on playing. Users can also quit the game halfway through in which case the current state of the game will be stored.

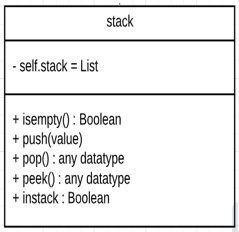
Input/output/storage table

|  |  |  |  |
| --- | --- | --- | --- |
| Inputs | Processes | Storage | Outputs |
| - mouse clicks  - keyboard presses (based on current key mapping)  - Score and Username | Calculate most efficient path to maze position  Calculate enemy line of sight and player position  Sort high scores into top 10 and separate into csv format. | CSV file sorted by high score  CSV file storing game status after exiting the game | Draw player/ enemy position  Draw game maze  Sorted list of high scores |

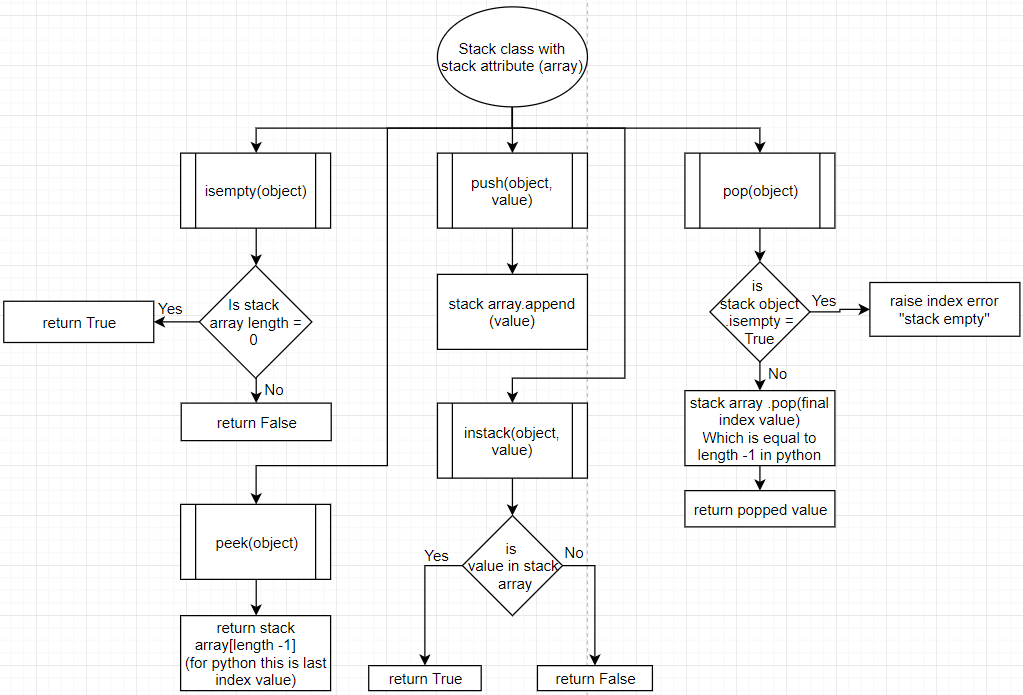
## 2.3 Data/ data structures

My game will be programmed using event driven programming. This means that the game will be run from states. These states could be menu state, play state, options screen etc.

For my maze generation algorithm, I will need to use a stack to store the history of the positions visited whilst creating the maze. Below is design for the stack. The stack has an extra method when compared to a normal stack – instack(). This will check if an item is in the stack.



The stack will be used to store a history of positions visited during the execution of the maze generation software. This means that the program needs to be able to know whether it has visited a node or not. Recursive backtracking isn’t used here since that creates a perfect maze (one with no loops) and for my maze to be beatable there must be loops. The stack is used also to backtrack if the maze generation has run itself into a dead end. In this case the maze must go back to a previous position and try a different route. This is outlined in the flowchart for maze generation in more intense detail.



**Frequently used variables / data:**

* Colour variables (hex values in array) (global)
* Grid data structure (or class object/ maze array)
* History stack
* Enemy position array
* Player position array
* Powerup position array
* Entity layer grid array (used to describe player positions on the maze)
* Leader board cache array
* Screen width integer
* Screen height integer
* Button class object

**Heap / priority queue:**

Priority queues will be used multiple times throughout this project mainly to facilitate efficiency improvements. This is for algorithms such as Dijkstra pathfinding and ray casting to improve efficiency of execution.

Python (the language which I will be using) has a library specifically for priority queues. This implementation uses a minimum heap to store items in a priority queue. This implementation seems to function faster than a theoretically more efficient Fibonacci heap however this is likely a python specific phenomenon. Which mainly occurs for Dijkstra since more heappush() operations are performed in Dijkstra which are more efficient using heapq than a fib heap. A good article illustrating the differences:

<https://bit.ly/2RgRHlF>

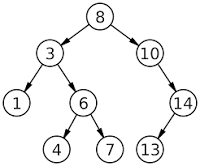
A priority queue is a data structure that follows a first in first out principle (FIFO). This means that the first item to enter the queue is the first to leave. A priority queue is a more specific version of a queue data structure where every element in the queue is assigned a weighting and the element associated with the lowest weighting is released from the queue first.

For example:

A queue may look like this in order:

((1, “Jim”), (2, Sarah)ray, (15, Michael), (inf, Echo))

The first item in each tuple of this list is the weighting and the second is the value associated with that weighting. When popping off this list the first item returned is the item with the lowest weighting so in this case (1, “Jim”). This is known as a min heap (or queue). A max heap returns items in the opposite order from highest weighting to lowest.

This is particularly useful for Dijkstra since this would give the algorithm the shortest path from a node first increasing runtime efficiency. It is also useful for ray casting since rays will only be cast onto the closest walls first then the walls that are further away and would cast onto walls that are within a reasonable distance that are worth checking for an intersection. For ray casting in particular, a heap increased efficiency by a very large proportion; originally (for a 200 by 200 maze of walls) the time it took for 360 rays in a full circle from an enemy to be calculated took around 16 seconds. After utilising the heap, the calculation for all 360 rays took 0.2 seconds which is 80 times faster (disclaimer: a 200 \* 200 maze will likely never be used in my game since it is simply too large a maze to be playable or fun).

A heap is an implementation of a priority queue that stores items in a traversable binary search tree used for sorting. The way this works can be visualised as shown in the image.

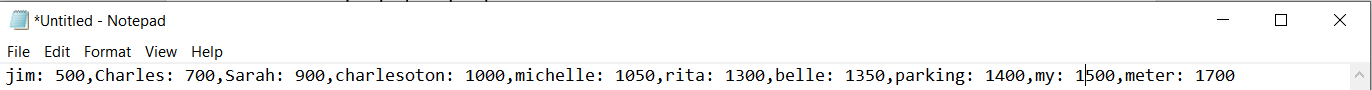
When storing items into, or pushing items onto the heap, the heap will store the first item as the root node of the tree. Assume this item has a weighting of 8 as shown above. The left and right nodes (or children) of any node (or root) in the tree will store an item smaller than the root and larger than the root respectively. So as before we only have one node in the tree, with weighting 8. Suppose we wish to add a node to the tree with a weighting of 10. 10 > 8 therefore the node with weighting of 10 will be the right child node of the root node with a weighting of 8. Now we wish to add a node with weighting 3. 3 < 8 therefore we would add this node as the left child node of the node with a weighting of 8. Any root node can only have 2 children therefore if we add a node with a weighting of 6 (6 < 8) would go to the left child node. Then we check the left child node against the node we are trying to push onto the heap. Since 6 > 3 the node with a weighting would be put onto the right child node of the node with a weighting of three. This is done for all nodes that are being pushed onto the heap.

A heap is an efficient implementation of a priority queue due to the way it sorts items in the heap. This improves efficiency since sorting is very efficient, all you have to do is place nodes into right or left child nodes of the root nodes which is much more efficient compared to a sort algorithm such as merge sort for searching the array (heaps don’t actually have to sort the items in the list). As a result, to find the smallest node all you must do is traverse to the left most root node. This is known as in order traversal. This has an O(nlog(n)) efficiency when searching for the smallest element. If we had to sort the list first this would take O(nlog(n)) efficiency as well however sorting must be done each time an item is added to the list which would make it not very useful for Dijkstra.

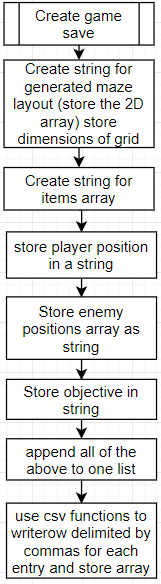
## 2.4 Storage of data

**Leader board storage:**

Data in the leader board will be stored in csv format. This is done because there will be no need to create a database for the leader board since there will only be two data entries stored in it. These two pieces of data are username and score.

In csv format these would be store like so:

Each entry is delimeted by a coma and the colon is used to help the program identify where a score starts and username ends.

**Save file storage:**

## 2.4 Core algorithms

**Maze generation algorithm:**

The most important algorithm in my program is the maze generation algorithm. This is what the game is built on top of and if it doesn’t meet my requirements than my project isn’t complete. As a reminder the main objective for the maze generation is that it is created procedurally (randomly). The maze must also be beatable so must have loops so that players can avoid enemies. The maze will be created as a 2D array with each item in the array being another array containing 1 or 0.

1 means wall

0 means open path

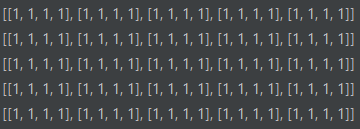
A 2 by 2 grid of completely blocked off nodes would look something like this in array form:

[ [[1,1,1,1], [1,1,1,1]],

[[1,1,1,1], [1,1,1,1]] ]

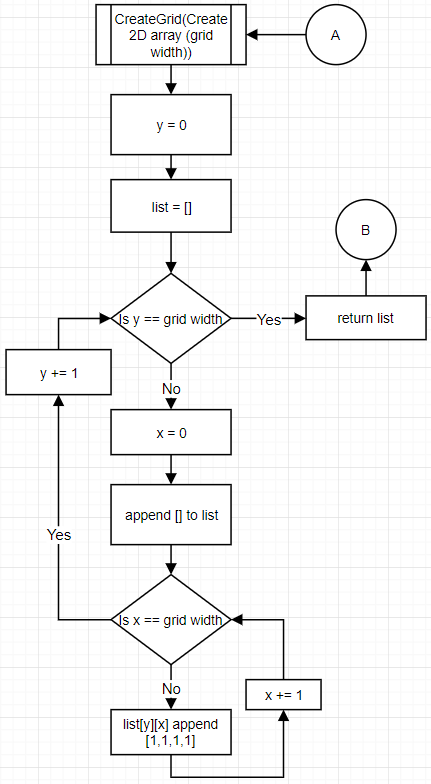
Each index in the list would correspond to North West South East in order or NWSE

A 5\*5 grid would look like this (with same conditions as before)



A grid like this is initialized so that the maze algorithm can work with to create a maze

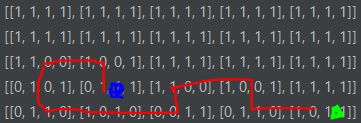
**Flowchart for grid creation:**



The maze generation algorithm will use multiple grids as shown above known as layers. These grids will be worked on and a path from a start node to the exit node will be created. These layers are overlapped onto the actual maze to create a full maze with paths leading from every node to another node.

**Layers:**

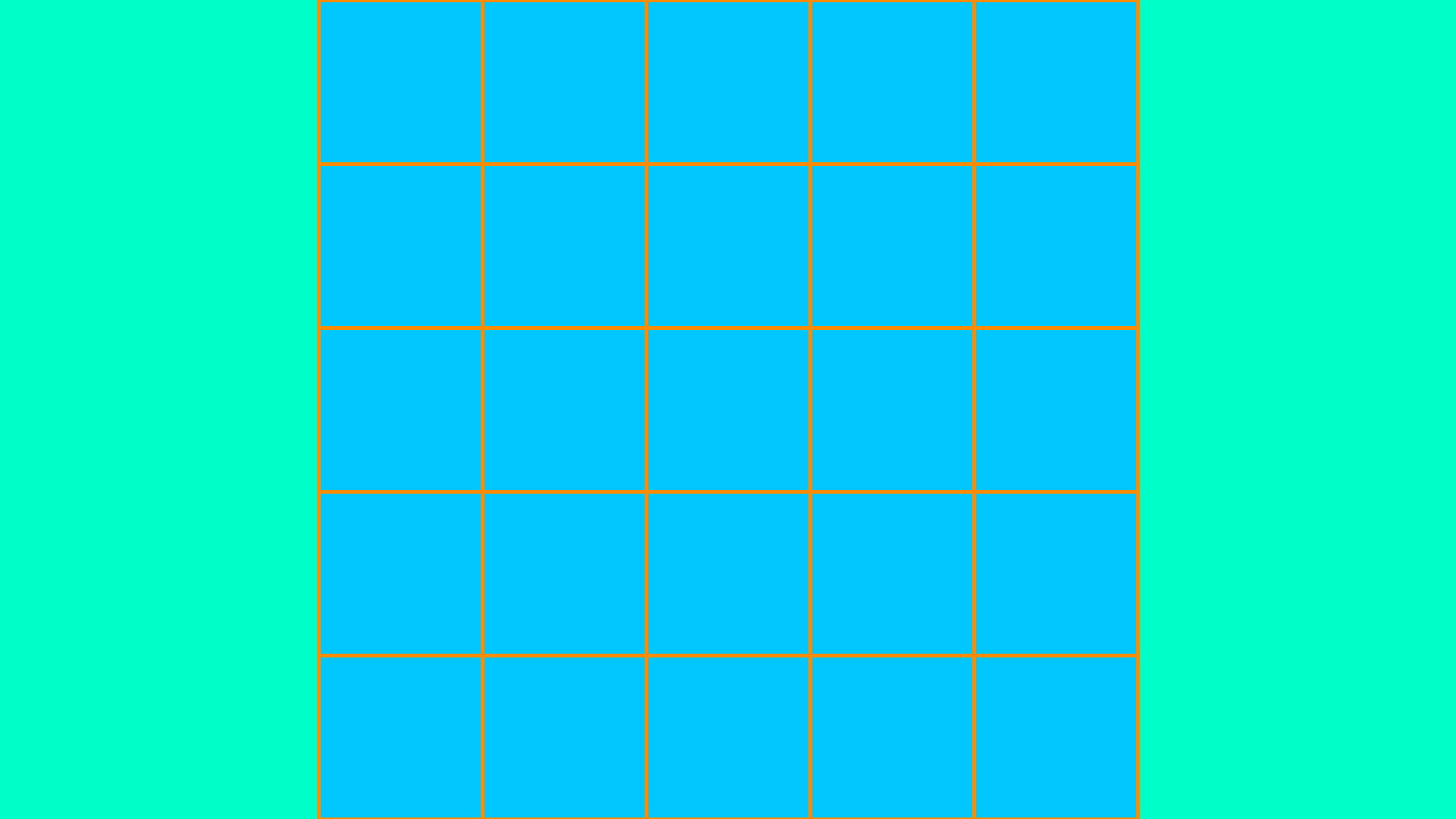
A layer is a grid of the same size as the actual maze. The layer will have a single path leading from a node to the exit node. A layer for a 5\*5 grid will look like this:



Blue is the start node and green is the exit node. Red is the path the algorithm took for this layer. As is shown above the path from the first node to the next is northward. This means that the 1st index of that positions array is a 0 indicating a path north. Also, the 3rd index of the next node is 0 indicating a path south. This connects the two nodes and every node from start to finish is connected in the same way.

**Snakes to create layer paths:**

The best way to visualize a path creation is by using the imagery of a very strong snake. This snake is infinitely long. It is also capable of breaking through walls and it is trapped in the maze of nodes that are all blocked off. At the start it is only in one node. As in the array above, it starts as shown: (the spiral being the snake)



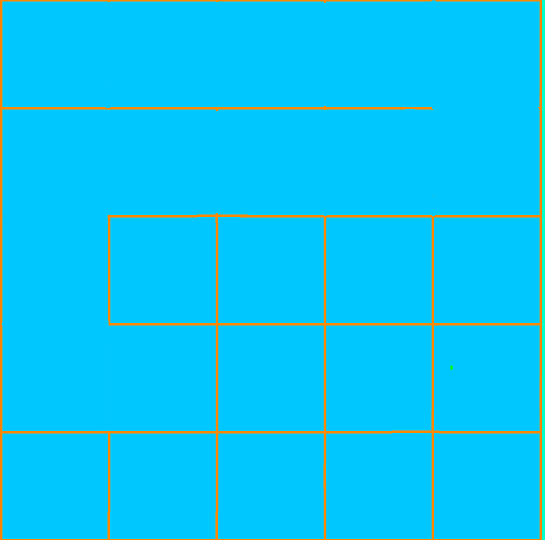
The snake is confined by a set of rules.

1. The snake cannot know where the exit is and therefore travels in random directions
2. The directions it chooses must not be to anywhere it has been before otherwise the snake will overlap itself and get tied up in a knot
3. The snake can backtrack but must rebuild any walls it has broken
4. Once the snake reaches the exit position the snake should not travel further.
5. Every position it visits must be recorded and if the position is backtracked away from the position must be removed from the record. (this is the history stack)

Say the snake decides to travel:

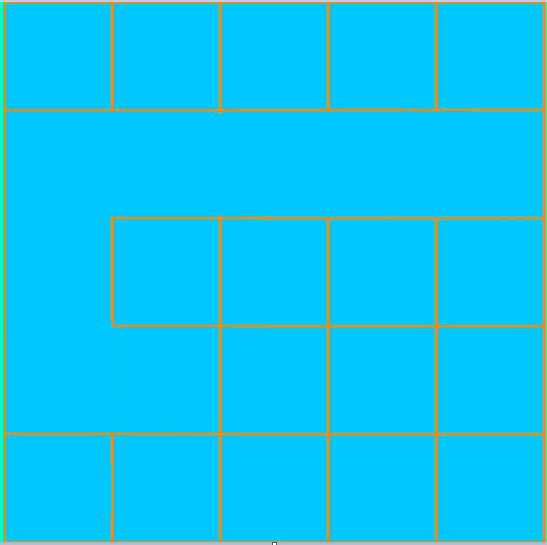
(West, North\*2, East\*4, North, West\*4) with an exit position of the bottom right of the maze.

The path will look something like this:

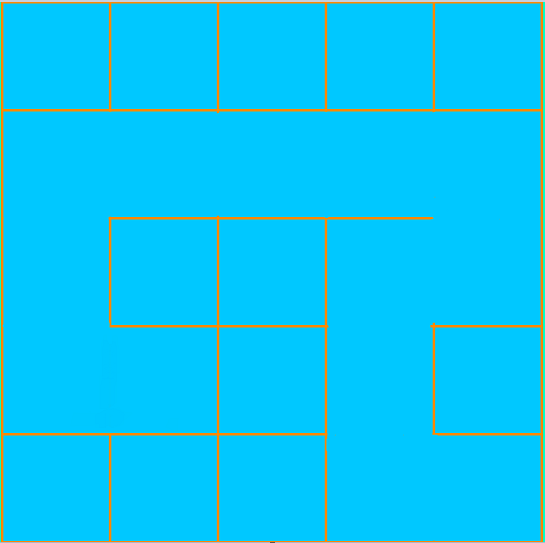


Clearly there is a problem; the snake can no longer travel anywhere other than where it has been before or it will be caught in a knot. So, the snake backtracks until there is a path it can take. In terms of data on the computer the history stack will be used to identify previous locations in order and once the snake backtracks successfully the location in the history stack (stored as an array of x and y co-ordinate) will be popped off striking it from the record.

After the snake backtracks it ends up here.



The snake is programmed to be somewhat intelligent so the first path it takes after it backtracks will be anything but in the direction that got the snake stuck. In this case that is south.

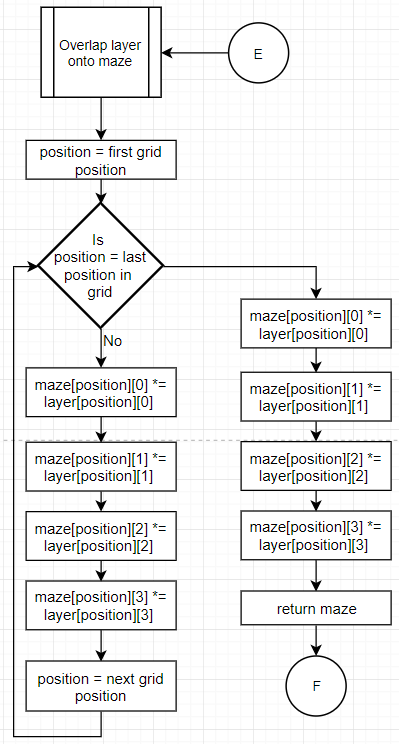
 So, let’s say the snake goes:

South, West, South\*2, East

The grid will look like this at the end and as a result we have a completed layer. This layer is overlapped onto the maze as shown in the next flowchart.

After the layer is overlapped onto the maze the layer is refreshed as a new grid and the maze generator creates a list of unvisitable positions in the grid (by storing positions of nodes with value [1,1,1,1] in the maze). A random node from this list is chosen as the new start position for the next layer to be created. This process is repeated until there are no more blocked off nodes in the maze array.

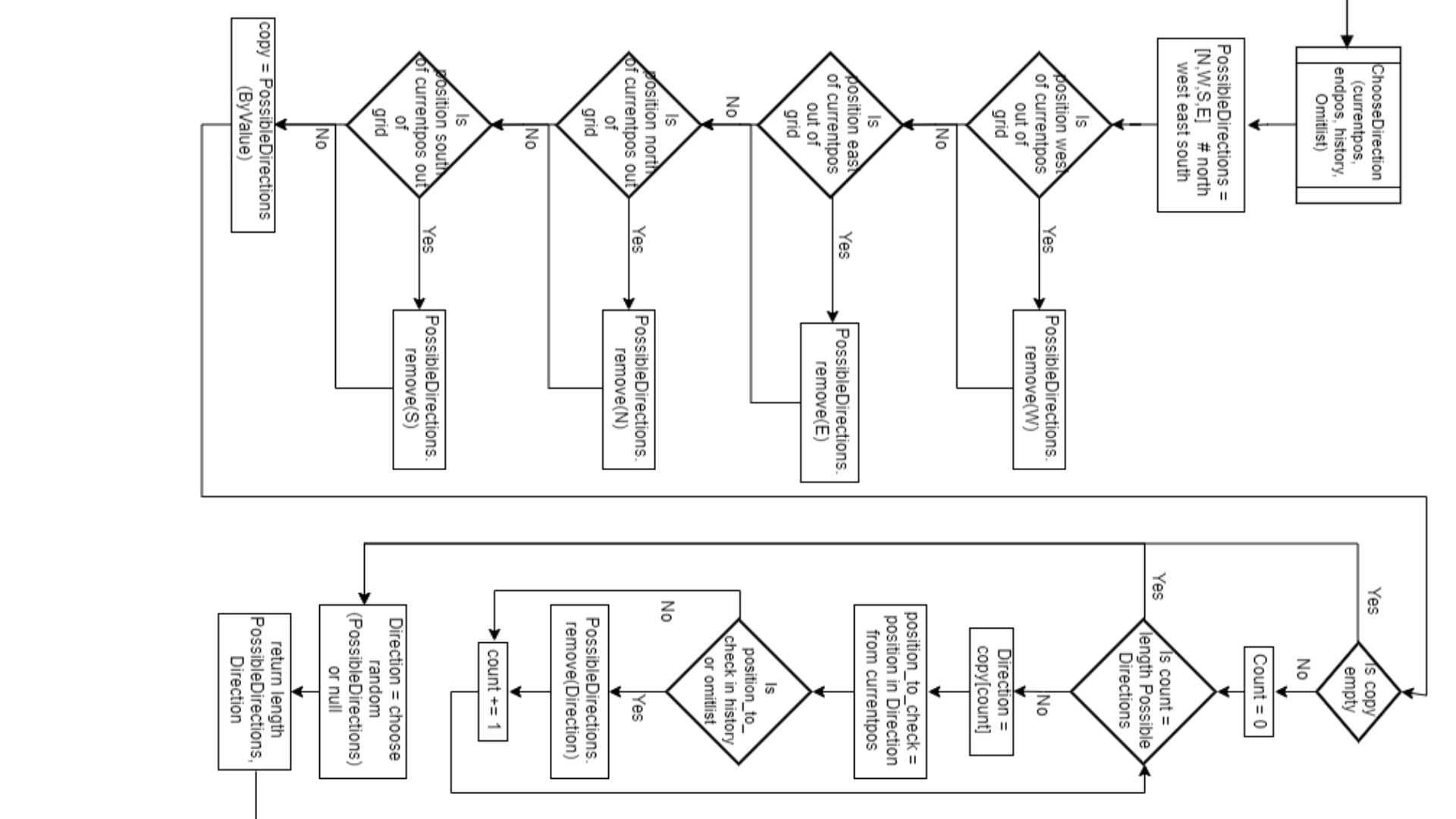
**Overlap layer onto maze:**

This maze overlapping works in a similar way to a bitwise AND operation. Every open path is layered above a wall and overrides the wall creating a path on the actual maze.

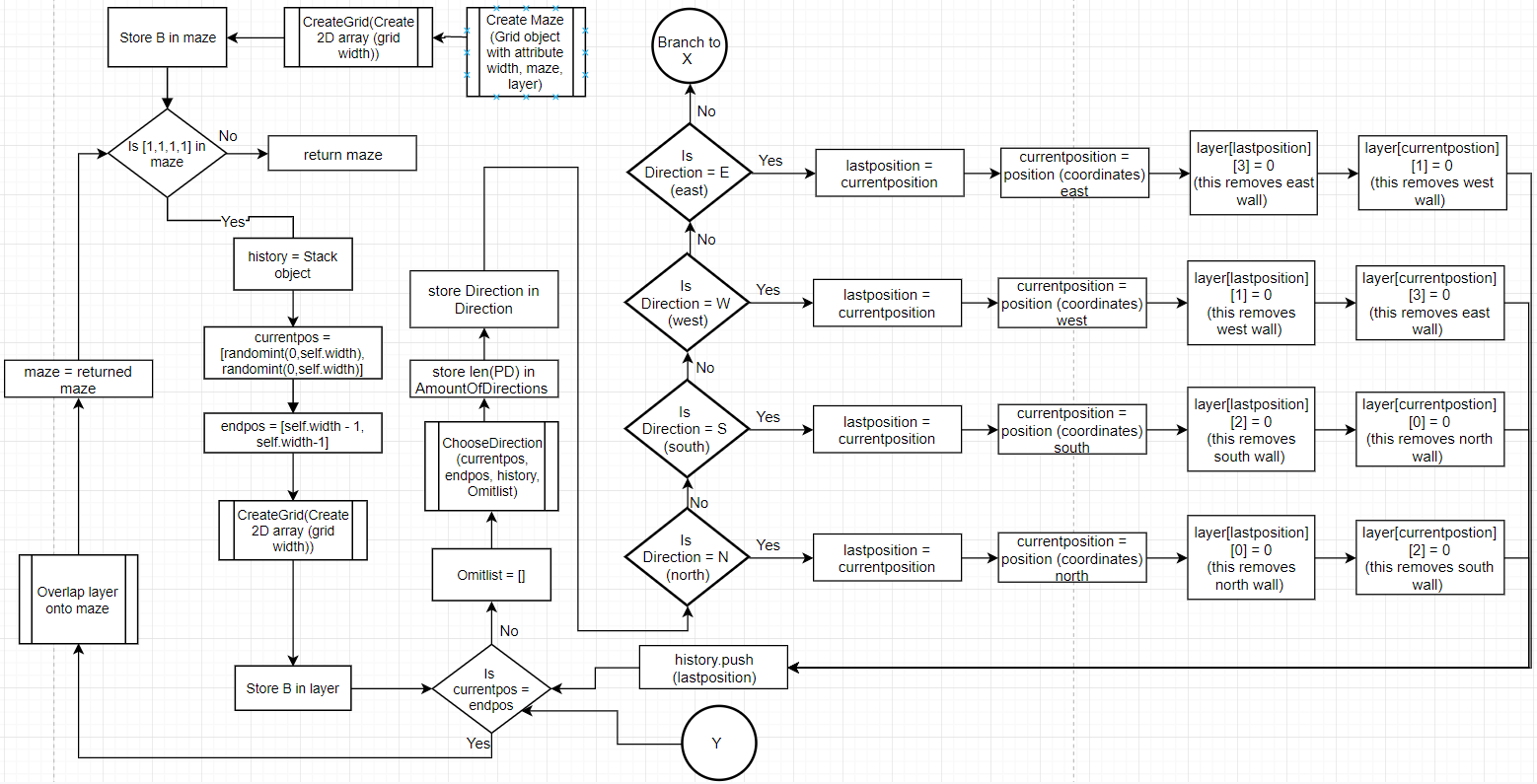
**Notes on layer optimization:**

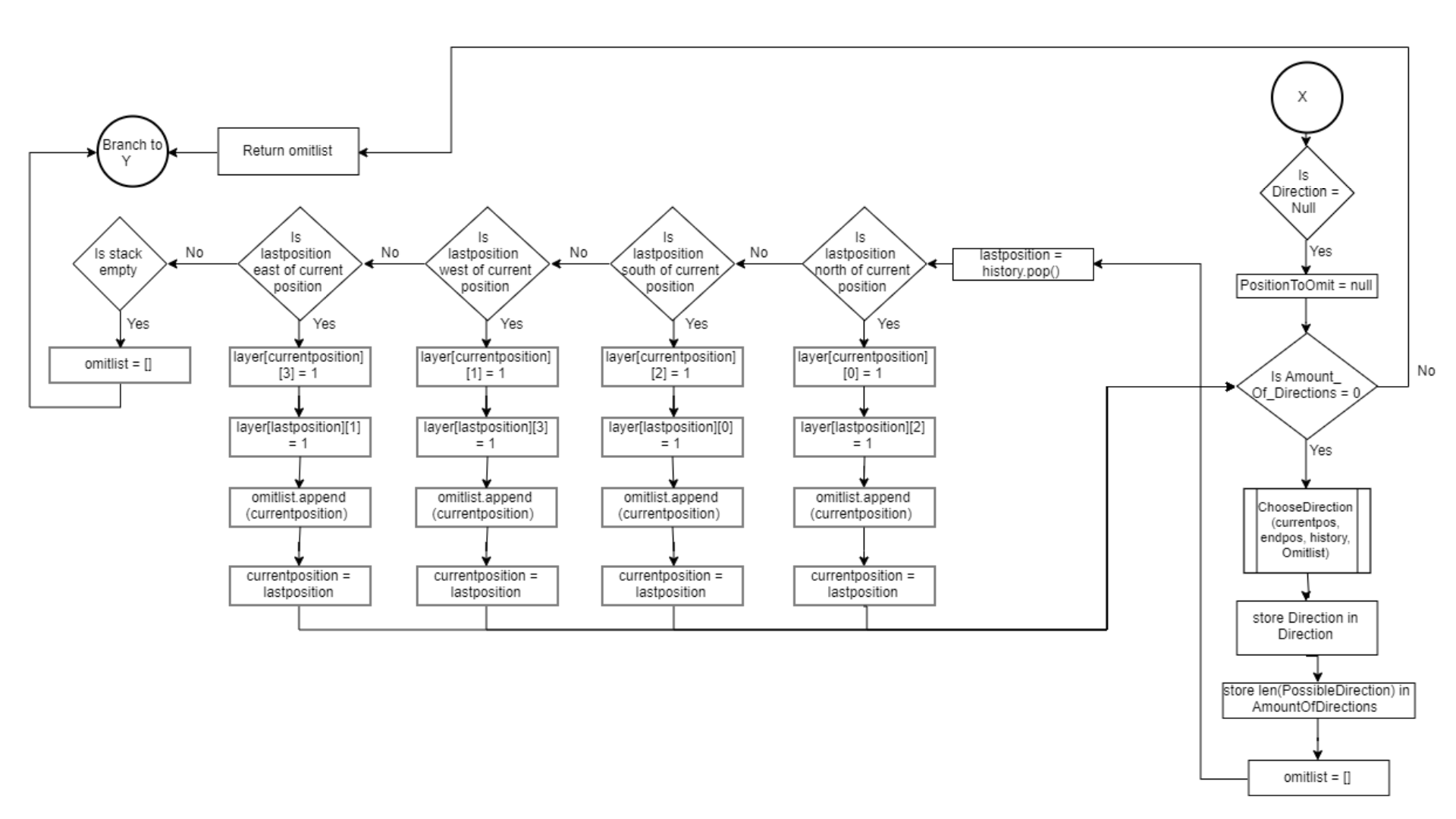
Recently there have been efficiency improvements to this algorithm. Instead of creating a new width\*width grid for each layer, it is much faster to store a template of a fresh layer and copy that onto layers. This reduces the amount of time it takes to generate the maze overall especially for larger mazes with many layers. This template will be stored as an attribute of the grid class called template.

**Choose Direction algorithm:**

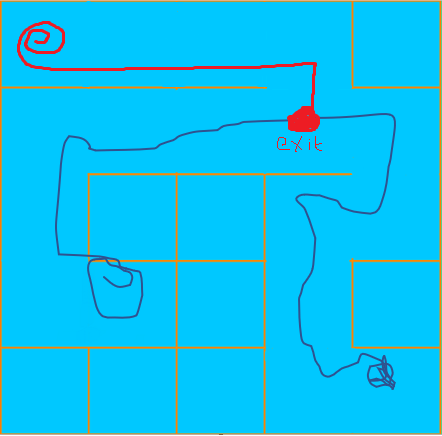


**Full maze generation as described before:**





**Efficiency improvements and maze sparsity improvements:**

An efficiency improvement has been made since creating the algorithm above. Now instead of only allowing a snake (as described before) to exit the maze once it reaches a specified exit point, now the snake will also exit if it hits a point in the maze that has already been visited once or twice (depending on how sparse you want the maze to be). This means that as shown before a second snake will create a path that looks something like this (blue represents first snake red represents second). 

This new snake moves in random directions from its start point until it reaches a previously visited point on the maze by another snake (in this case a point which the blue snake has visited). This however would create a perfect maze which isn’t desirable since when the game is played it would be impossible to pass an enemy blocking the path to the exit by taking a different path (a perfect maze is a maze with no loops (only one path from a node to any given node)). To remedy this issue, we can manipulate this rule a bit to say that the snake can exit at a point where at least two snakes have been before.

**Line of sight algorithm:**

My line of sight algorithm uses a technique known as ray casting. It works by mathematically describing line segments from the observer (in this case the enemy) to its environment. The ray is used in a similar way to a sensor. Using mathematical calculations that utilize matrix calculations, all intersections between walls and the ray can be calculated to exist or not and the ray can also calculate the exact point of intersection. This can be used to generate 3D graphics (which could be a consideration of this project). It may also be used to calculate whether the closest intersection is the player and whether to set an enemy as having detected the player or not and at what location.

The mathematics behind calculating whether an intersection exists uses matrix mathematics. A summary of this calculation is shown below:

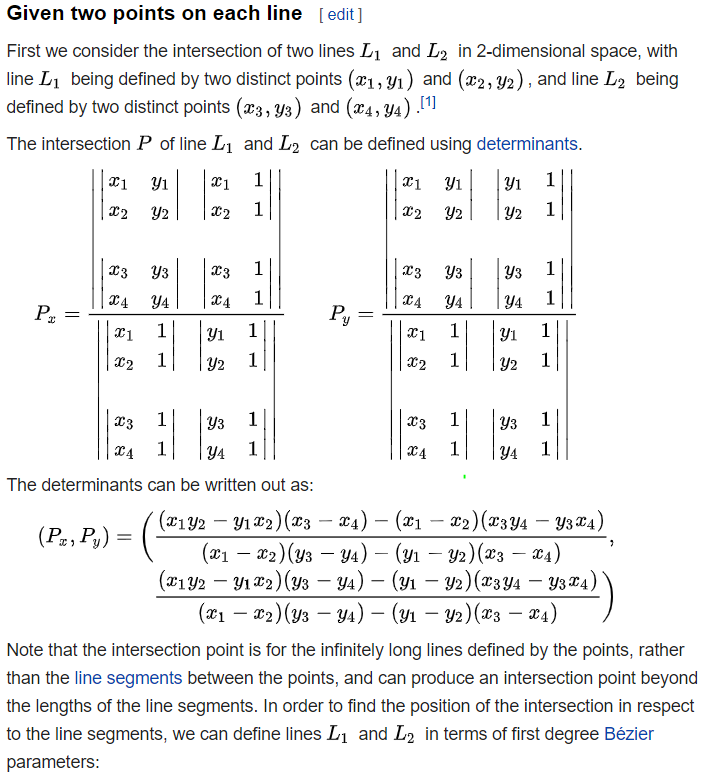
Start point of ray = (x1, y1)

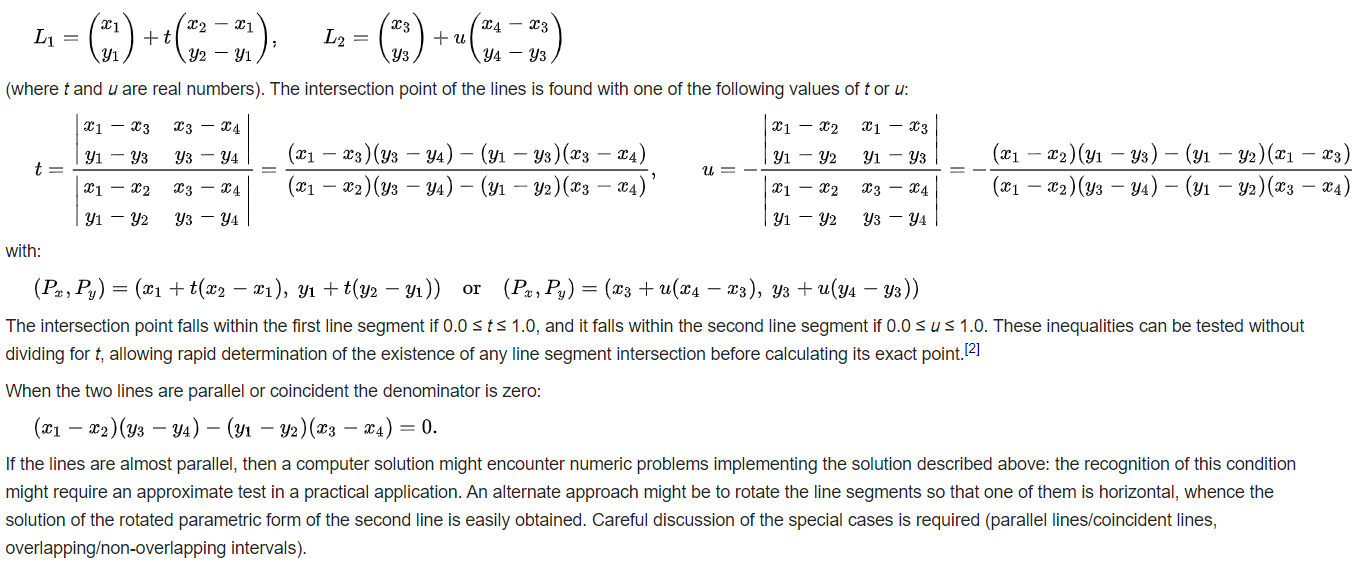
End point of ray = (x2, y2)

Start point of wall = (x3, y3)

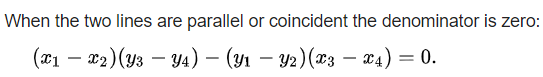
End point of wall = (x4, y4)

An extract from [Line–line intersection - Wikipedia](https://en.wikipedia.org/wiki/Line%E2%80%93line_intersection#cite_note-GGIII-2) or <https://bit.ly/2p7F8xu> summarizes this calculation pretty well:



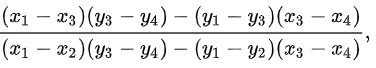


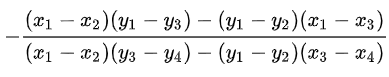
The main parts of this algorithm we are concerned with is:



This calculation allows us to determine whether there is even an intersection between the ray and the wall. This uses the denominator of the values used to calculate point of intersection. If this equation is true, there is no possible intersection between the ray and the wall. Otherwise, there may be an intersection, but this may only be relative to the cartesian equation of the lines and it may not be within the bounds we are concerned with (so it may be beyond the end of the wall)

To work out whether the intersection is within our bounds we use the following values of t and u:

t =



u =

Notice how the denominators are equal and the first calculation at the top of this page. If 0<= t <=1 and 0<= u <=1 then there is a point of intersection within our bounds

Now to work out the point we use this formula:



Or this formula:



Where Px and Py are the coordinates of the point of intersection (the notation relating to matrix multiplication which is beyond the scope of this project since it is used for 3D rendering).

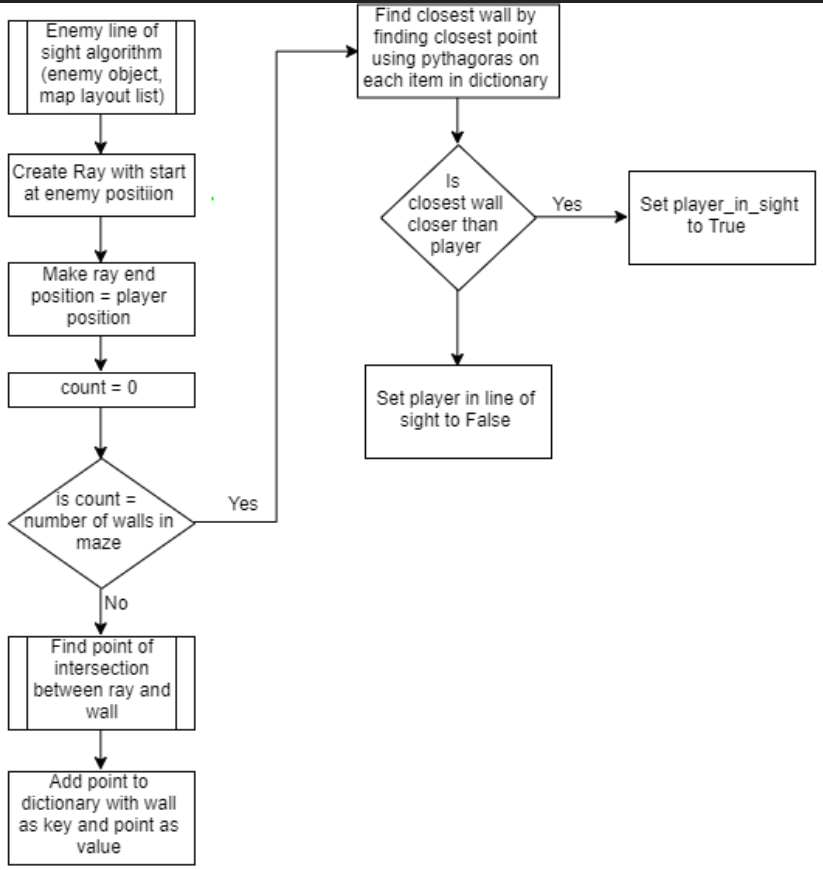
This way of calculation is efficient since it doesn’t attempt to calculate the point before it determines the actual point. Ray casting is a technique used in the 1981 Wolfenstein for 3D rendering on weak machines of 4 MHz clock speeds.

Credit for this goes to Antonio, Franklin (1992). "Chapter IV.6: Faster Line Segment Intersection"

Flowchart for the algorithm used for line of sight calculating and rendering to screen:

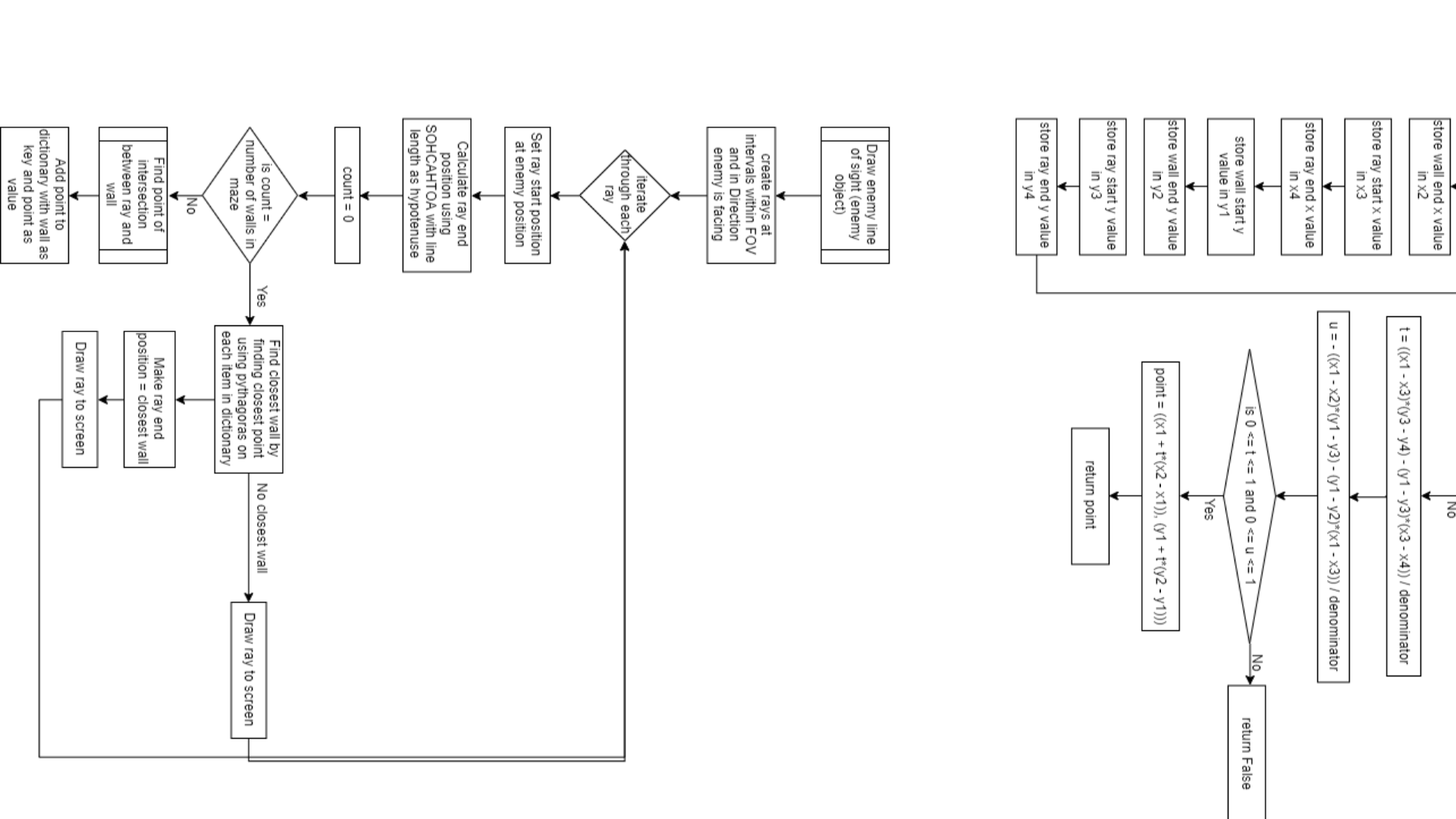
**Calculating player in line of sight:**

A single ray is used for efficiency since no other rays are needed (except for rendering to screen).



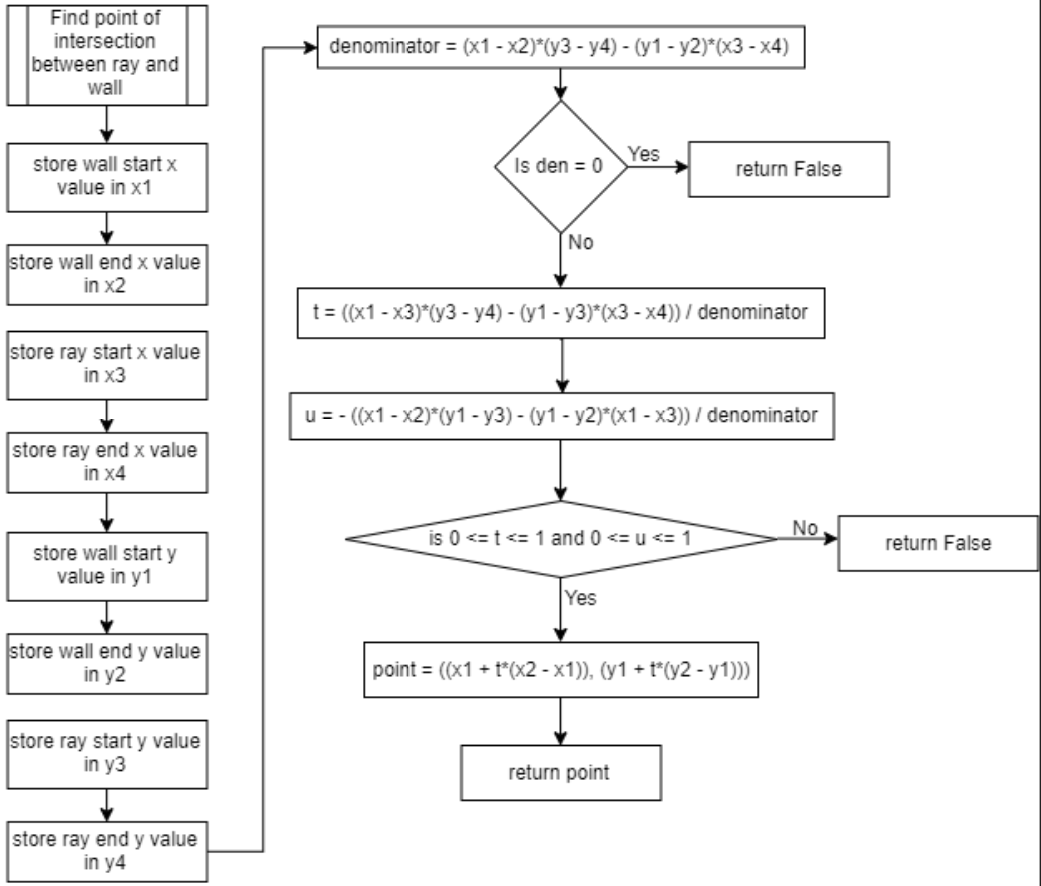
This is only for determining whether the player is spotted or not.

**Render line of sight onto screen:**



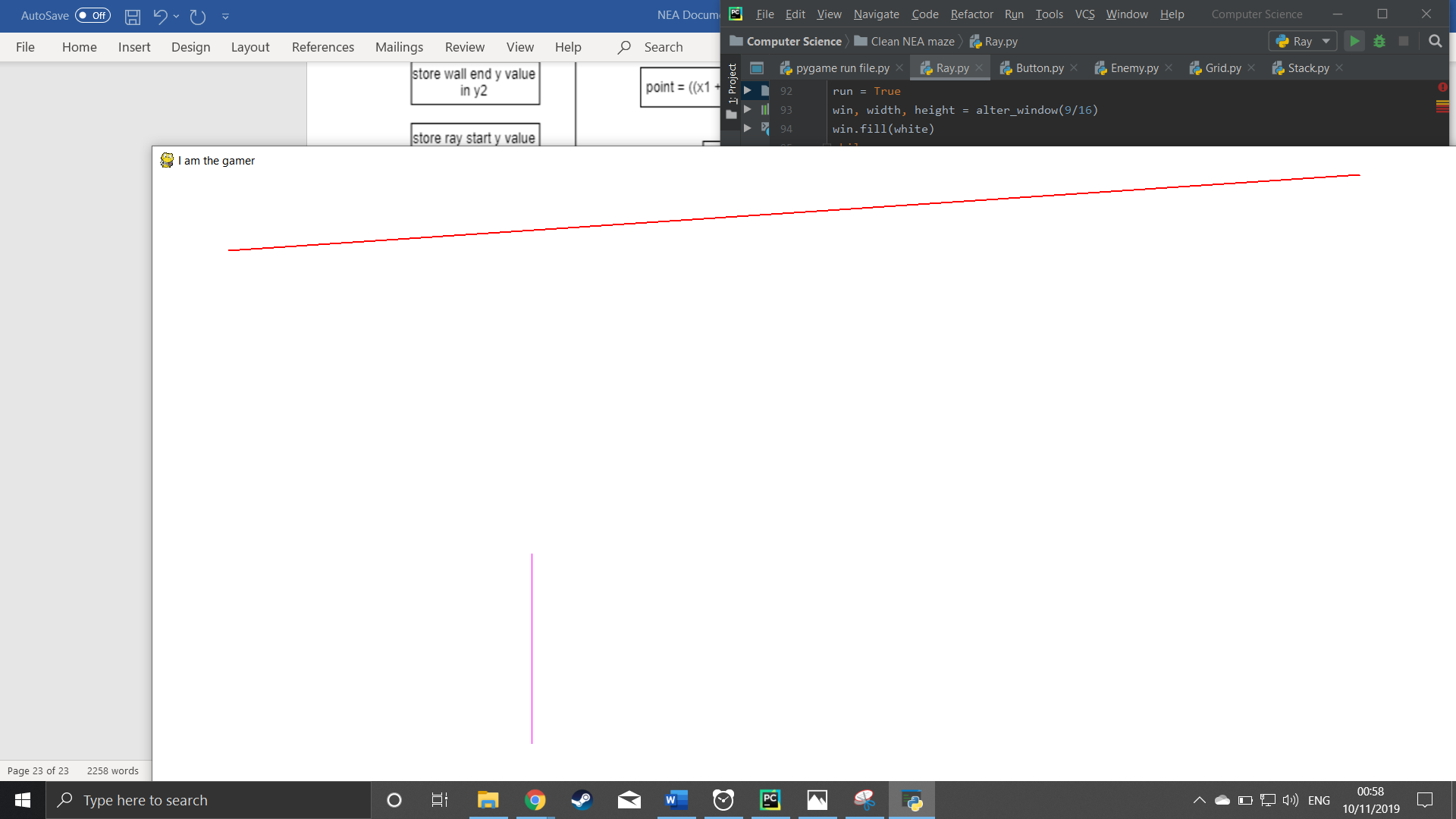
An enemy will have many rays that are used to render the line of sight so that the player can visualise the enemy’s range.

**Mathematical calculation for intersection algorithm (as described) that each ray will use:**

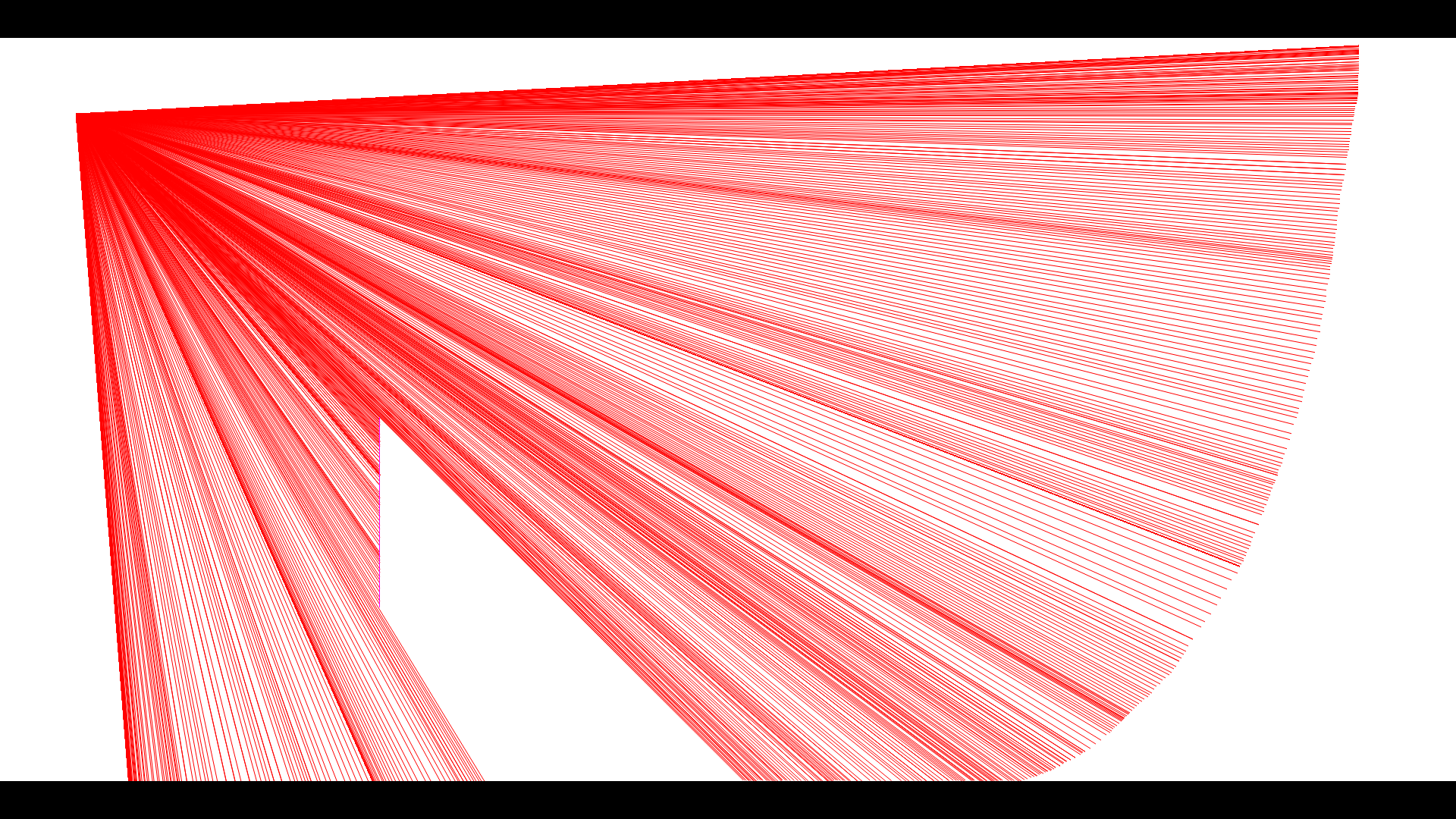


**Ray prototype images:**

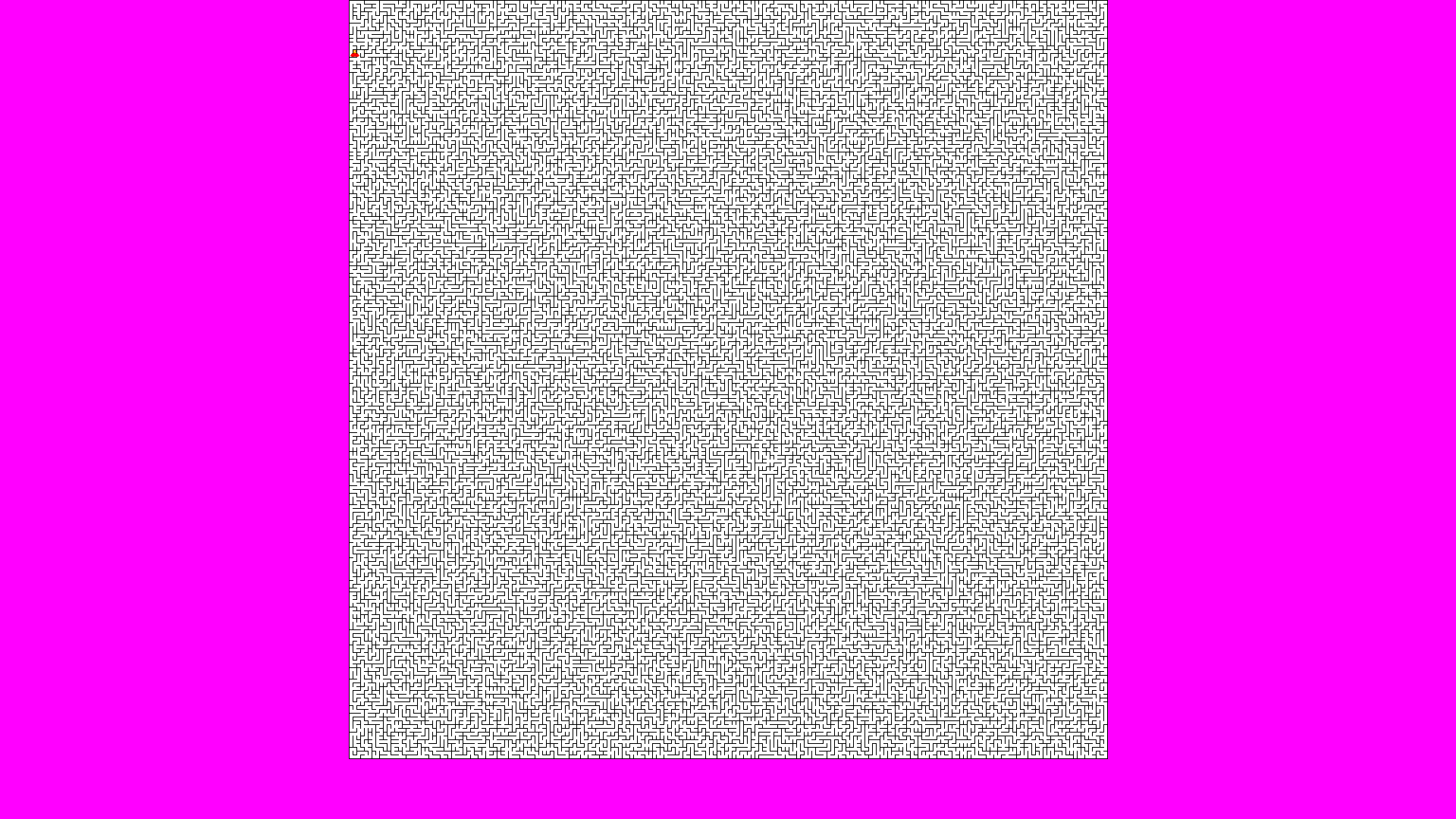
I have created a prototype for this algorithm to use as a proof of concept. The prototype only contains one wall and a ray that follows the mouse. Rays that are drawn to the screen are not removed so that the way the code works can be visualised better.



The red lines are the rays and the purple line is the wall. After sweeping behind the wall using my mouse a shadow is cast as shown below. This may be useful for creating shadows and perhaps even a working fog of war. It takes 0.23440074920654297 seconds to calculate 720 rays from 2 separate start points in a 20\*20 grid with many walls. Since a simple mathematical calculation is used the efficiency of the code is very large. If there are issues with efficiency then a heap can be used to store walls by distance from ray start-point and only walls that are close enough are used to calculate the rays endpoint. However this is a consideration that may be implemented if there is enough time left to do so.

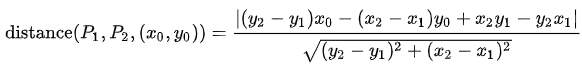


However, there is a problem here. Consider a 200 \* 200 maze. If we draw rays from an enemy in any given position (say 360 one for each angle in a 360 degree FOV) and a ray has say a distance of 15 blocks then we have a fatal flaw here. Every ray must be checked for intersection with every possible existing wall in the maze whether it is more than 15 blocks away from the wall or not.



1 A perfect maze (mathematically speaking generated by algorithm)

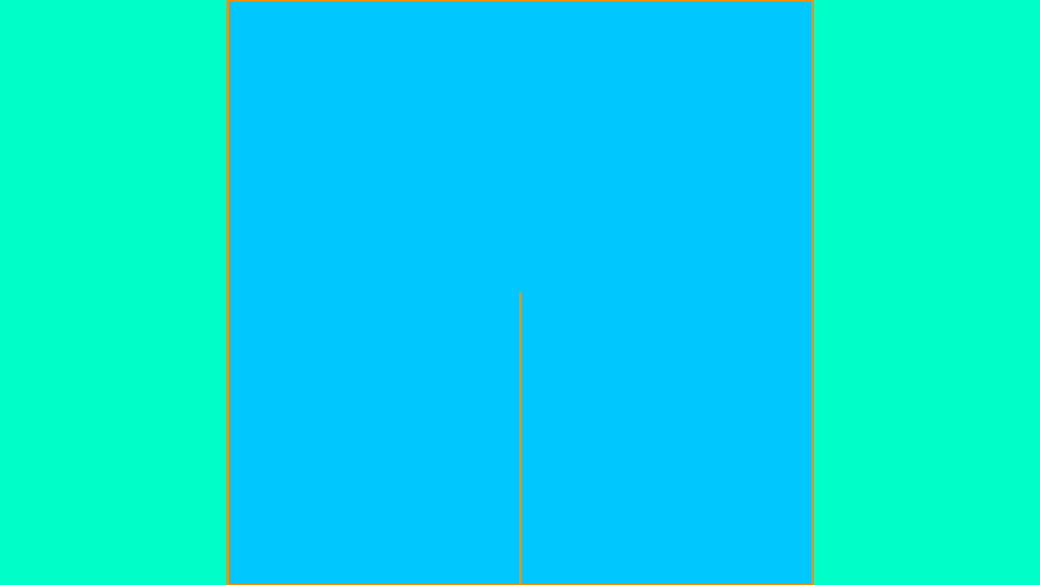
Through testing on a 200\*200 maze (using pythons time.time() to calculate the amount of time it takes to calculate 360 degrees of rays) it takes around 16 seconds for all ray intersections to be calculated. As a result, the program becomes highly inefficient and very slow. A solution to this problem is the utilisation of a heap that stores only walls that are within a reasonable distance from the start position of a ray. The white blob is the enemy’s rays drawn onto the grid. Using this method, it took 0.2 seconds to calculate all 360 rays.

A heap is explained previously in this documentation however the tl;dr of it is that it is a priority queue that returns an element with the smallest weighting. In this case the weighting is the smallest distance between a ray and a wall. This is calculated using the following formula ((x0, y0) is the rays start position and P2 and P1 is the line defined by its start and end points).

**Path finding algorithm:**

For my path finding algorithm to work a graph data structure needs to be implemented. A graph is a data structure that consists of nodes and edges. A node in my implementation would be a position in a maze. A path between that node and the adjacent nodes will be the edges. Enemies need to be able to find the shortest path from one node to another in order to successfully ambush the player.

The graph doesn’t need to do much other than be used to find shortest path. An adjacency matrix will be used instead of a list because, although an adjacency matrix may be memory inefficient, the amount of memory it will use is insignificant for the sizes of maze I am using. Matrices will reduce the time complexity of my code compared to adjacency lists since looking up a connection between two nodes is as quick as requesting data from a location in memory without need for calculation.

A 2\*2 maze would look like the chart below in memory as a graph represented by an adjacency matrix.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| positions | (0,0) | (0,1) | (1,0) | (1,1) |
| (0,0) |  | 1 |  |  |
| (0,1) | 1 |  |  | 1 |
| (1,0) |  |  |  | 1 |
| (1,1) |  | 1 | 1 |  |

Every 1 in a box represents a connection with weight of 1 between the positions in the rows and columns. My game will have different terrain types that will be used to set weights to the edges.

I already have a rough implementation of this. Once a maze is created, an adjacency matrix should be calculated immediately and overwrite the previous matrix. This is to prevent the AI using an old matrix to calculate its movement path.

A 50\*50 maze was tested to see how much time it would take to create the matrix and the times that were calculated averaged at around ½ a second. This isn’t much time at all especially because a matrix is only calculated once after a maze is generated and never again.

However, a 50\*50 maze uses a lot of space as it is a 2500\*2500 2D array in matrix form. Furthermore, most of this space is redundant as an undirected maze is symmetrical diagonally so there are repeated values. Also, there are very few connections between nodes. However, the space used is irrelevant compared to a (on average) 4GB stick of RAM. In future an adjacency list may be used so that the game can be played on much slower, smaller (in terms of main memory) devices.

**Dijkstra / A star efficient path finding:**

My enemy AI will use A star search which is an extension of Dijkstra. Dijkstra’s algorithm is an algorithm that searches for the shortest path between two nodes (or locations on my maze). The algorithm needs to be able to handle unseen mazes since mazes are procedurally generated. Dijkstra will also be used to show the user the path they will take before they choose to move by drawing a line that follows the path given by Dijkstra. It should be noted that diagonal movement between two vertices (nodes or locations on a maze) is not possible.

Firstly, a matrix must be calculated along with the maze

**Matrix creation algorithm:**

As said before, an adjacency is required for Dijkstra to be implemented. This is a data structure that allows me to store connections between any two positions in the maze. If there is no connection this will be stored as a connection with an infinite weighting (using pythons float(“int”)).

For this algorithm what needs to be understood is the way connections are illustrated in the maze at first. A connection between a node and another node depends on their location and whether there is a wall between them or not.

This is simple to figure out:



In the maze array shown above nodes are elements in the list. Each element is an array that stores connections North, West, South and East in that order (NWSE). 1 means there is a wall and 0 means it is open. In the example above the first node has a connection south and east. Therefore, for this node, the matrix would store a connection between the node and the neighbouring south and east node. For this entire maze (2\*2) the matrix will look something like this each index representing a node.

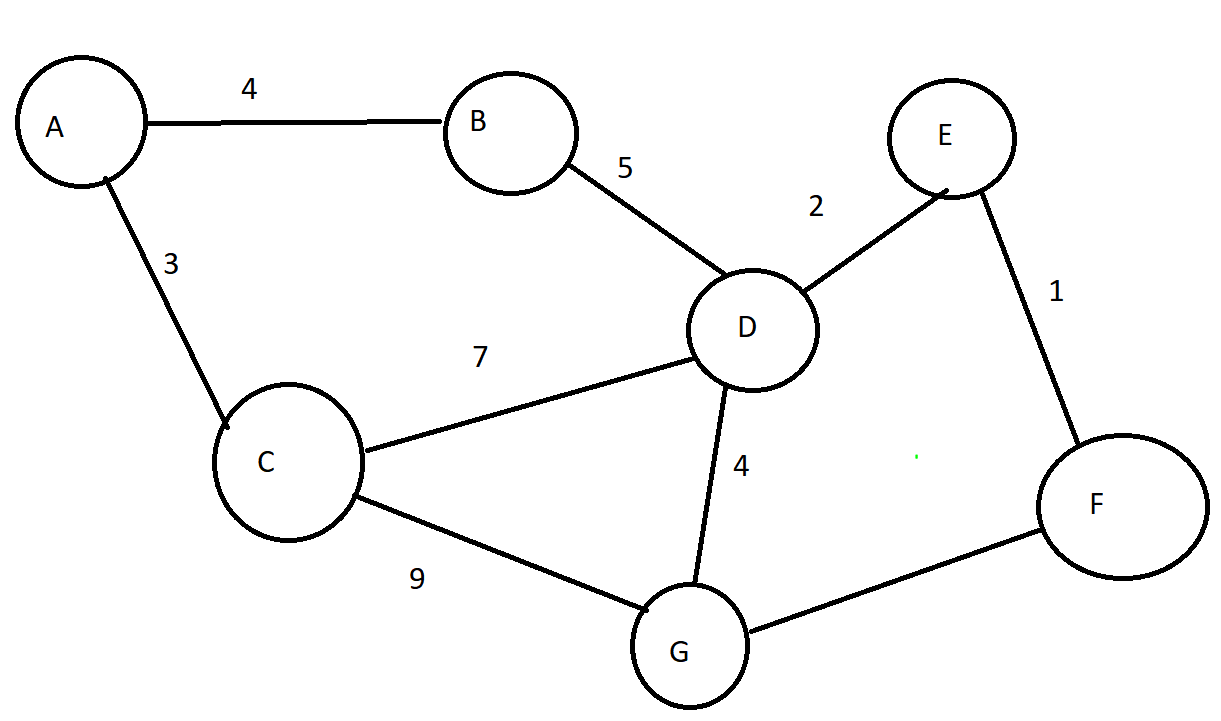
**Maze as node indexes: Maze corresponding matrix:**

|  |  |
| --- | --- |
| **0** | **1** |
| **2** | **3** |

As said a matrix (graph representation of the maze) is created alongside the maze

**Dijkstra path finding algorithm:**

Say you are given the following graph (edge weights are numbered next to lines) nodes labelled as circles



Your algorithm is tasked with finding the shortest path from A to F

Dijkstra tackles this by creating a priority queue containing every node alongside a source set as None and a weight from that source initially set as infinity. Except for the source node which has a weight of 0. This looks something like what is shown below:

|  |  |  |
| --- | --- | --- |
| source | node | Weight |
| None | A | 0 |

|  |  |  |
| --- | --- | --- |
| Source | Node | weight |
| None | B | infinite |

Etc etc…

All of these sets are stored in a priority queue ordered by weighting. The first node is taken from the priority queue (in this case A). And added to a visited queue

The neighbours of this node are all evaluated 1 by 1 with their weighting and source node updated to match the weighting from the source node to them. These are added to the visited queue.

Then the node with the next lowest weighting is taken from the visited queue to have its neighbours evaluated. The source node and weighting is evaluated the same way. If the neighbouring node being evaluated has a weighting that is higher than the new weighting, then the value is updated otherwise the value is not updated and kept the same.

This process is repeated all the way until the algorithm first reaches the exit node (F).

Now we have a visited queue containing all nodes that the algorithm has visited along with the node they came from and the weighting from that node.

So to create the path from this visited queue the algorithm has to go from the exit node backwards.

The exit node is removed from the queue and the weighting is added to a running total and the node is added to a list of nodes.

The source of this exit node is checked and removed from the visited queue and the process is repeated until the algorithm reaches the source node

At this point there is now a list of nodes in reverse order going from the exit node to the source node and we have the weighting of that entire path.

All that is left to do is to return the entire list in reverse and the weighting

And that was the Dijkstra algorithm.

This Dijkstra algorithm is used for player and enemy movement and so is crucial to this project