Prosser Reece

[Email address]

Centre number – 20570

Candidate number - 4482

Abstract

Welcome to Arena Assault, a fast paced, procedural generated game where you will face relentless enemies and bosses wanting to kill you. WILL YOU SURVIVE!

Arena Assault

Survive the arena! Conquer the chaos!

Contents

[Analysis 1](#_Toc210037860)

[Project Summary: 1](#_Toc210037861)

[Problem analysis: 1](#_Toc210037862)

[Background research: 2](#_Toc210037863)

[End user interviews 4](#_Toc210037864)

[Objectives: 5](#_Toc210037865)

[Modelling: 6](#_Toc210037866)

[The Ray: 7](#_Toc210037867)

[Horizontal Intersections 8](#_Toc210037868)

[Vertical Intersections 10](#_Toc210037869)

[Design 12](#_Toc210037870)

[Mission Statement: 12](#_Toc210037871)

[Hierarchy Graph 13](#_Toc210037872)

[Key Data structures: 14](#_Toc210037873)

[Complex Functions: 14](#_Toc210037874)

[Function 1: Ray projection 14](#_Toc210037875)

[Function 2: Calculating Ray properties 16](#_Toc210037876)

[Raycasting 18](#_Toc210037877)

[A\* Pathfinding 20](#_Toc210037878)

[File Structure: 22](#_Toc210037879)

[Technical solution 24](#_Toc210037880)

[Testing 25](#_Toc210037881)

[Evaluation 27](#_Toc210037882)

[Appendices 28](#_Toc210037883)

# Analysis

## Project Summary:

My first-person shooter game allows the user to walk around a procedural generated map with a weapon and be able to shoot enemies that spawn in random locations that are trying to attack you and eliminate you. Your aim is to eliminate all the enemies and try and survive the waves where more enemies keep spawning every time. At wave 10 you will fight a boss that will be harder to kill then all the other enemies. The boss will have random attacks at random intervals. Your aim is to dodge these attacks while trying to eliminate the boss. Once eliminated, you will face more waves of enemies until you are eliminated. The aim is to survive the longest time.

This game is a skill and action game; you will need to create tactics to survive the waves of enemies that are trying to attack you and have good reaction time for enemies that are sneaking up on you.

This game is an infinite game with infinite waves so there is no winning the game however, each time you kill an enemy you will receive 1 coin which you will be able to spend in the shop on the start menu. This will allow you to either save up to buy a new weapon or be able to buy customisation for your weapon so like increase ammo hold or faster firing.

You lose the game when the enemies/ the boss kill you and you will return to the start menu. After this you can enter the shop or start another round. It will also display what wave you got to.

Your scores are going to be the waves you have reached.

## Problem analysis:

My shooter game is going to be solving many problems like similar gameplay, storage space and profit over fun. The main problem with games now is for example Fortnite and rainbow six siege is there is only limited maps to play on. So, after multiple games and getting the same maps, the game starts to become boring and not as fun to play. With my game it will be using procedural generation to generate different levels each time you play so the game will always feel fresh and keep you engaged.

Most modern games now require high storage space and require hundreds of Gigabytes of storage just to download and play the game. Also, most games now need expensive graphics to play and require consoles and expensive PCs. My game is going to require a small storage space and won’t need high end tech to be able to run it, and you will still be able to have a fun and engaging time playing.

Game now focuses on profit and requires money to be able to customize things in game e.g. skins and weapon skins and don’t focus on the player having fun. To solve this, my game is going to focus on user experience and allow the user to have a fun and enjoyable time without having to spend a fortune. Abilities and skins in the game will all require in game currency to buy it that can be obtained from playing the game and eliminating enemies in it.

## Background research:

There is many action and shooter games out there. Gunfire reborn is a first-person shooter in which the user must head into levels full of enemies with distinct attack patterns and tough bosses which are randomly spawned at different parts of the map. The aspects of the game I like are the UI/layout, the use of a pathfinding algorithm for enemies to locate you, and the procedural generated levels. I intend to use these features in my game however my game will be using Raycasting.

User Interaction & Design*:*

Firstly, the minimalistic UI of Gunfire Reborn appeals to me, especially the clever use of colour depth and the red outline for bosses to signify their superiority and power. While the game features a detailed 3D map, my game will adopt a ray casting algorithm. I have chosen ray casting as it allows the game to look like its 3D without the difficulty of coding a 3D algorithm. The game’s art style is unique, displaying a modern aesthetic. This creates an immersive and enjoyable experience for the user.

A screenshot of a video game

AI-generated content may be incorrect.

A video game screen with a couple of guns

AI-generated content may be incorrect.Gunfire reborn also offers a single-player mode or team up with friends in 5-player co-op mode. I am not going to corporate the use of multiplayer in the game as my game will be focused on first person where you have all the control over the game. The game also uses a pathfinding algorithm to ensure the enemies can find the quickest path to the player which is something I also want to implement in my game.

Gameplay

Players at the start can select heroes with different abilities and are able to experiment with the different attacks the hero can do. The game offers many different types of waves, where each wave has a different number of enemies you must fight. It also incorporates a boss fight at the end where the health bar is displayed at the top. The game has a shop where you can buy different guns to fight with. Players can experiment with different weapons, items and hero abilities. The game is skill based requiring players to adapt to enemy attack patterns.

Gameplay Flow

The gameplay flow can be visualised as:

Start Menu

↓

Main Menu

|------Select game mode (singleplayer/multiplayer)

|------Start Game

|------Shop (Buy/Upgrade Weapons)

|------Settings

↓

Randomized Level Generation

↓

Player can explore map

↓

Enemy Waves

|------Combat (3D Mechanics)

|------Random attack patterns

|------Pathfinding algorithm for enemies to find player

|------Boss Fight (At Specific Intervals)

↓

Can continue to carry on player will become new Stage of game

↓

In app purchases to buy heroes

↓

Return to Main Menu (Loop for Next Level or Buy Items)

↓

Restart can select different maps and heroes

Features

|  |  |  |
| --- | --- | --- |
| **Feature/Action** | **Gunfire Reborn Implementation** | **My Implementation** |
| **Game design** | Uses 3D to create the game (unity engine) | Raycasting algorithm to make 2D game appear 3D |
| **Enemy Waves** | Procedural generated enemies and bosses | Procedural generated spawning of maps and enemies spawn at random times |
| **In app purchases** | Require purchasing the game. | Free game experience |
| **Boss Fights** | Dynamic boss fights with unique attributes | Intense boss battles that influence gameplay progression |

Algorithms:

Raycasting- The raycasting algorithm will allow me to code in 2D and display it as 3D. It Casts rays from the player’s FOV in the direction they are looking. It then Checks for intersections which include horizontal and vertical intersections with walls or objects depending where the player is looking e.g. looking right or looking up. Once it determines whether the player is facing up/down left or right it steps through the grid using tan a to find the next wall hit.

Once a wall is detected, the algorithm calculates the distance from the player to the wall using the Pythagorean theorem. This distance is used to determine:

* The height of the wall slice to render (closer walls appear taller)
* The brightness or shading of the wall (closer walls are brighter)
* The texture column to display, based on the exact hit point

This process is repeated for every ray across the screen

A\* pathfinding – This uses a specific function called (heuristic function) to determine the best path. It does this by considering many factors like movement cost(G), heuristic cost(H) and total cost(F).

F = G + H

It picks the best node by comparing which one has the lowest total cost. This finds the most efficient way to get to the player.

In my project the A\* pathfinding allows the NPC to navigate through the map to get to the player in the most efficient way.

Procedural Generation- Uses an algorithm which will generate unique content. The algorithm will use seed values which will allow the same seed value to produce the same map. This will allow multiple seed values to be created allowing for different maps each time to be generated. The different algorithms are: Perlin Noise, Random Walk, Cellular Automata, or Fractal Algorithms

## End user interviews

My game is designed for a wide range audience, mainly children aged 13 and up. I have chosen this age due to the inclusion of action and shooting which may not be appropriate for younger children. The goal of this game is to provide an exciting and action-packed experience without the need for high end hardware. Unlike many games now that prioritizes profit over fun, my game primarily focuses on user experience and fun through procedural generated levels and infinite levels.

My game is aimed at children who are primarily busy but want to have fun. As my game is going to be simple, minimalistic and require not a lot of time to play, it allows the user to enjoy the time without having to worry about time.

However, the game isn’t just targeted at children who are always busy, it can be played by anybody who just wants to have a good time.

1. Have you played any first-person shooter games before? If so, which ones and how long was this ago. Did you encounter any problems?

“I have played a fps game called rivals about a month ago. It was very pay to win however the movement and gun mechanics was very smooth and clean.”

1. What features are most important to you in a first-person shooter game?

“The features I like most was the range of weapons it provided. This made it so I had to change my playstyle with each set of weapons. Also, you can change the different crosshairs in the game which will make it suit me more.”

1. What aspects of a game make it enjoyable for you?

“It’s not complicated it was a very simple matchup, and you are easily able to play a match under 5 minutes, so it is very timesaving.”

1. What makes a game “fun”?

“When you eliminate someone, it is very satisfying, and it has stats that displays your win streaks and how many people you have killed.”

1. Do you prefer long story games or quick action-packed games? WHY?

“I prefer quick action games because I can play it without worrying about time and I don’t like the games that require a lot of time to play.”

From the answer I have found out that the user would want me to implement some additional settings to allow the crosshair to be changed and add a kill counter. Also, from the answers the user also agrees that most game now are not as good as they used to be and are pay to win so if you spend money, you have a higher advantage over other people.

Pay to win game can occur when people are able to buy things that have an advantage to other players for example the user could buy a more powerful gun than the person they are fighting.

My game wouldn’t have any of this, so it provides a fair experience for everyone, so they all have the same advantage as each other. This makes my game focus on skills.

## Objectives:

1. Implement a Complex user-defined OOP model
2. When the space bar is pressed, the character will jump
3. When the A, S, W, D the character should move in that direction

-A to move left

-D to move right

-W to move forward

-S to move backwards

1. Random map generation

-To make the map different every time to keep the user engaged

1. Implement an A\* pathfinding algorithm

-this will allow the enemies to get to the player at the quickest time

1. Implement a boss fight

-where it has different attacks and much more powerful than normal enemies

- HP much more significant

-Higher DPS ( Damage Per Second)

1. When right mouse button clicked it will shoot the gun

* This will allow the bullet to travel straight killing enemies in it
* When contacted with the walls it won’t affect it only when it encounters an enemy

1. When shooting an enemy, it will kill them or deal damage to them based on where the shot is

-Headshot (instant kill)

-Body shot 25 damage

-Leg shot 12 damage

1. Implement a health bar for the player

The player will have 100 health

1. Implement a health bar for the enemies and boss

-The enemy will have 50 health

-The boss will have 200 health

1. Give enemies attacks

-This will make it so the user can’t just roam around freely

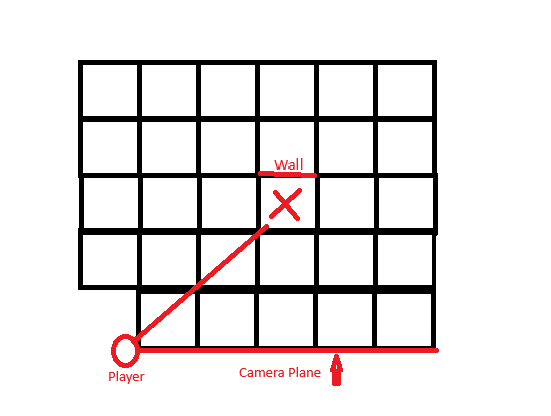
1. Have different kinds of enemies

* Include enemies that have more HP
* Different attack abilities
* Different movement speeds

1. Implement an Raycasting algorithm
2. Add textures and backgrounds to make it look appealing
3. Implement a shop where you can purchase different weapon abilities

## Modelling:

Raycasting:

Raycasting allows a map of 2D code to become 3D. Each square of the map can be either 0 (there is no wall) or a positive integer e.g. 1 (there is a wall)

Raycasting works by calculating for every x on the screen it sends out a ray at the player’s location depending on where the player is looking. This ray moves forward on the 2D map until it hits an object. When it hits the wall, it calculates the distance from the player to the wall using trigonometry and Pythagoras’ theorem.

The image on the left shows a ray that is cast from the player’s position to detect the wall. Once it has detected the wall it calculates the distance which is used to show how the 3D projection should look.

### The Ray:

we will first need to work the angle that we

Will keep adding onto the FOV / number of rays.

This will allow us to know when to cast the ray out

We calculate it by using the formula:

Ray Angle = Player Angle – FOV / 2

In my code I have implemented it here

A computer screen shot of a program

AI-generated content may be incorrect.

Next, we will need to keep adding the Ray Angle

We calculated to the FOV/ Number of rays which will then

Cast the ray out.

We calculate it by using the formula:

Ray Angle = FOV / Number of Rays

A screen shot of a computer program

AI-generated content may be incorrect.In my code I have implemented it in the

castRays function as shown on the right.

As you can see it keep looping through

The code and keeps adding the ray angle

To find out when to cast the ray.

### Horizontal Intersections

For the horizontal intersections we need to

calculate both the X and Y intersections

and the Ya and Xa

The X and Y intersections are the coordinates of when

the ray hits a wall

The Ya is the y distance from the first hit to the second hit

The Xa is the x distance from the first hit to the second hit

The general equation for the angle is:

Tan(a) = Ya / Xa

This is calculated by using trigonometry as when you connect the points it forms a triangle

This is shown on the image here

First, we need to find the first Y intersect for the horizontal intersection. This depends

if the player is looking up or looking down.

The reason for the -1 at the end is to check if there is a wall in the tile above.

If the player is looking up:

The equation is:

Y = floor (Py / tile size) \* tile size -1

Where Py is the players y coordinate and y are the first y intersection

A black background with white text

AI-generated content may be incorrect.

If the player is looking down:

The equation is:

Y = floor (Py / tile size) \* tile size + tile size

Where Py is the players y coordinate and y are the first y intersection

Next we need to calculate the first X intersection however the equation wont change depending on which direction the player is looking

The equation is:

X = Px + (Y – Py) / tan(a)

Where x is the first x intersection, Px is the players x coordinate, Y is the first y intersection, Py is the players y intersection and tan(a) are the angle in between the player and first hit point as shown on the image

This is where I have Implemented it into my code.



Next we have to work out the step size to get to the x and y coordinates for the horizontal intersections

The equation for the x coordinate step size is:

Xa = Ya / tan(a)

where Ya is the vertical intersection and tan a is the tangent to the line sina/ cosa

This is where I have Implemented it into my code.



For the Y coordinate tis depends if the player is facing up or down:

A screen shot of a computer code

AI-generated content may be incorrect.The equation for the y coordinate step size facing up is:

Ya = -tile\_size\_y

The equation for the y coordinate step size facing down is:

Ya = tile\_size\_y

### Vertical Intersections

For the Vertical intersections we need to

calculate both the X and Y intersections

and the Ya and Xa

The X and Y intersections are the coordinates of when

the ray hits a wall

The Ya is the y distance from the first hit to the second hit

The Xa is the x distance from the first hit to the second hit

First, we need to find the first X intersect for the vertical intersection. This depends

if the player is looking right or looking left.

If the player is looking right, the equation is:

A computer screen shot of a code

AI-generated content may be incorrect.X = floor(Px / tile\_size\_x) \* tile\_size\_x + tile\_size\_x

where Px is the players x coordinate and x is the first x intersect

Now this changes if the player is looking left:

The equation for this is:

X = floor(Px / tile\_size\_x) \* tile\_size\_x – 0.00001

where Px is the players x coordinate and x is the first x intersect

We subtract a very small value as this will avoid rounding into the next tile

Next we need to calculate the first Y intersection however the equation won’t change depending on which direction the player is looking

The equation is:

Y = Py + (X – Px) \* tan a

where Py is the players y coordinate, Px is the player X coordinate and X is the first X intersection



Next we have to work out the step size to get to the x and y coordinates for the vertical intersections which is different from the horizontal intersection

The x coordinate depends on if the player is facing left or right

The equation for the x coordinate step size facing left is:

A computer screen shot of a black background with white text

AI-generated content may be incorrect.Xa = -tile\_size\_x

The equation for the x coordinate step size facing right is:

Xa = tile\_size\_x

Now we have to work out the y coordinate step size for vertical intersection which is:

Ya = Xa \* tan(a)

Once a ray has hit a wall we have to calculate the distance from the player to the hit point. We do this by using the Euclidean distance. This will allow the program to know how far the object is away and able to draw the object the correct distance it is based on where the player is standing.

raw​= sqrt ((Xhit​−Xplayer​) **²**+(Yhit​−Yplayer​) **²)A screen shot of a computer code

AI-generated content may be incorrect.**

# Design

## Mission Statement:

My Raycasting game will be used to have a fun and enjoyable experience without having to spend loads of time on the game. It will be an FPS (first person shooter) game where you will encounter waves of enemies trying to kill you where it will use an A\* pathfinding algorithm for them to find you. To move in the game, you will use WASD on the keyboard to move (W – forward A – left S – down D- right). This game is designed for you to take a break from your studies and be able to enjoy a game that doesn’t take your entire day away.

### Hierarchy Graph

Still making the hierarchy \_\_\_\_\_\_\_

### Key Data structures:

Grid Map (2D Array)

The Map is presented with 0’s and 1’s where 0 is a non-walkable tile and 1 is a walkable tile

MAP = [  
  
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,],  
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,],  
 [1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1,],  
 [1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1,],  
 [1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1,],  
 [1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1,],  
 [1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1,],  
 [1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,],  
 [1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,],  
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1,],  
 [1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1,],  
 [1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1,],  
 [1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1,],  
 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1,],  
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,],  
  
]

The raycasting engine uses the map to determine visibility and rendering. Rays are cast from the player's position and projected outward until they hit a wall tile. The method converts world coordinates into grid indices and checks whether a wall exists at that location. This enables realistic line-of-sight and shading effects.

A\* Pathfinding

The same grid is used to guide AI movement. The pathfinding algorithm treats each open tile as a node in a graph. It calculates the shortest path from a start point to a goal by evaluating movement costs and avoiding wall tiles so the enemy is able to get to the player efficiently.

### Complex Functions:

### Function 1: Ray projection

It handles ray direction, grid stepping and wall collision

PROCEDURE project\_ray(angle, player, map)

angle ← AngleConvert(angle)

facing\_down ← angle ≥ 0 AND angle ≤ π

facing\_up ← NOT facing\_down

facing\_right ← angle ≤ 0.5 × π OR angle ≥ 1.5 × π

facing\_left ← NOT facing\_right

// Horizontal ray check

IF facing\_up THEN

first\_horizontal\_y ← (player.y DIV tile\_size\_y) × tile\_size\_y - 1

ELSE

first\_horizontal\_y ← (player.y DIV tile\_size\_y) × tile\_size\_y + tile\_size\_y

ENDIF

first\_horizontal\_x ← player.x + (first\_horizontal\_y - player.y) / TAN(angle)

step\_y ← IF facing\_up THEN -tile\_size\_y ELSE tile\_size\_y

step\_x ← step\_y / TAN(angle)

WHILE within\_screen\_bounds(first\_horizontal\_x, first\_horizontal\_y)

IF map.has\_wall\_at(first\_horizontal\_x, first\_horizontal\_y) THEN

horizontal\_hit ← (first\_horizontal\_x, first\_horizontal\_y)

EXIT WHILE

ELSE

first\_horizontal\_x ← first\_horizontal\_x + step\_x

first\_horizontal\_y ← first\_horizontal\_y + step\_y

ENDIF

ENDWHILE

// Vertical ray check

IF facing\_right THEN

first\_vertical\_x ← (player.x DIV tile\_size\_x) × tile\_size\_x + tile\_size\_x

ELSE

first\_vertical\_x ← (player.x DIV tile\_size\_x) × tile\_size\_x - 0.00001

ENDIF

first\_vertical\_y ← player.y + (first\_vertical\_x - player.x) × TAN(angle)

step\_x ← IF facing\_right THEN tile\_size\_x ELSE -tile\_size\_x

step\_y ← step\_x × TAN(angle)

WHILE within\_screen\_bounds(first\_vertical\_x, first\_vertical\_y)

IF map.has\_wall\_at(first\_vertical\_x, first\_vertical\_y) THEN

vertical\_hit ← (first\_vertical\_x, first\_vertical\_y)

EXIT WHILE

ELSE

first\_vertical\_x ← first\_vertical\_x + step\_x

first\_vertical\_y ← first\_vertical\_y + step\_y

ENDIF

ENDWHILE

RETURN horizontal\_hit, vertical\_hit

ENDPROCEDURE

What this function does is it calculates the direction of a ray based on the players viewing angle and determines whether the ray is face up, down, right or left. Once it has done this it will then perform 2 separate checks to see whether it is vertical or horizontal

HORIZONTAL: Calculates the horizontal intersection point and steps across the grid in horizontal increments checking each tile whether there is a collision with the wall

VERTICAL: It performs the same action as horizontal however it does it vertically – calculates the first vertical intersection and steps across the grid vertically checking for intersections until it has collided with the wall

The function returns the coordinates of the first horizontal and vertical wall hits, which are used to determine which wall the ray hits

### Function 2: Calculating Ray properties

Purpose: works out distances, distortion and rendering properties

PROCEDURE calculate\_ray\_properties(horizontal\_hit, vertical\_hit, player, angle)

IF horizontal\_hit EXISTS THEN

horizontal\_distance ← Distance\_inbetween(player.x, horizontal\_hit.x, player.y, horizontal\_hit.y)

ELSE

horizontal\_distance ← ∞

ENDIF

IF vertical\_hit EXISTS THEN

vertical\_distance ← Distance\_inbetween(player.x, vertical\_hit.x, player.y, vertical\_hit.y)

ELSE

vertical\_distance ← ∞

ENDIF

IF horizontal\_distance < vertical\_distance THEN

wall\_hit ← horizontal\_hit

distance ← horizontal\_distance

colour ← 160

hit\_vertical ← FALSE

ELSE

wall\_hit ← vertical\_hit

distance ← vertical\_distance

colour ← 255

hit\_vertical ← TRUE

ENDIF

// Correct for fish-eye distortion

distance ← distance × COS(player.rotationAngle - angle)

// Shade based on distance

colour ← colour × (70 / distance)

IF colour > 255 THEN

colour ← 255

ELSE IF colour < 0 THEN

colour ← 0

ENDIF

RETURN wall\_hit, distance, colour, hit\_vertical

ENDPROCEDURE

This function compares the distances of the horizontal and vertical intersections to determine which one is closest to the player. It then:

Selects the closest hit point

Applies the fisheye correction to eliminate the curviness of the lines when the player is looking at them

Calculates a shade value which affects how light or dark the wall is depending on the player’s distance from it.

### Raycasting

This is where we draw the ray and the walls, so it projects it as a 3D view which is displayed to the user. This is also where texture is applied to the wall and how we draw the npcs on the map so we can see them in 3D as well. To work all of this we need to use complex maths to work out the ray hit both vertically and horizontally and also we will need blitz the texture onto the walls and work out the area it has to take upon

CLASS Raycasting

PROCEDURE Constructor(game, player, map, renderer, npc)

// Sets up initial values for raycasting

self.game ← game

self.rays ← empty list

self.player ← player

self.map ← map

self.renderer ← renderer

self.objects\_to\_render ← empty list

self.npc ← npc

ENDPROCEDURE

PROCEDURE castRays()

// Casts rays across the field of view

self.rays ← empty list

rayAngle ← player.rotationAngle - (FOV / 2)

FOR j ← 0 TO NumRays - 1

ray ← NEW Ray(rayAngle, player, map)

CALL ray.cast()

APPEND ray TO self.rays

rayAngle ← rayAngle + (FOV / NumRays)

NEXT j

ENDPROCEDURE

PROCEDURE render(screen)

// Renders rays and textures to simulate 3D projection

count ← 0

FOR EACH ray IN self.rays

Distance\_safe ← MAX(ray.distance, 0.2)

LineHeight ← (tile\_size\_y / Distance\_safe) × screenHeight

LineHeight ← MIN(LineHeight, screenHeight)

draw\_start ← MAX(0, (screenHeight / 2) - (LineHeight / 2))

texture ← renderer.textures[ray.wall\_type]

texture\_width ← GET\_WIDTH(texture)

IF ray.hit\_vertical THEN

hit\_point ← ray.wall\_hit\_y MOD tile\_size\_y

texture\_column\_index ← INTEGER((hit\_point / tile\_size\_y) × texture\_width)

ELSE

hit\_point ← ray.wall\_hit\_x MOD tile\_size\_x

texture\_column\_index ← INTEGER((hit\_point / tile\_size\_x) × texture\_width)

ENDIF

texture\_column\_index ← CLAMP(texture\_column\_index, 0, texture\_width - 1)

column\_surface ← CREATE\_SURFACE(1, GET\_HEIGHT(texture), TRANSPARENT)

BLIT\_TEXTURE(column\_surface, texture, texture\_column\_index)

scaled\_column ← SCALE\_SURFACE(column\_surface, Resolution, INTEGER(LineHeight))

DRAW\_SURFACE(screen, scaled\_column, count × Resolution, draw\_start)

count ← count + 1

NEXT ray

FOR EACH npc IN game.npcs

dx ← npc.x - player.x

dy ← npc.y - player.y

angle\_to\_npc ← ARCTAN2(dy, dx)

angle\_diff ← AngleConvert(angle\_to\_npc - player.rotationAngle)

IF ABS(angle\_diff) > (FOV / 2) THEN

CONTINUE

ENDIF

screen\_x ← (angle\_diff / FOV) × screenWidth

sprite\_size ← MIN(screenHeight, (tile\_size\_y / Distance\_inbetween()) × screenHeight)

sprite ← SCALE\_IMAGE(npc.get\_current\_frame(), sprite\_size, sprite\_size)

ground\_y ← screenHeight DIV 2 + sprite\_size DIV 2

DRAW\_IMAGE(screen, sprite, screen\_x - (sprite\_size DIV 2), ground\_y - sprite\_size)

NEXT npc

ENDPROCEDURE

ENDCLASS

This puts the entire raycasting process together. It casts multiple rays across the players FOV and uses this data to:

Calculate the wall height based on the distance

Selects and scale the correct texture

Draw vertical slices to simulate depth in the game

Render NPC sprites

All together this gives an illusion of a 3D world by projecting 2D maps onto the screen

### A\* Pathfinding

Still working on it\_\_\_\_\_\_

## File Structure:

Final\_raycasting: Casts the ray and project a 3D view of the game based on the map’s layout.

Main\_final:Constructs the main game loop where everything is called there

Map\_Final:Draws the map and stops players from

NPC\_sprite:Where the enemy sprite is kept

Player:Contains all the functions to allow the player to interact with the game

Rendering:Load the texture and draws the sky and the floor

Settings:Where all the game settings are kept

Sprites:

Weapon:Where the weapon functions are and the weapon settings

Database GUI

# Technical solution

# Testing

To ensure my game functions reliably and meets its design objectives, I will carry out a structured testing strategy focused on the most impactful features that make the game runnable.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test No | Purpose of Test | Test Data Used | Expected Outcome | Actual Outcome | Timestamp |
| 1 | Test Raycasting work | Player rotates view in game | When the game is loaded it should be displayed as 3D where the walls are unwalkable and the player when looking around should continue loading the walls |  |  |
| 2 | Test If procedural generation works | Load game twice in different seeds | Load the game twice the map should have changed and everything should continue to work normally |  |  |
| 3 | Test A star pathfinding works | Enemy spawns and tracks player | When fighting the enemies the enemies should be able to reach you at a quick pace |  |  |
| 4 | Make sure enemies are attacking you | Enemy enters enemy range | When loading in the game the enemies should know where you are and should start to shoot at you instantaneously |  |  |
| 5 | Check if gun fires | Player presses right mouse click | Shoot the gun and make sure it shoots – should fire a bullet out in front of it |  |  |
| 6 | Check if ASWD works | Press A, S, W, D | When in the game press wasd separately and the player should move around  -A to move left  -D to move right  -W to move forward  -S to move backwards |  |  |
| 7 | Check if jump works | Press spacebar | Press the space bar and the player should jump in the air |  |  |
| 8 | Check if bullet doesn’t go through walls | Shoot directly at wall | Shoot the gun at a wall. It shouldnt go through it and should just intercept the wall dealing no damage |  |  |
| 9 | Check if damage correlates to where bullet is hit on NPC | Shoot head,body,legs | Shoot the gun at the enemies head, body and feet it should correlate to this damage:  -Headshot (instant kill)  -Body shot 25 damage  -Leg shot 12 damage  This can be determined by the health bar |  |  |
| 11 | Check if boss fight works | Reach stage 10 | When you have got to stage 10 there should be a WARNING that pops on the screen and a big boss with health buffs and damage buffs spawn in the game |  |  |
| 12 | Check if health bar works | Player takes damage | Shoot the player on the body and see if the damage correlates to what has been taken off from the health bar |  |  |
| 13 | Check if there are different kinds of enemies | Load game | When loaded in the game there should be multiple different enemies that have spawned with different health, damage and size |  |  |
| 14 |  |  |  |  |  |
| 15 |  |  |  |  |  |
| 16 |  |  |  |  |  |

Full system run through:

Scenario: A realistic gameplay session from start to finish:

* Game loads with procedural map and Raycasting
* Player moves, jumps, and shoots
* Enemies spawn and track player using A\* pathfinding
* Combat occurs with correct damage zones
* Health bar updates
* Boss fight triggers at stage 10
* Game continues without crashes

Test Strategy:

First I will test the functionality of my code which includes: movement (W A S D), Raycasting and collision with the wall. These are required for the game to operate smoothly and provide a 3D view for the player. I will verify that features including wall renderer, unwalkable spaces and that raycasting detects the walls intersections and displays them properly with the correct textures.

Next I would verify user input which includes: Jumping, shooting and enemy interactions. I will make sure the inputs that are outside the expected range and handled with and don’t cause the game to crash or break, helping maintain a stable experience

I would also test the AI systems, especially the A\* pathfinding algorithm. I will watch to make sure enemies navigate around the map in the most efficient and effective ways. I will also test how well it works during open spaces, around corners and during combat. This will make sure that the enemy’s AI works effectively in any scenario

For procedural generation, I will test the consistency of the map layouts and make sure they are different every time while also maintaining playable gameplay

Finally, I will test the visual and audio feedback including: health bars, sound effects and damage dealt. These will confirm that the player receives accurate feedback during the gameplay that isn’t misleading or any disadvantage.

# Evaluation

## Objectives

Objective – Raycasting MET

I managed to get my Raycasting working with minimal issues do my game is able to project a 3D view of the 2D map and enemies. This gives my game a realistic view and creates a more immersive experience when playing.

I tested the game to see if raycasting is working by rotating the players FOV and confirmed that walls and enemies appear at the correct angles and distance from the player. There are no major glitches that cause the games to crash when moving or rotating

Objective – Shop system NOT MET

Without the shop system the player has no way to unlock new abilities which may reduce the progression of the game and will get a bit boring after player for a while. This makes the game feel like the game has something missing and not the full experience

There is no interface for purchasing abilities. There is no currency system to purchase new abilities.

## User Feedback

“What did you think of the raycasting game and how was the experience”

“The 3D effect of the game is cool and made it more fun to play. The movement is good and smooth but could be more realistic like looking around I am not able to look 360 degrees. The texture and the enemy’s animations and kill sounds make the game immersive however after playing for some time there felt like there wasn’t much to do.”

Evaluation:

Is it realistic? Yes - the user played the game and gave an honest feedback

Is it fair? Yes - they liked the visuals and movement while pointing out missing features like progression

Is it representative? Yes - it reflects the current place of my project and links to my evaluation

My response: It confirms that the raycasting engine is good and functional which was my main goal but also shows the need for a more advanced gameplay e.g., progression in the game

## Future improvements

Dynamic lighting and shadows – This would make my game much more realistic and would add atmosphere.

I would have to use a distance based shadowing that works on the walls and ground and implement a light source object

Enemy line of sight detection – Makes enemies feel more like AI (Intelligent) adding more strategy in the game

I would use raycasting on the enemies to the player’s visibility and would make them react to when the see the player

PROTOTYPE:

Step 1 - Using raycasting, the enemy checks if there’s a clear path to the player. If the player is in the player’s sight the enemy will switch state and perform a run animation

Step 2 - If the player is spotted, the enemy begins chasing them around the map.

This will overall make my game more immersive and realistic. It will make it more engaging for the user and make it more fun to play.

# Appendices