# **Wave Based Ray Casting Game Analysis**

## Project Introduction:

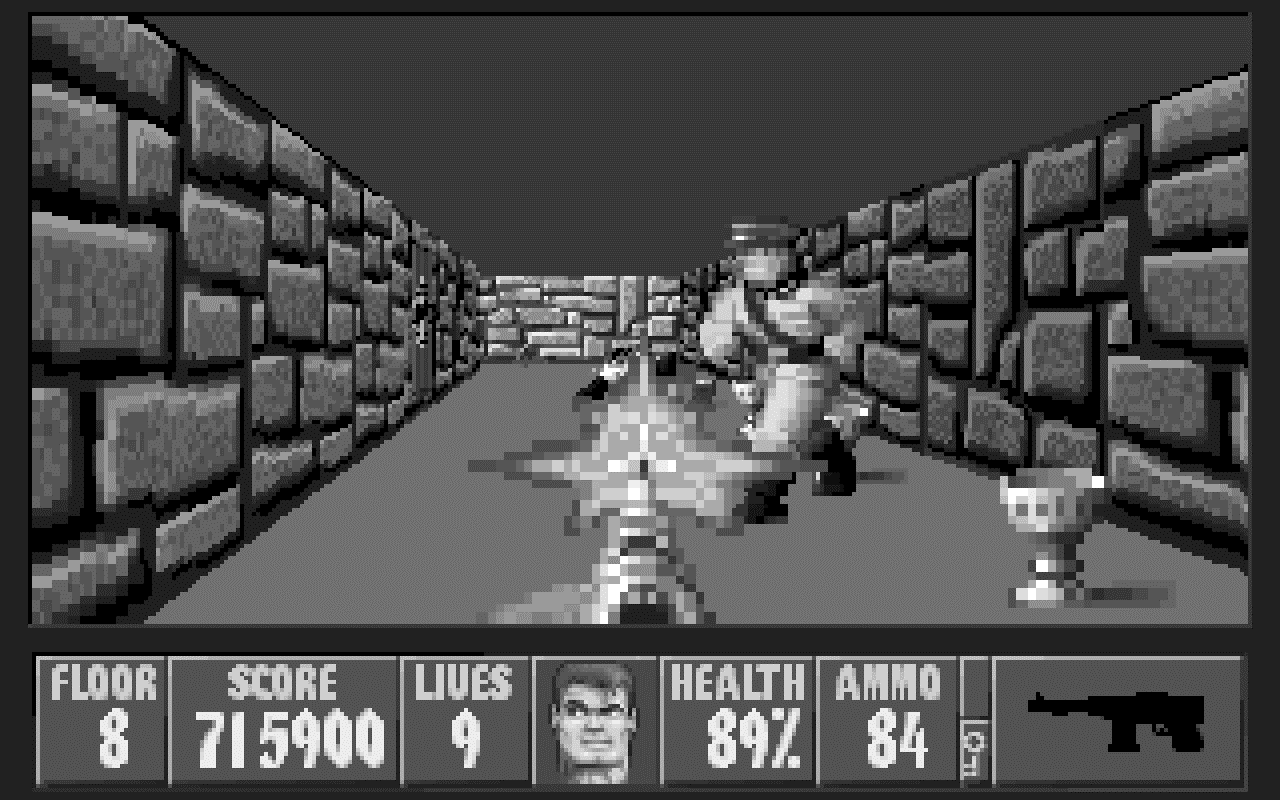
In the modern day, Gaming is a giant in the field of entertainment. Video Games can come in many different forms, with a seemingly infinite amount of detailing available, along with a myriad of content possibilities. However, Video Games also need to have a graphical representation for the user. Many methods and algorithms are available and lots of consideration is required to decide the best type of implementation for such a Video Game. One such option to be considered is Ray Casting. Ray Casting today is considered somewhat primitive in the world Video Game development. However, has many benefits over the modern standard. It bases its simplicity of a 2-Dimensional environment which can then be converted with Ray Casting to be given a 3-Dimensional representation. A sort of “Fake” 3D.

## Problem Definition

My goal is to create a wave-based zombie shooter, which will use Artificial Intelligence and Ray Casting to create a 3-Dimensional look. The game will consist of a main map, which is a large pre-generated grid. It will also use 2-D Chunk Based [Space Partitioning](https://en.wikipedia.org/wiki/Space_partitioning) to allow for a larger more optimised map. The map will have entry points for the “Zombie” like enemies that spawn into the map each wave to enter through. It will also have sections that will add a challenge for players, such as confined spaces, while also having more open spaces with blocks of cover from the enemies. The Player will interact with the enemies with a projectile-based weapon such as a gun by firing projectiles at enemies and killing them. Therefore, each enemy will have attributes, such as: health, damage, size, etc. These enemies will need be rendered differently to the map as they should be distinguishable from walls. The enemies will also follow a path finding algorithm like Dijkstra’s or A\* to intelligently follow and attack the player. The Player will have the ability to gain points from attacking enemies and then use these points to purchase upgrades for themselves as the waves progress and the difficulty is increased. There will also be health packs that spawn and allow the player to regenerate health to give them a chance to survive longer.

# Background Research

As I mentioned earlier, Ray Casting as an algorithm for rendering Video Games is quite old, therefore the games I took most inspiration from is 90’s games such as Wolfenstein 3D (1992) and Zero Tolerance (1994). I also wanted to research specific systems in these games, such as how maps were handled, not only for method of storing data but also keeping calculations optimised and efficient. Also analysing what about these games make them good that I could incorporate to my game fun.

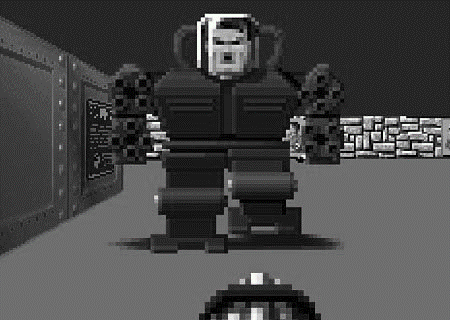


Wolfenstein 3D (1992).

## Hardware limitations

This game made strides in the fps (first person shooter) genre, by adding an interactive 3D scene, all working on very limited hardware. The Game was initially introduced for the PC, where computer hardware at the time could consist of 1 - 8MB of RAM, 0.5 – 1MB of VRAM, 66 MHZ CPU clock speed. This meant that calculations had to be very basic and kept very limited to allow for any playable Frame Rate. As I am making my Game on modern hardware, I will have many luxuries that developers at the time would not have had. I feel this is important to consider when I am implementing certain systems and algorithms as I don’t want to write poor code and want to make things as efficient and optimised as I can.

## Story

It was set in Nazi Germany, were you play William Joseph “B.J.” Blazkowicz, whose goal is to fight through rooms of Nazi soldiers, with the final intent of killing Hitler. The game however does add multiple fictional aspects to make the game less serious and more fun. Such as Robot Hitler and other separate “Mini Bosses” to add a sense of progression. It Game technique it uses is to add hidden rooms, which contain items and routes of progression such as health or ammo. I think this is something to consider for my Game.

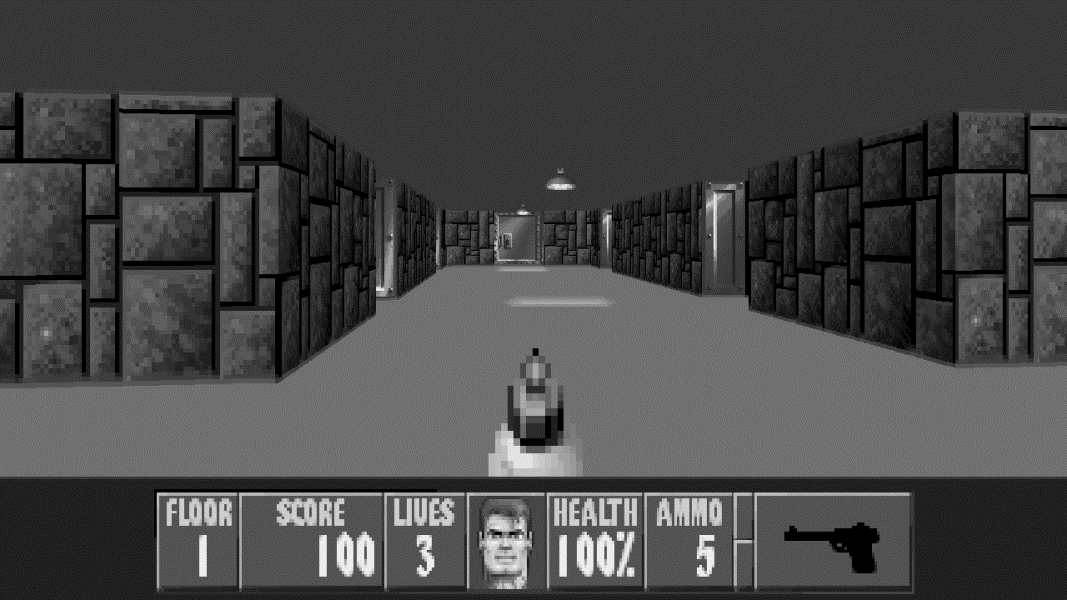
Robot Hitler

## Algorithms

Wolfenstein 3D uses grid-based ray-casting to render the Map and walls. It however uses sprite-based rendering to draw characters and weapons and collectables. This is because to allow for variable height in the enemies and collectables a different algorithm must be used, by instead of rendering a vertical column based on distance of a ray. A sprite is rendered and angled towards the player to give the illusion of enemies looking at the player. The health is just a player attribute which is reduced when an enemy attacks the player. However, medic packs can be found around floors and levels to increase the players health. The map is large grid of cells where each cell has a specific Tile Type which describes its attributes, such as texture needed, functionality (can it move).

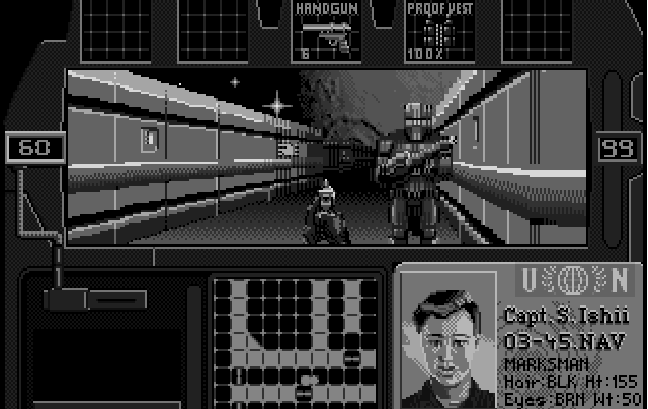
## Design / UI (User Interface)

The User Interface is a simple design, only displaying the necessary pieces of information to user: floor, score, lives, health, ammo, weapon. There is also an icon of the player which will dynamically change when the player takes damage, where the face get’s bloodier as the players health decreases closes to 0. I think I want to include similar pieces of information to the user as it makes the space on screen feel less empty. However, I intend to display it in a slightly more concise way. As mentioned earlier with the hardware limitations, this has had an impact on the resolution of Game assets. Wall textures are pixelated, only displaying a simple brick pattern, with the player icon being very low-resolution pixel art. I intent to only include very few assets in my game as I feel they are unnecessary and will just affect the efficiency of the game negatively.



Wolfenstein HUD

## Zero Tolerance (1994)



Zero Tolerance (1994)

Zero Tolerance was a game from 1994, also implementing a Ray Casting engine design. With a similar HUD to Wolfenstein, by allocating a whole part of the screen to it. I feel that in my Game I want to have a lighter weight HUD, where the focus is more on the surroundings. Therefore, I wanted to use this game as another example of the style at the time, and how I would like to stray away from that.

# Features

|  |  |
| --- | --- |
| **Feature** | **Description** |
| Weapons | Weapons will be used as a way of interacting with the enemy. The weapons will be used to shoot and deal damage to the enemies. They will have to frames of animation, a holding state and a shooting state. Different weapons will have different attributes such as damage and fire rate. Different weapons can be acquired after collecting points from killing enemies. If the player hits an enemy, the enemy will display a red blood stain, this indicates to the player that they have dealt damage to the enemy. |
| Collectables | Collectables will sit on the ground and will give something to the player to help the player survive longer. Current collectables consist of: Health, Ammo. Health pack regenerating the players health bar by certain amount. Ammo replenishing the players source of ammo. |
| Waves | The Waves system will be an indication of the stage at which the player is within the game. Gameplay will change as waves develop; this will be what makes the game interesting. |
| Enemies | Enemies will be the main source of gameplay, providing the only threat to survival for the player. They will act as “Zombies”, mindlessly travelling in the direction of the player. Where enemies will become more powerful and fast as waves progress. They will be controlled by a path finding algorithm, such as Dijkstra’s or A\*. This will give them a realistic feel and provide a sense of threat to the player. |
| Points | Points will be rewarded to the player upon dealing damage to enemies. Different amounts of points will be rewarded based on the weapon used. Points can then be redeemed for upgrades to player such as: Max Health increase, Increase in Ammo Capacity, Improved Weaponry. |
| HUD (Heads Up Display) | The HUD will display pieces of useful information the player and will allow the player to have a sense of direction and base the next move from this information. For example, playing more restrained when on lower health. My goal is to make the HUD feel “light-weight”, so it doesn’t clutter the screen for the player. |
| GUI (Graphical User Interface) | I want my game to have a basic GUI just to control attributes of the game such as: pausing, exiting the game, viewing guide, restarting level. I intend to do this with my own GUI framework made in python to have more control over attributes. |

# Intended User Interview

Me: Do you play a lot of first-person shooters?  
Adam: yes, games like Call of Duty and Rainbow Six Siege.

Me: What do you like most about the games you play?

Adam: I like all the different guns that the games have and using them to strategize.

Me: What do you like least about the games you play?

Adam: I wish they were more casual, so I could play for a shorter time instead of needing an hour to have a game.

Me: I am making a game in which you traverse dark levels filled with enemies to get to the end, with weapons and powerups scattered throughout. Does this sound like something you would be interested in?

Adam: yeh that sounds like it would make an interesting game. How many levels are there?

Me: There will be 5 main levels and a tutorial level at the start, displaying all the different pickups and powerups so the player understands the game content.

Adam: So, it’s quite a small game? That suits me because I was talking about games being too long. How many weapons are there?

Me: There are 4 weapons, they are quite different from each other in ammo capacity, damage and fire-rate to create a varied experience.

Adam: That’s a decent amount. How am I getting these weapons, do I buy them from a shop?

Me: No, they are obtained from pickups in the level which have been pre-placed.

Adam: Ok, so they are designed for level situations. Speaking of levels, how are the levels created? Are they designed or randomly generated?

Me: I have made the levels before hand, as I feel this gives a more controlled and entertaining experience when only playing through 5 levels.

Adam: Yeh, that sounds like it would be more fun if the levels are designed.

Me: My final question is, do you think this game sounds like something you would want to play? Do you have any final comments for the game?

Adam: I think the game sounds really good. From just hearing about the game, there isn’t anything I think you should change.

== Interview Complete ==

// Blank space has been left for document formatting

# Technologies / Algorithms / Theory of my Game

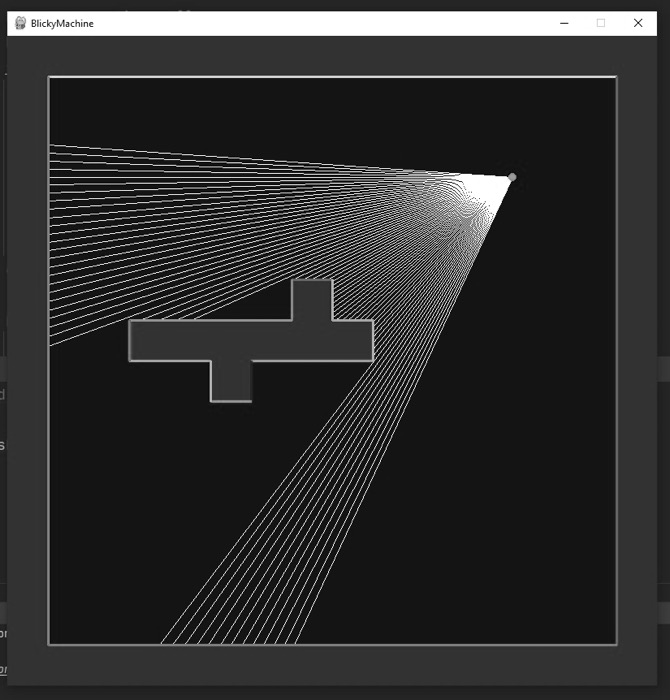
As I am developing my game on Modern hardware, the concern for optimisation is less relevant. However, all calculations are being completed on the CPU which is slow in comparison to the speed of a graphics card. Also, the language I am using, python, is Quite slow. This does mean that not every problem can be brute forced and some thought is needed for certain solutions.

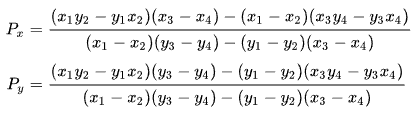
## Programming Language

I have decided to program my game in Python using the Pygame graphics library. I decided to use Python for multiple reasons. Python is a very quick language to develop in, giving the programmer a lot of short cuts and comfort features that make life easy. However, with these comforts comes possible decreases in program efficiency, which do need to be accounted for when implementing algorithms. I also chose Python as it is a very simple language to write in, which means I don’t have to spend time thinking about language syntax, I can spend my time understanding algorithms and not have to be limited to language knowledge. Python is also a very widely used language that will provide many opportunities for trouble shooting.

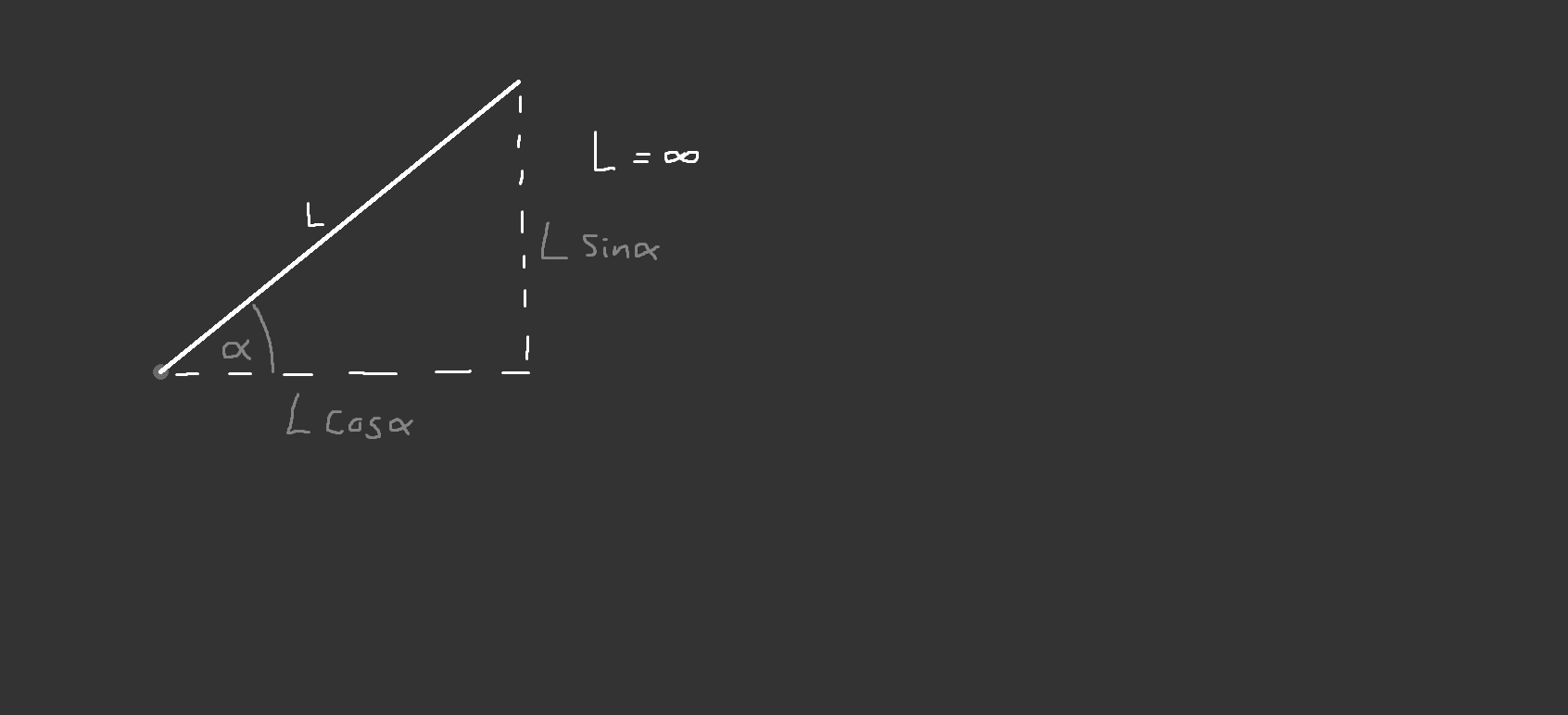
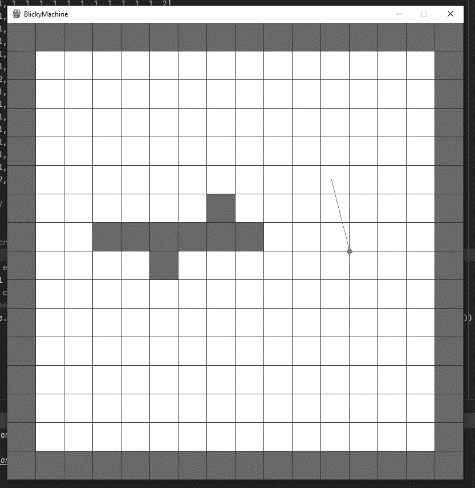
## Algorithms

### Ray Casting

Ray Casting an algorithm which is used to render a fake 3D scene from a 2D top-down image. The algorithm takes distance data and can recreate a scene. It gets its name because the basis of retrieving this distance data is attained by projecting an invisible ray from a player’s position and checking if it collides with a wall and then taking the distance to that wall if it does. The image on the right has shown the Rays as white lines and the player as the green dot. As the image shows, the rays are calculated from the players position and passed into a mathematical function to calculate the distance to what it hits.

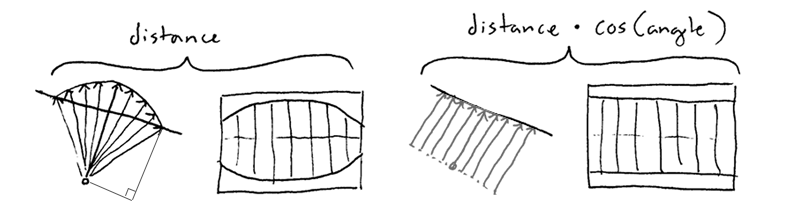


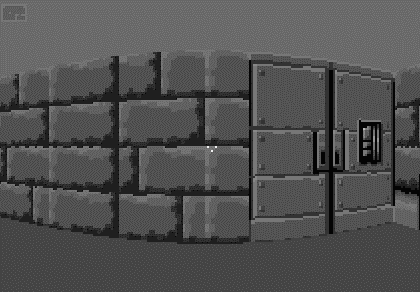
Line-Line intersection Equation

The way a ray is casted is shown in this diagram. The length of the ray “L” doesn’t matter, it just must be very large so that we can be sure that we can see all walls near us. What is important is the angle at which the ray is casted, shown in the diagram using alpha. We then can get the width and height of the ray as a vector using polar coordinates, or just multiplying the length of the ray by sin or cos for the height and width respectively. Once we have this ray defined as a vector, we can pass it through the mathematical equation known as “line-line intersection” which will return if the ray hit a wall and then the length of that ray from the player to the spot on the wall it hit.

Once we have the distance data for the rays. We then construct a vertical column for each ray that was fired on the screen, with the height of this column being dictated by the length of the ray, where the height of the column is calculated by

columnHeight = (CELLSIZE \* height) / RayLength *if* RayLength > 0 *else* width

By constructing these columns one after another, it gives the effect of a wall. However, a key thing to note is that distance of the ray used should not be the raw distance calculated. If we use the raw distance data, a fisheye affect appears as the rays being shot are based on the circumference of a circle. Therefore, to remove this fisheye affect, the distance must be calculated by (distance \* cos(alpha)).



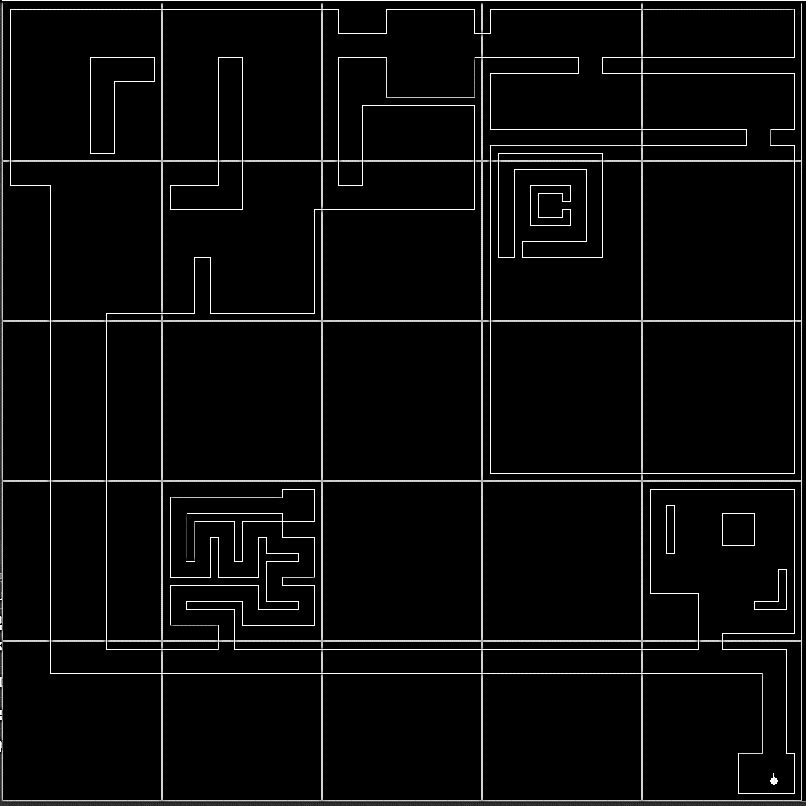
### Chunk System Structure

I used a chunk system as the data structure to store my map and therefore I wanted to mention this before how I handle the map.

A chunk system is a type of space partitioning algorithm, which partitions geometry into smaller spaces. It is important as it provides a simple boost in efficiency, it allows for only parts of the world around the player to be drawn to the screen instead of the whole word which would be very slow and intensive on a large world.

If I have a 100x100 world, where the coordinates of this world go from (0,0) to (99,99). If I want to get information from the world, I can do so very easily because I can just index into the map array in the position of the coordinate. However, this becomes very slow when, every frame, I have to iterate over this entire world. So, one method of fixing this is to split the 100x100 world into a 10x10 of chunks where each chunk contains a 10x10 of world information, so before to get the 0th index in the full map it would just be “world [0]” however now it would be “chunks [0][0]” because I’m accessing the 0th element from the 0th chunk.

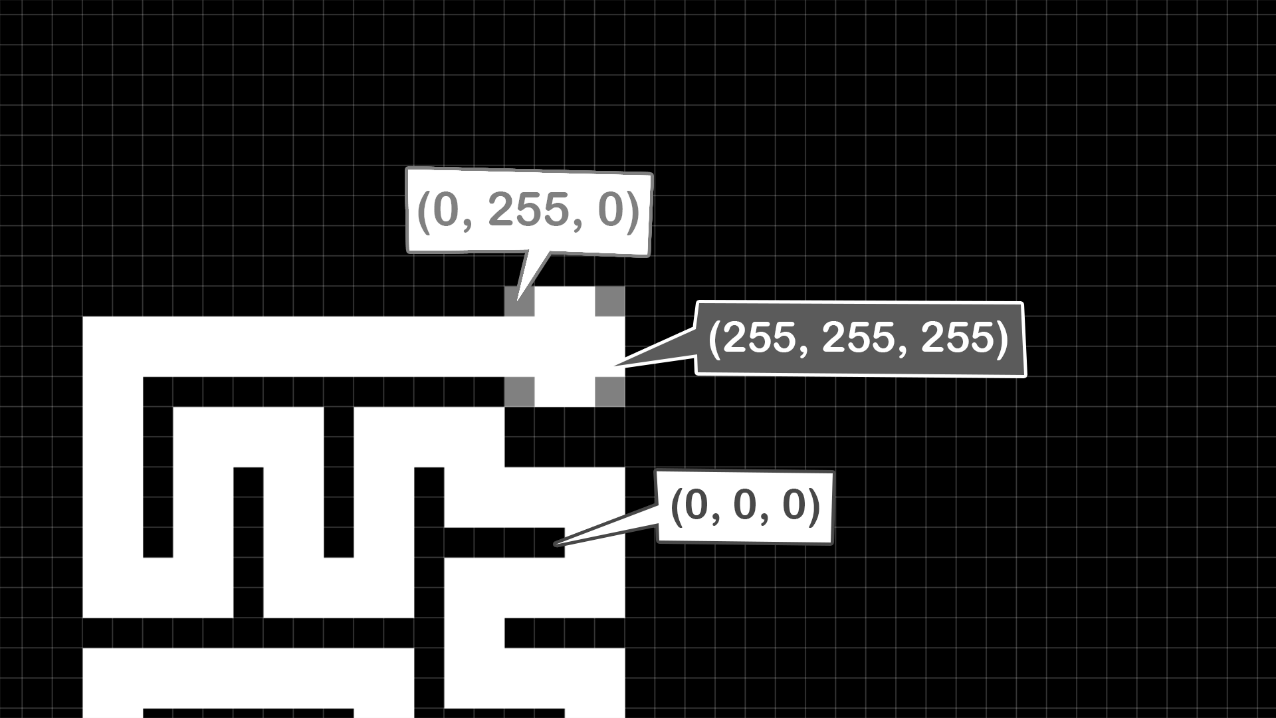
The idea is to most efficiently store the information for this structure to be formatted. To do this I use a mix of a Dictionary structure, where the key is the coordinates of the chunk, and then a list of self-made “line segment” objects, which just stores both points of the line, with each chunk having corresponding world information.

The chunks system has 2 essential methods for interacting with the structure, which is: Insert and Query. Insert, takes in a point coordinate and calculates which chunk should it be in, by taking the x and y and floor dividing by the size of each chunk, and adds it to that chunk’s world information. For example: if each chunk has a size of 10x10 then the point (36,12) would be x = floor (36 / 10) = 3 and y = floor (12 / 10) = 1 so the chunk coordinate would be (3, 1). Query would take a coordinate and a radius, where the radius is how many chunks around the coordinate given should be returned, and then it calculates which chunk the coordinate given would be in (3, 1). Using this information, I can easily calculate which chunks are surrounding (3, 1), for example top left diagonal would be a decrease in the x and y to make (2, 0). Then I iterate over these chunks in the radius and put all the world information into a list to be returned to the user. The user now has a list of reduced world information.

Example of map, split into chunks.

### Map-Generation

For my game, a map is a vital part of the experience and requires algorithms to generate, construct and handle. I need a way of storing the map data that allows me to easily access the information quickly and efficiently. For this I used a mix of both a chunk-based structure and a linked list.

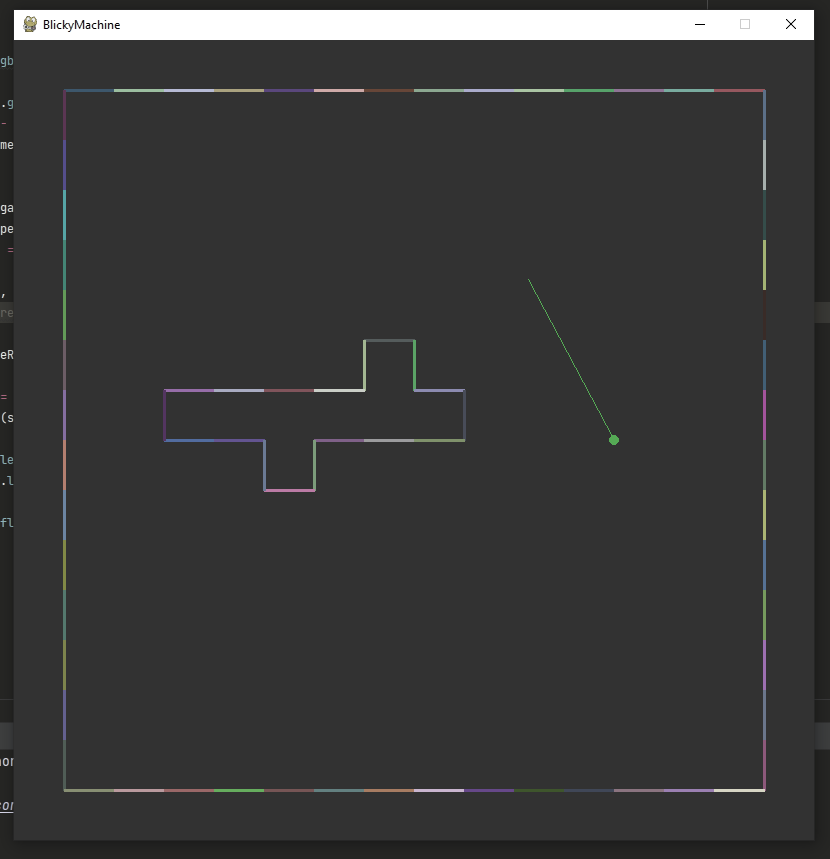
Due to maps having a lot of detail and information, it can be an arduous task to create a map and make quick or large changes. To solve this problem, I decided to store my map as a PNG Image. This solves many of my problems, it allows me to make changes very quickly and efficiently, it is very easy to add new tile types, and it keeps the data in a very compact form (compact in terms of use, not file size). Typically for storing the data of a map, an array would be used as it can be kept in written code and neatly ties together with the program. However, the biggest drawback to this, is that maps cannot be that large, as having a 100x100 list in hard code would be very large. Also viewing the map easily, requires some sort of formatting function to view it. Whereas, when stored in an image it can be viewed easily because it is already an image file. The only drawback I found, when using an image as a storage medium, is that as the only way to denote the tile is by pixel colour value, it can be very easy to be off by some very small amount and then that pixel is automatically not recognised by my program. This doesn’t happen all time, and my solution was to provide limits to the colour values for white and black, which has helped, however I think that this storage medium only works because I am storing only a few different colours. With the obvious benefit of an image being I can create it in photoshop by drawing it.

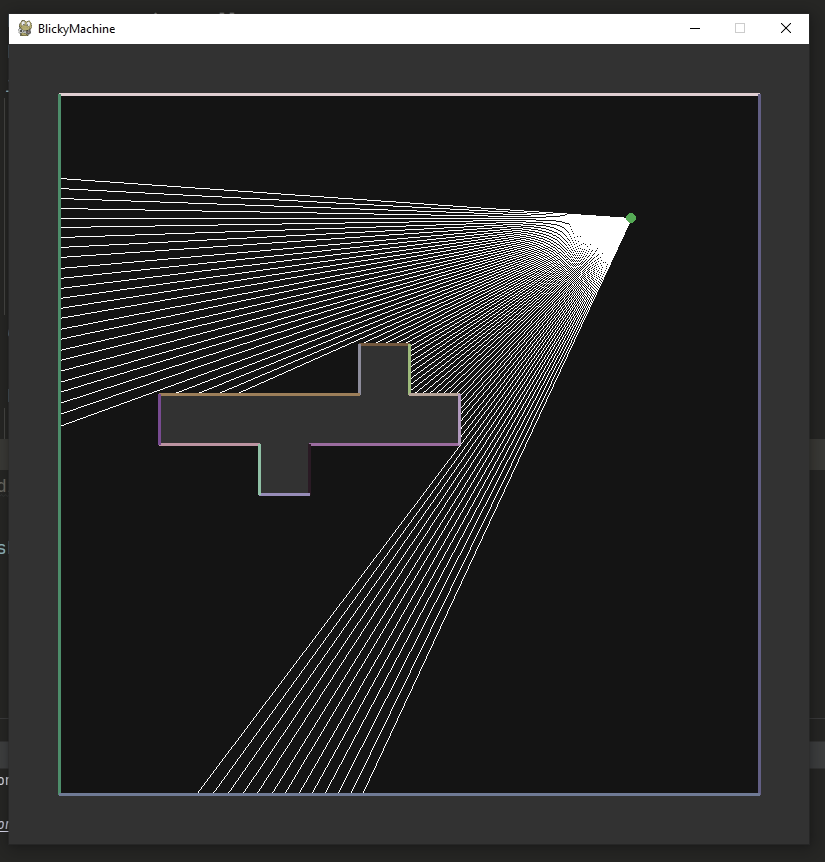
RGB Data for pixels

However, the image is made up of pixels, yet my ray caster works with line segments. Where each wall is represented by two points making up a line. Therefore, I needed to make an algorithm to convert between pixels and a chunk system of line segments.

### Cell Outlining Algorithm (Wall generation)

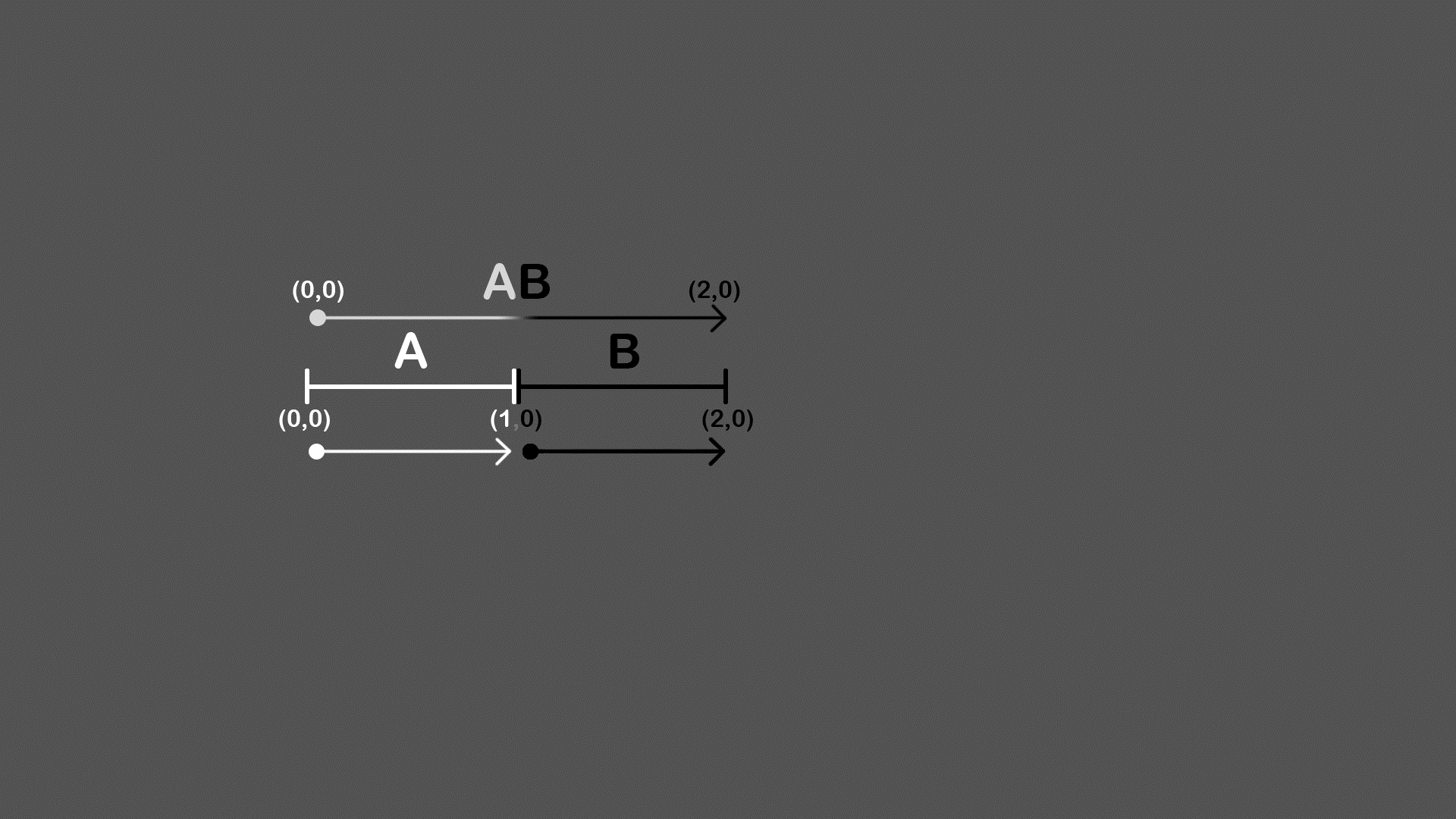
I needed an algorithm that could take this pixel data and return a chunk system of line segments. Once I have this chunk system of line segments, I can then use it to calculate ray intersects every frame in an optimised way. The data I had was a PNG Image, so my first step was to iterate over each pixel and get the RGB value, using the python pillow library. Then I would add I certain tile ID based on the colour. Now I have a 1-Dimensional linked list of every cell in the map.

The objective of this algorithm was to generate full, connected lines, given an array of tile data. Effectively outlining the entire map with lines, which could later be used as walls.

My idea was to first generate many small line segments based on each tile. Then take these small line segments and for line segments which are both connected and going in the same direction, I could replace these two small lines with 1 long line. I can generate the initial small lines, by iterating over the grid of chunks and checking adjacent cells to see if they’re empty. If the tile above was empty, I could create a line there, but if on the right there is a wall, then I would know to not create a line there because it wouldn’t be seen. Then once I have done this for each cell in a chunk, I could take all these lines and insert them into the main chunk map, to be used for the actual ray casting.

Shape made up of many small lines

Same shape, but with minimum amount of lines

To further note on how I go about checking if two small lines could be made in to one large line. In my map there are only two possible orientations for lines: vertical or horizontal. A line could be considered vertical if it had different y coordinates and horizontal if otherwise. Then I would need to check if the two lines shared a point, and if they did, I would need the two non-shared points which would act as the coordinates for the new longer line segment. To do this, for horizontal lines I first check that all y coordinates are the same, as two lines could have the same x, but be on different y’s. Then I check that to see if two points match by performing a logical AND on two sets containing the x coordinates of each line respectively, e.g.,

Two lines, begin combined into one

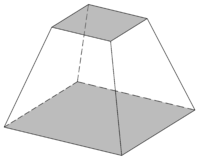
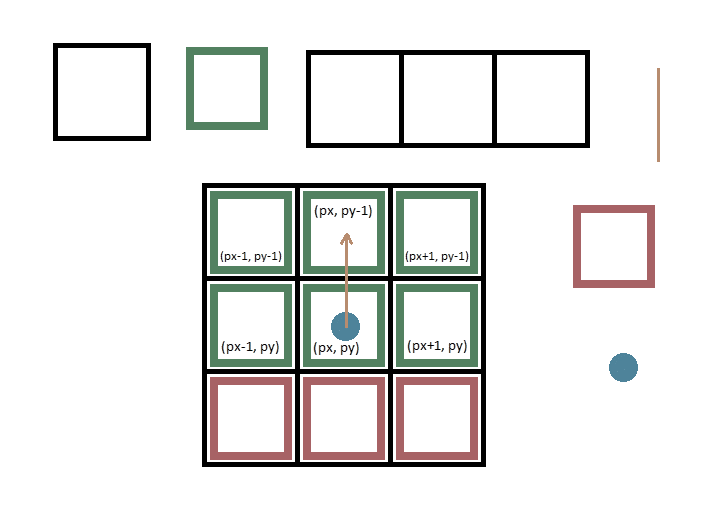
*if* {self.x1, self.x2} & {*otherLineSeg*.x1, *otherLineSeg*.x2}:

Then, because when I construct the smaller lines, I always construct them left to right, and I know that when I am creating these longer lines, I am always iterating from left to right. Then I know that the longer line segment, will just be the first point of the first line to the second point of the second line.

lineSeg(self.x1, self.y1, *otherLineSeg*.x2, self.y1)

This can be repeated with inverted x and y’s to get the same result for vertical lines.

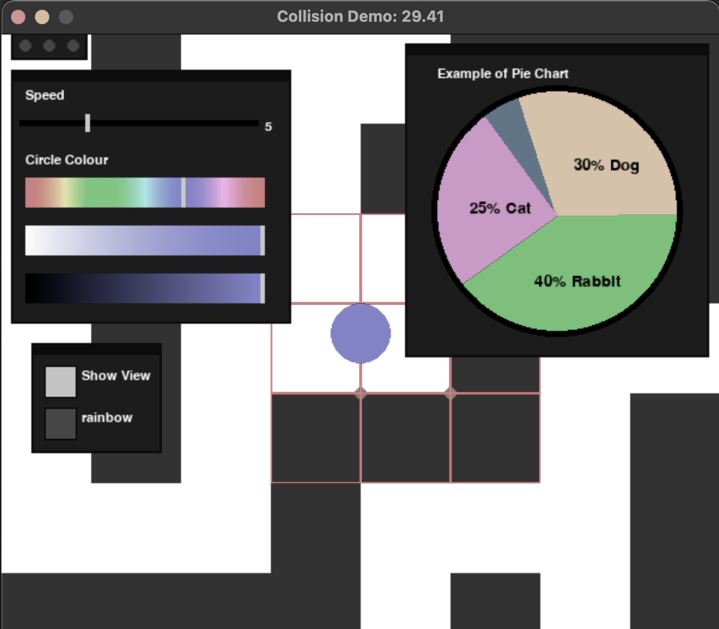
### 2D Chunk Frustum Culling

Frustum culling is a method of program optimisation and is typically a 3-Dimensional method. However, the premise of the algorithm can be applied to 2-Dimensions. In 3-Dimensions, the view of the player is defined as a shape called a Frustum, which is a square-based pyramid that has had the top cut off. So, Frustum Culling is only rendering things that are inside of the Frustum. This gives a large boost to efficiency as we are removing most of the calculation normally done by not calculating things behind us. As my game is a faux 3D game, I can’t apply the algorithm exactly. Instead my view range is more of a sector, therefore I would only want to render things inside my view sector.

Frustum

To do this, I can very easily decide what is in view by the direction I’m looking in, and then picking the chunks surrounding me accordingly to render. For example, if I was looking straight up. I wouldn’t want to render the 3 chunks directly behind the player. I can create offsets from the players chunk to render based on which direction the player is looking in, then put these offsets in a lookup table to quickly decide which chunks to render.

### Immediate Mode Graphical User Interface Library

In my game, I want to be able to have buttons and sliders to control specific aspects of the game. For example, on the start screen of the game, I want to have a button to start the game, but I also want a slider that can control the resolution. I also want a pause screen that has buttons that allows the player to exit the game or restart a level.

To achieve this, I decided to write my own User Interface library complete with buttons, sliders (variable sliders, colour sliders), a pie chart system for representing player data, a tab system to control the state of open UI widgets, the ability to move windows by clicking and dragging and some sort of window hierarchy system to sort widgets into layers.

The system works by having wrapper objects, which work as the basis of storing all UI objects. Wrappers act as small windows which contain the actual Interface objects. A wrapper could contain some buttons, some sliders, which are all positioned with absolute coordinates, relative to their parent wrappers original position, wrappers cannot be resized. This makes the moving of windows very easy as when the player clicks on the window to move it, the subtraction of the wrapper position from the cursor position gets a vector between them, which can then be applied to the cursors position to calculate the new position of the wrapper and therefore each elements screen coordinate inside the wrapper respectively. Each UI object inherits from a base object class, which contains basic information about the object, such as position within wrapper and width and height of “hitbox” which is later used to check for interactions with the object. So, when the user clicks the screen, the system uses a simple AABB collision check with the hitbox of each wrapper and the mouse cursor, then the same check is done with the collided wrapper to find the object being clicked. Each object shares methods that can be called and these object methods are what apply functionality to the system. For example, the “LeftButtonDown” method defines what should happen when the left mouse button is clicked down on the object. In the case of the button object, it first checks that the mouse is within the button itself and then alternates the current state of the button i.e., on => off, off => on.

Graphical user interface, application

Description automatically generatedAnother part of the GUI system is the Tab system. In which there are small buttons in the top left corner that each control its own wrapper, respectively. They programmatically generate more tabs when another tab is created, which makes adding tabs very quick and easy. Each wrapper has an attribute which controls whether the wrapper is open or closed. Then when using the wrapper, it will only be loaded by the system if it’s attribute is set to open. This makes the logic of the tab system very simple, as the tab just alternates the state of its respective wrappers open attribute, when clicked. The window hierarchy system gives wrappers an order of precedence so that they do not get buried under other wrappers. When the mouse is clicked and the clicked wrapper is calculated, the wrapper that was clicked is simply moved to start of the global list of all wrapper objects “GUI\_Objects”.

The way I incorporated this into my game, was with the many different state screens (pause screen, start screen, death screen…). These were used to give interactive control over the state of the game. For example, in the pause screen, it contains a Restart Button, Replay button and an Exit button. In the start screen of the game, it contains a variable slider that controls the resolution. I found this GUI very useful in creating these elements.

### Sprite Handling System

Diagram

Description automatically generatedThe sprite system is used to handle all the calculations and drawing of each sprite in the level. The sprites are stored as Game Sprite Objects which store information about the position in the level of the sprite and the image used to draw the sprite. These Game Sprite Objects are generated at the start of runtime by calling the generate sprite object’s function, which iterates over all tiles in the level and if that tile is representative of a sprite, then it is added to a list of sprite objects, this list is then returned, and the sprite objects can be used in the system.

Then during run time, the “draw\_sprites” function is called which is a multi-stage process for correctly positioning and layering the sprites. The first stage is to calculate certain information about the current “camera” position (camera in this case is just referring to the dimension of the sector of vision in the 2D scene for the player), the information required is the camera plane as a vector and the direction of view as a vector. Args[0] = Distance to plane, Args[1] = Width of plane

lp = createVector(args[1] / 2 \* math.cos(player.facing - math.pi / 2), args[1] / 2 \* math.sin(player.facing - math.pi / 2))

rp = createVector(args[1] / 2 \* math.cos(player.facing + math.pi / 2), args[1] / 2 \* math.sin(player.facing + math.pi / 2))

planeV = createVector(rp.x - lp.x, rp.y - lp.y)

dirV = createVector(args[0] \* math.cos(player.facing), args[0] \* math.sin(player.facing))

Text

Description automatically generatedThe next stage is to sort the sprites based on their distance from the player, so that sprites that are furthest away will be drawn last. Then each sprite is iterated over to be drawn to the screen. First a vector from the player to the sprite is calculated to be used in the screen coordinate calculations. To make sure the sprite is only drawn when in front of the player, the dot product of the direction vector and sprite vector is calculated, where if that dot product is > 0 then I know the sprite is in front of the player. The Plane, Direction and Sprite vectors are all passed into a function to calculate where on the screen the sprite should be positioned to give the effect of being in a 3D environment.

After the screen position has been calculated, it checks that the screen coordinate is on screen and fits the expression 0 < screen.x < width. This is important as the screen x is used to make sure the sprite isn’t drawn when behind a wall. It calculates the wall slice that it is in, then by comparing if the distance from sprite to player is less than wall to player, the sprite can be confidently drawn in front of any walls. Finally, any run time scaling or re-positioning of sprites to account for different resolutions are applied, and the sprite is drawn to the screen. This is then repeated for every sprite in the system. Note that the camera position is only calculated each frame, to account for player movement, and isn’t calculated for each sprite iteration.

### Enemy AI System and Pathfinding

The Goal of each level in my game, will be to reach the end of the level. However, this wouldn’t be very difficult without some sort of obstacles to get in the way. The obstacles I chose to be in my game, were zombies. These zombies will intelligently locate and navigate to the player, then once within a given range, will deal damage to the player until the player is dead. For the zombies to be able to navigate a complex system of corridors and locate the player, some sort of special algorithm is required. The algorithm I decided to use in my game, was the A star pathfinding algorithm. I chose this method, as it is supposedly the gold standard in efficiency and accuracy, as it is used in almost all large games with some sort of intelligent pathfinding.

A picture containing chart

Description automatically generatedThe algorithm works by first formatting the world data into a graph data structure with a system of connected nodes. To achieve this first step, I iterate over all tile data and, for each not wall tile, create a node object with coordinates at the centre of each tile and some other information set to defaults. Then, to form the graph of nodes, each node must have reference of its neighbour. To do this, I previously stored the coordinates of each node within a chunk system, then I iterate back over all the newly created nodes. Now, I can query the chunk system at each side of the node to not only check if there is a valid node at that position, but also to then get reference of that node to add to the list of neighbours.

Node Graph of map displayed as connected points

Once the data has been put into a graph, the data can be used to calculate the shortest path between 2 nodes (the player and the zombie). First, a simple yet effective optimisation is to simply delay the path calculation, as it would make the program slow to be completed every frame. Therefore, a delay is added of 100 frames in between each calculation. A noticeable issue with the strategy is that if the player is moving quickly while near the zombie, then the delay in calculation can become quite apparent. However, I was able to nullify this effect, by giving the zombie path finding a 2-stage sequence, so when the player is not in direct view of the zombie, it uses the A\* algorithm, but otherwise the zombie will go directly for the player when in view. This not only creates a far more realistic effect but is overall more efficient as less A\* solver function calls are required.

The A\* solver function uses the graph structure to progressively calculate the shortest path. The system of graph nodes each have special attributes: parent node, heuristic cost, distance from start, reference to neighbour nodes. The first step is to iterate over each node and reset these attributes, where each nodes heuristic values are set to infinity other than the starting node which defaults to 0. The starting node is added to the list of nodes to check, and then each of its neighbours are iterated over. The local heuristic is calculated first, by adding the root nodes local and the distance to the current neighbour. If the new local heuristic is less than the neighbours local, then the neighbours local is updated with the new calculated local. Then the neighbours parent node is updated, with the root node, and finally the global value is calculated by doing the raw Euclidian distance to the end node and adding the neighbours just calculated local value. This process is repeated for each neighbour of the current testing node, and then every node to be tested. Until the current node is the end node, and there are no more nodes to test.

Now that the path for the zombie to take to get to the player is found. I needed a function to convert this graph system of connected nodes. Into a walkable path for the enemy object to follow. To do this, I created a function which goes through each node and adds it’s coordinate positions to a list of instructions for the object to follow. It starts by looking at the end node, and progressively sets the current node equal to the parent of the current node, adding the coordinates to the list every time a new node is reached. Then once the current node has no parent, then the current node is the starting node. After this function is complete, the list of instructions is given to the zombie object for it to follow. The zombie movement function creates a normalized vector between the zombie’s position and the next position in the list, which is then multiplied by the zombie’s speed to make them move realistically through the coordinates, and not jump to the next position. The list of instructions is generated for each zombie in the level, where each zombie has its own corresponding movement instructions. The player and zombie nodes are calculated by taking the current absolute position and dividing it by the size of each node in pixels, then flooring the value to the nearest cell coordinate. Taking this value and using the graph chunk system to find the corresponding node.

# Objectives

* Map loads in from PNG image
* Map is divided into a chunk system
* At least 2 weapons
* Points can be gained from damaging enemies
* The game doesn’t end until the player is dead
* Zombies display they’ve been shot
* Zombies don’t spawn all at once
* The game runs at below 20% CPU usage
* The player has the option to upgrade the weapon
* The player has the option to upgrade the armour / max health
* The player can shoot and strafe at the same time
* The game can be paused
* The game can be restarted
* The enemies use pathfinding algorithm to move towards the player
* There are collectables on the floor for the player to pick up
* The player only can see walls from adjacent chunks
* Add a completely hidden feature that could only be found by prior knowledge or mistake.
* A GUI Main menu that displays:
  + Game guide
  + Exit
* A HUD that includes:
  + Health
  + Ammo
  + Wave Number