B.Sc Computer Games Development, Year 3, Project I

Technical Design Document

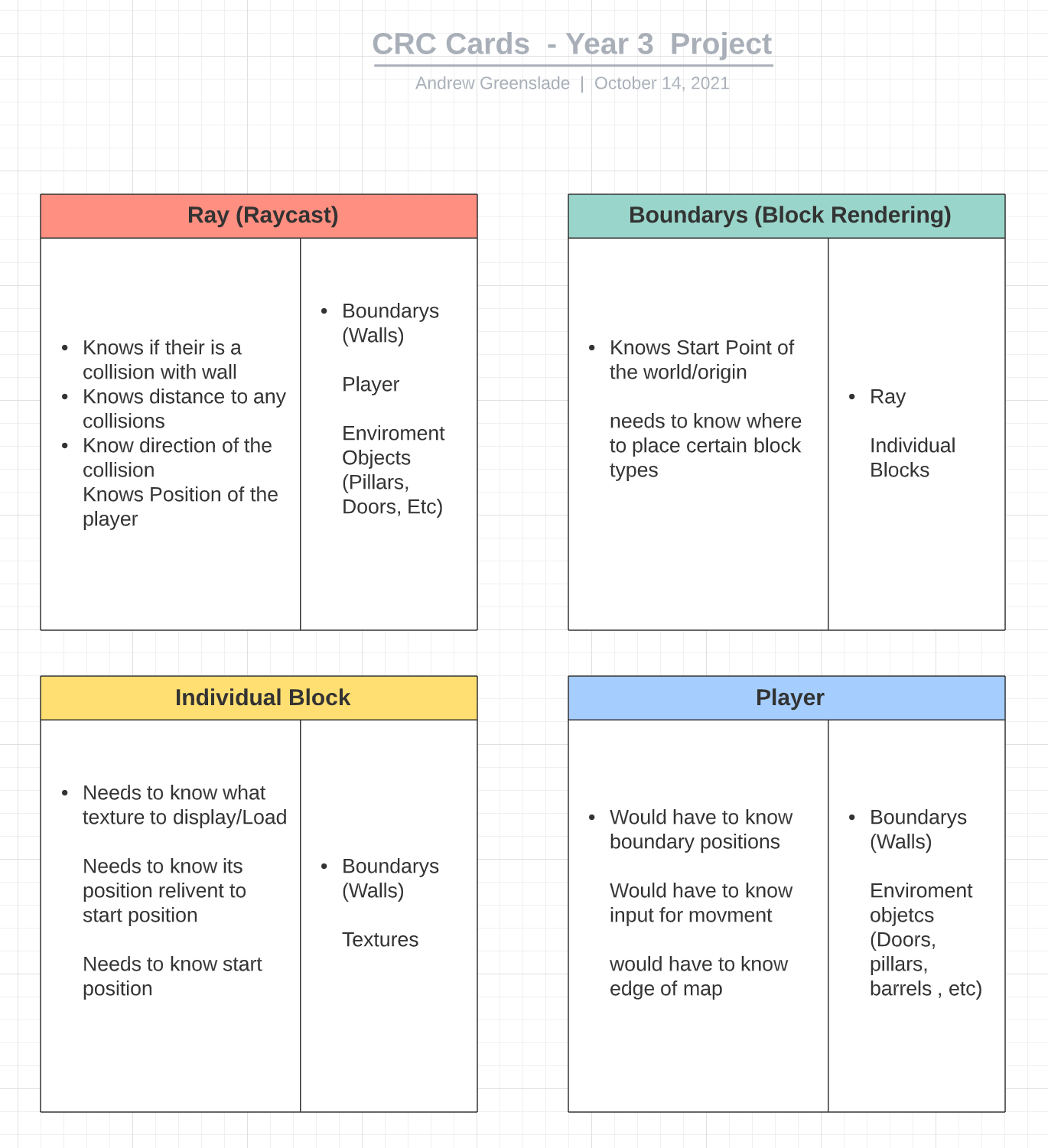
14/10/2021

**“Totally a 2D Game”**

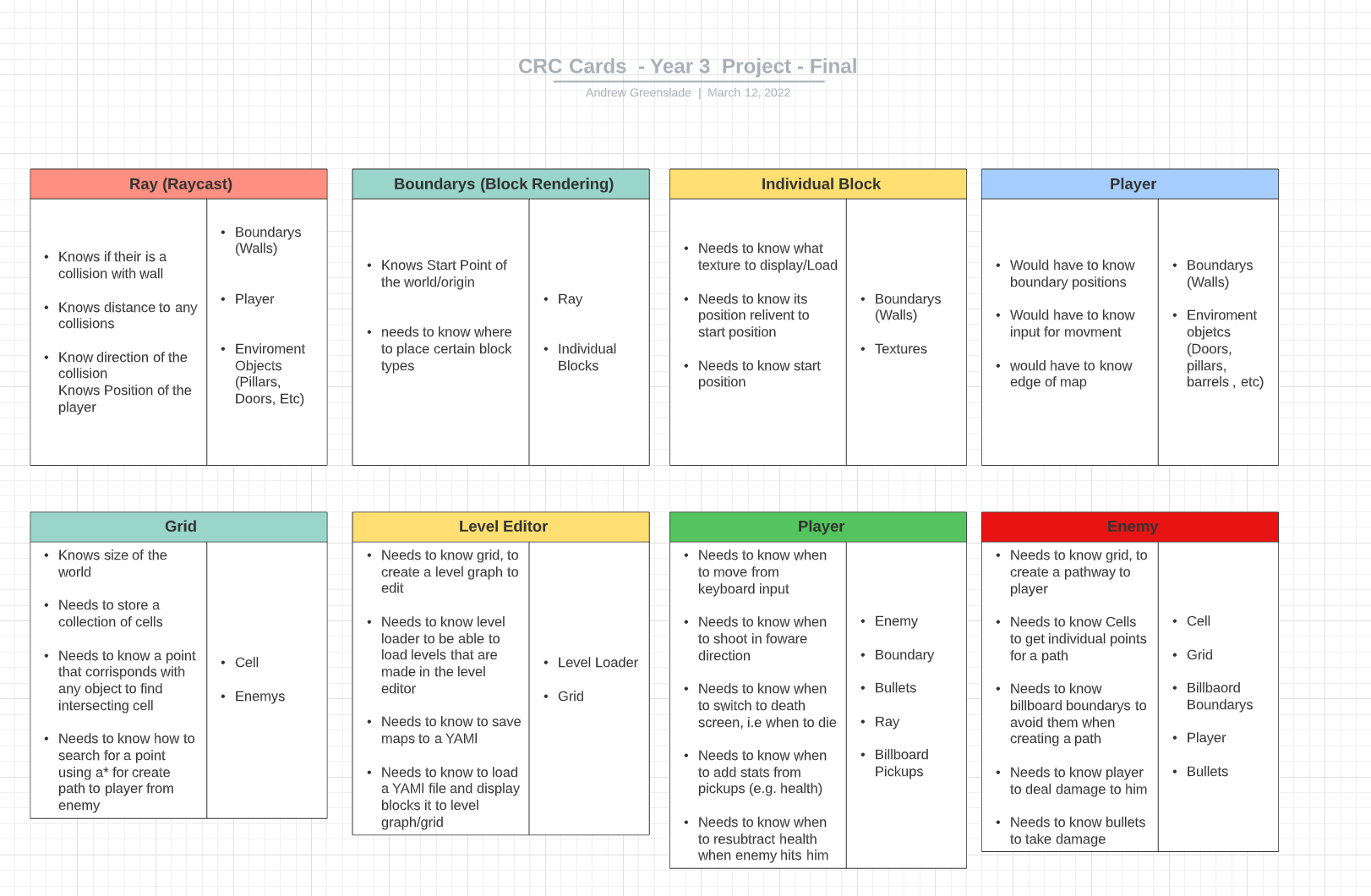
*Authors: Andrew Greenslade, Sten-Matias Kutt*

Project Supervisor: Noel O'Hara

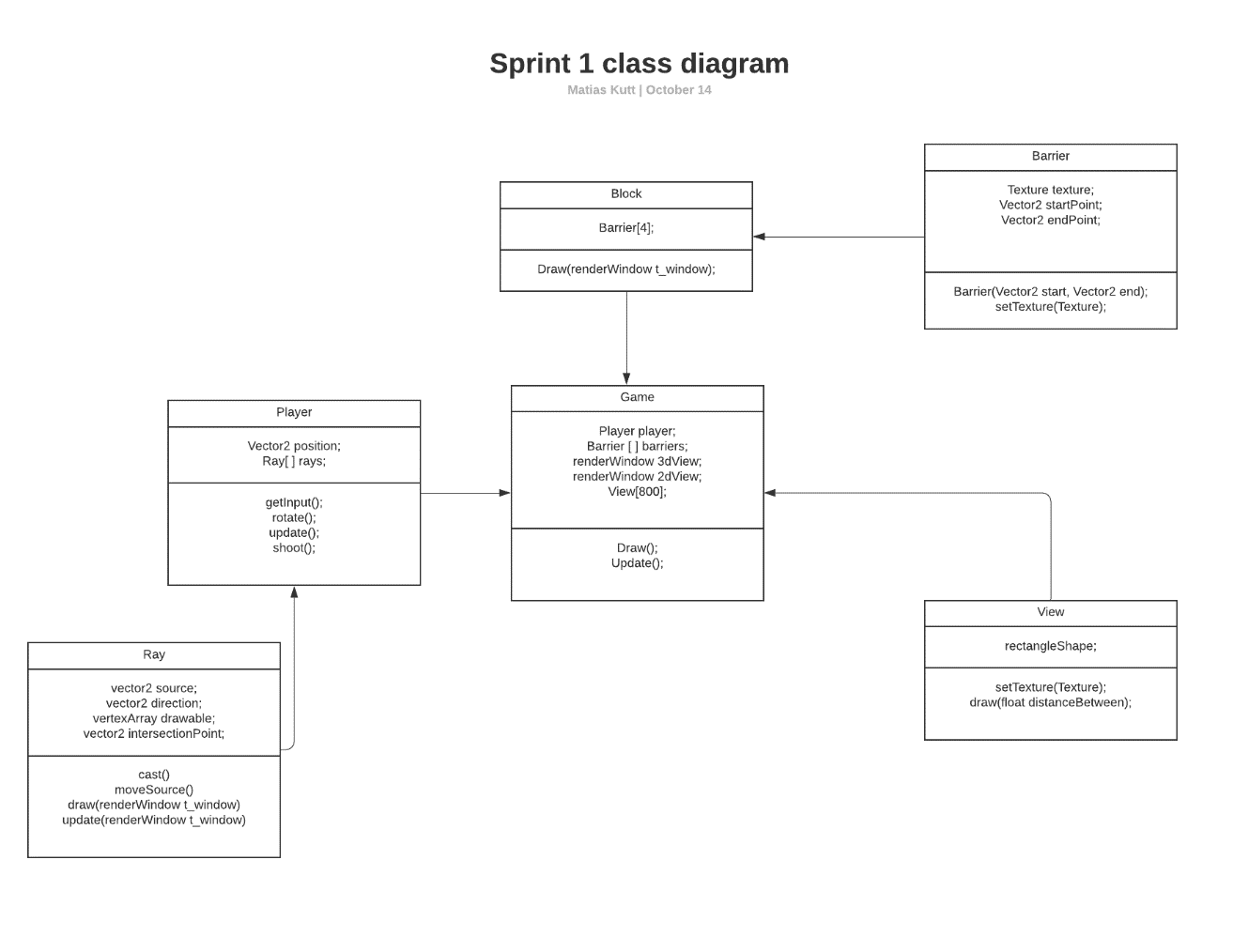
# CRC CARDS – Stage One



# CRC CARDS – Final Stage



# Class Diagram, Sprint One



# Class Diagram, Final Sprint Three

# Technology

## Existing libraries used:

* SFML: Input, graphics, audio. https://www.sfml-dev.org/
* THOR: Vector math. https://bromeon.ch/libraries/thor/index.html
* ImGUI: Realtime debug menu. <https://github.com/eliasdaler/imgui-sfml>
* YAML API: Used to read/write to/from YAML files

## Applications used

* Visual Studio 2019/2020: IDE
* GIMP: Image editing and sprite creation.
* Reaper: audio mixing.
* NVidia shadow play: Recording.
* GitHub: Version control.

## Installation

* Both SFML and THOR are set up as environment variables within windows, and linked appropriately through VS.
* ImGUI is a drag and drop feature and is simply included in the project as .cpp and .h files, and linked via headers where necessary.
* Any of the applications listed above were simply installed onto machines as recommended by the developer(s).

# Sprint 1 Report

## Feature Design

For the first sprint, the focus will not necessarily be on gameplay features, instead for our project we initially need to create a three-dimensional render based on the two-dimensional world. Since this is the main rendering method we will be using for the game, this must be completed before we begin any work on “fun” gameplay features.

For the first sprint, we need to create a ray that will cast out in a field of view an amount of times equal to the width of the window that the three-dimensional view will be rendered in. This is because each ray will correspond to a vertical slice of the render window. If the ray hits a boundary, or wall, it will measure the distance between the collision point and the source of the ray and use this distance to determine the size of the rectangle being drawn within the vertical slice on the rendering screen.

If we get this completed in a proper amount of time, we can use the same principle to begin rendering textures. SFML allows you to “split” textures and we can use this to map textures to points on a line. So, the feature that will be worked on in the first sprint is three-dimensional rendering based on a two-dimensional map.

## Ray-casting:

1. This feature is important because it is fundamental to how things will be drawn. We are using ray-casting to determine where the walls will be positioned, enemy placement, etc.
2. We are using the Thor addon library for SFML, and its polar Angle function to achieve this ray casting effect.
3. This is achieved by passing it, the origin of the ray, and a direction/unit vector.
4. This is done initially / for a arbitrary long value, until it hits a wall and uses the line intersection formula to see if it hits the line.

## Boundary's:

1. This feature is fundamental to the game as well because it is the bases for where walls should be placed for the 3D view
2. We are using SFML’s vertex array for drawing lines where the walls should be drawn in the other view
3. We are then checking if the ray hits these boundary's / walls to achieve the 3D View

## Player:

1. This is a point in the 2d View where the Ray-cast originates from.
2. We are using SFML’s isKeyPressed function to see if a key is pressed and rotate / move the player accordingly.

## Individual Blocks

1. This feature will at first determine the color of the wall, then will determine the texture later on.
2. This will determine where the walls/blocks should be layered out over the map
3. We will use YAML to load this from a external file, to allow for multiple level layouts.
4. These will be stored as an int in an array, with 0 being no block, 1 being brick, two being windows, four being escape door, etc

# Summary of planned work

This table breaks up each feature into “micro jobs” and assigns a feature to each member of the team.

|  |  |  |
| --- | --- | --- |
| **Features and tasks** | **Time (Hours)** | **Team Member** |
| Setting up game loop | 1 hour | Mati |
| Setting up a test scene with 2 lines | 30 mins | Mati |
| Setting up 2nd window | 30 mins | Mati |
| Setting up 3D math's | 2 hours | Mati |
| Setting up Player movement | 30 mins | Andrew |
| Setting up YAML loading | 2 hours | Andrew |
| Set up raycasting of environment | 2 hours | Mati |
| Set up a tutorial Level for bug testing | 1 hour | Andrew |
| Set up Block object for different colors/textures | 2 hours | Andrew |

# Work completed

This table indicates how long each part/feature of the sprint took to do, and also stated how long we originally estimated to do it and who did the feature.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Features and tasks** | **Time (Hours)** | **Actual Time taken** | **Team Member** | **Completed** |
| Setting up game loop | 1 hour | 45 minutes | Andrew | Y |
| Setting up a test scene with 2 lines | 30 mins | 1 hour | Mati | Y |
| Setting up 2nd window | 30 mins | 15 minutes | Mati | Y |
| Setting up 3D math's | 2 hours | 4 Hours | Mati | Y |
| Setting up Player movement | 30 mins | 30 mins | Andrew | Y |
| Setting up YAML loading | 2 hours |  | Andrew | N |
| Set up raycasting of environment | 2 hours | 8 Hours | Andrew/Mati | Y |
| Set up a tutorial Level for bug testing | 1 hour |  | Andrew | N |
| Set up Block object for different colors/textures | 2 hours | 3 Hours | Mati | Y |

## Sprint 1 Video Link:

<https://www.youtube.com/watch?v=nd6iYhMTw74>

## Uncompleted tasks

* Tutorial Level
* YAML loading

These tasks were left uncompleted as we realized the rendering system with textures would require more time than we anticipated initially. We also ran into issues with both our algorithm that determined distance to the closest wall, and our texturing system. Both features were deemed more important than a tutorial level and YAML loading. Both features will be worked on during the second sprint.

# Sprint 2 Report

## Feature Design

The features in sprint 1 were the skeleton of the game, in sprint 2 we plan to work on some of the fun gameplay elements that make the game more than just a simple proof of concept. Additionally, we will aim to complete the unfinished features from sprint 1 during this sprint.

## Sprites

1. A sprite rendering system will be the basis for any additional decorations we want to add to the level.
2. It will also provide the ability to render enemies.
3. It will provide the ability to render effects like explosions or bullets.

## Player abilities

1. The player should have the ability to shoot.
2. The player should be able to pick up consumables and use them, giving the player diverse benefits.
3. The player should be able to reload and run out of ammunition.

## Level Generation

1. We will create an initial level with an enemy that follows the player, alongside obstacles
2. The level design will be grid-based and have two different kinds of obstacle, passable and impassable.
3. The initial level will not have an exit door as it will be temporary

## Enemies

1. We will create an initial type of basic enemy, with no diverse AI or pathfinding. This enemy will simply follow the player around.
2. This feature requires the sprite rendering system to be implemented first.

# Summary of planned work

This table breaks up each feature into “micro jobs” and assigns a feature to each member of the team.

|  |  |  |  |
| --- | --- | --- | --- |
| **Features and tasks** | **Time (Hours)** | **Team Member** | **Completed** |
| Static sprite rendering | 3 | Mati | Y |
| Dynamic sprite rendering (enemies, effects) | 2 | Andrew | Y |
| Player shooting | 30 mins | Mati | Y |
| Player consumables | 1 Hour | Mati | Y |
| Player animations(firing) | 2 Hours | Mati | Y |
| Basic enemy logic | 2 hours | Andrew | Y |
| Level generation | 3 hours | Andrew | Y |
| Player ammo system | 1 hour | Mati | Y |

## Work completed

This table indicates how long each part/feature of the sprint took to do, and also stated how long we originally estimated to do it and who did the feature.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Features and tasks** | **Time (Hours)** | **Actual Time taken** | **Team Member** | **Completed** |
| Static sprite rendering | 3 | 4 | Mati | Y |
| Dynamic sprite rendering (enemies, effects) | 2 | 4 | Andrew | Y |
| Player shooting | 30 mins | 25 mins | Mati | Y |
| Player consumables | 1 Hour | 55 mins | Mati | Y |
| Player animations(firing) | 2 Hours | 1 hour 45 mins | Mati | Y |
| Basic enemy logic | 2 hours | 3 hours | Andrew | Y |
| Level generation | 3 hours | 4 hours | Andrew | Y |
| Player ammo system | 1 hour | 1 hour | Mati | Y |

## Sprint 2 Video Link:

<https://www.youtube.com/watch?v=Z99MyVG015U>

# Sprint 3 Report

## Feature Design

For the final sprint, we will finalize the sprite rendering system and level generation system. We will also focus heavily on creating the pathfinding required for the game’s enemy, along with creating different types of sprites in our rendering system.

The large feature that must be completed during this sprint is the pathfinding, the other features to be completed are small gameplay elements that just make the game more fun and playable.

## Sprites

* Create a sprite system that makes the sprite face the player (billboarding)
* Create a sprite system that can rotate independently of the players' position in the world, as opposed to billboarded, we could do this with an enumerator in a generic sprite class
* Both sprite types should be able to move in the world, to allow us to attach it to an NPC

## Pathfinding:

* The A\* portion of the project should be completed during this sprint, along with each of the features that entails. i.e., debug mode rendering, impassable areas.
* A debug menu should be implemented during this stage, ideally, we would like to have it running in real time to allow the player to disable/enable various features during gameplay such as texture rendering, sprite rendering etc.

## Map Loading:

* YAML level loading should also be completed during this sprint, as it was not completed during the previous one.

## Map Editor:

* A map editor that allows the user to change what is stored in tile. This includes the player, enemies, walls, sprites, etc.

## Map Design:

* We added multiple levels to the game, the exact amount I added was 3, and then two levels from the level editor, to demonstrate saving, loading, etc.

# Summary of planned work

This table breaks up each feature into “micro jobs” and assigns a feature to each member of the team.

|  |  |  |  |
| --- | --- | --- | --- |
| **Features and tasks** | **Time (Hours)** | **Team Member** | **Completed** |
| Billboard sprite rendering | 3 | Mati | Y |
| Rotating Sprites (Pickups) | 2 | Mati | Y |
| A\* pathfinding (On Load) | 6 | Andrew | Y |
| A\* Pathfinding (Dynamically throughout game) | 2 | Andrew | Y |
| A\* DEBUG (Showing Route) | 1 | Mati | Y |
| Debug Menu (To show various stages of rendering) | 3 | Mati | Y |
| YAML map loading | 4 | Andrew | Y |
| Multiple Levels | 3 | Andrew | Y |
| Level Designer | 4 | Mati & Andrew | Y |

## Work completed

This table indicates how long each part/feature of the sprint took to do, and also stated how long we originally estimated to do it and who did the feature.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Features and tasks** | **Time (Hours)** | **Actual Time taken** | **Team Member** | **Completed** |
| Billboard sprite rendering | 3 | 4 hours | Mati | Y |
| Rotating Sprites (Pickups) | 2 | 3hours 30 mins | Mati | Y |
| A\* pathfinding (On Load) | 6 | 6 hours | Andrew | Y |
| A\* Pathfinding (Dynamically throughout game) | 2 | 1 hour 45 mins | Andrew | Y |
| A\* DEBUG (Showing Route) | 1 | 30 mins | Mati | Y |
| Debug Menu (To show various stages of rendering) | 3 | 2 hours 30 mins | Mati | Y |
| YAML map loading | 4 | 4 hours 30 mins | Andrew | Y |
| Multiple Levels | 3 | 2 hours | Andrew | Y |
| Level Designer | 4 | 6 hours | Mati & Andrew | Y |

## Sprint 3 Video Link:

<https://www.youtube.com/watch?v=aitUb96oovA>

# Technical achievements

## Mati

The most obvious technical achievement in this project is the creation of a pseudo-three-dimensional rendering system using only SFML. This was a particularly challenging feature to incorporate as we had never done anything remotely like this, and there was extraordinarily little documentation available online that related directly to creating this kind of feature in SFML. The documentation and examples we did find were written in other languages, using other graphics libraries, or using complex math that would take a while to wrap our heads around.

In the end, we created a system that made practical sense, was explainable, efficient and produced a fantastic looking result, with the help of various online resources, we were able to create a rendering system from scratch that fulfilled the conditions we set for ourselves during the planning phase of this project.

The rendering system can be broken down into two parts, the distance-based rendering of boundaries, and the texturing.

Distance based rendering:

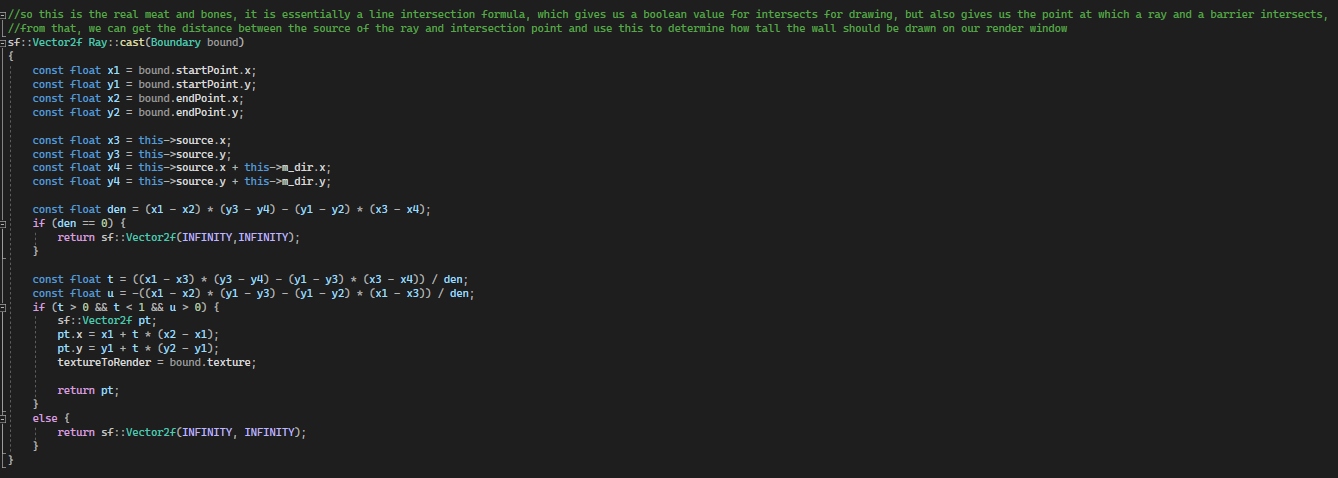
To summarize, the raycasting is composed of the following components, the source (player), the direction of the player, the rays, the boundaries, and the view.

The source is simply the starting point for each ray, it is attached to the player and moves around the world space using the arrow keys (or WASD)

The direction is simply a normalized vector to determine player heading.

The rays are composed of two parts, the source, and the direction, and they cast out infinitely in each direction, in our case, we have 160 rays cast out at a total angle of 60 degrees, giving our player a field of view of 60 degrees.

The boundaries are simply lines with a start and end point, we use a line intersection formula to determine whether a player’s ray intersects with a boundary or not. If it intersects, we store some data within the ray object that gets pulled from the boundary object, particularly an integer that is attached to the boundary object, alongside the distance between the source of the ray and the point of intersection. The former is used to texture a rendered three-dimensional wall, while the latter is used to determine how tall a wall segment should be drawn in our rendered world.



Each ray corresponds to an sf::rectangle object that is five pixels wide, since we have 160 rays, that means our render window is 800 pixels wide. We use the stored distance of intersection that each ray has, to stretch or shrink the height of the rectangle object, if it is further away, it is smaller, and vice versa. This is all we needed to have a basic, flat colored three-dimensional rendering system that was based on a two-dimensional world.

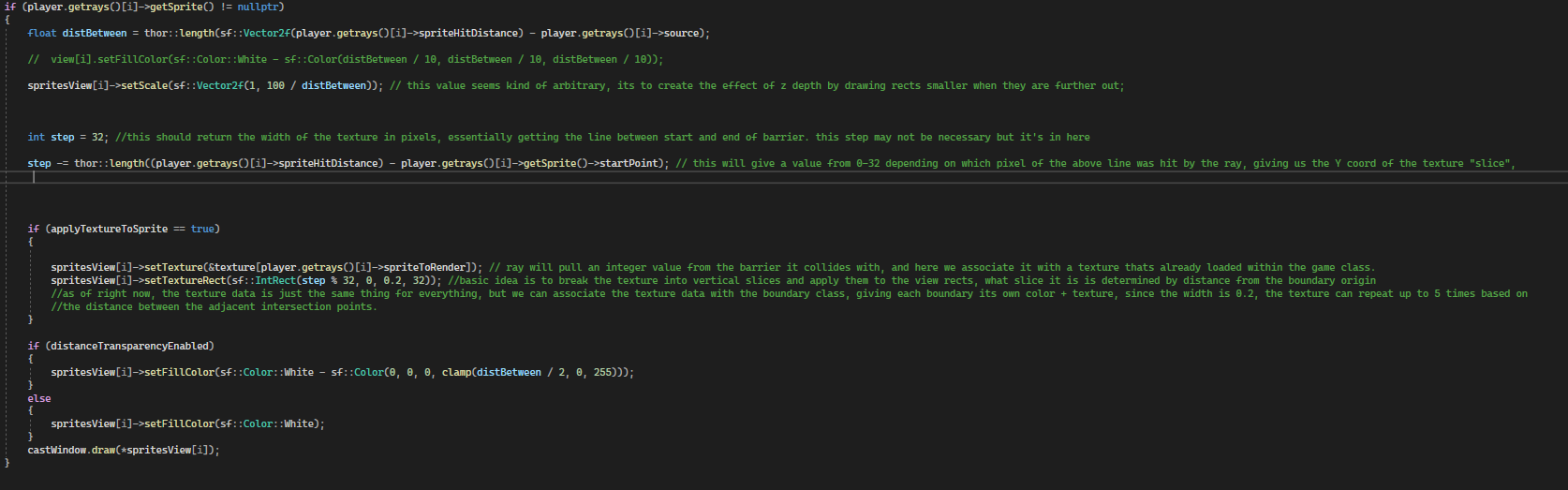
The next component of the renderer was the texturing, and this feature had no existing examples available anywhere online. The system I ended up developing was composed of the following components: The stored point of intersection, an integer value attached to a boundary object, and an array of sprites.

When a ray intersects with a boundary, the point of intersection was stored in the ray object as a vector, alongside this, the ray also took an integer value, called texture, from the boundary it intersected with.

In the game file, there is an array of sf::textures, each corresponding to the integer values that are attached to the boundary object, since each ray corresponds to a single sf::rectangle object for the purpose of rendering, we can use the stored intersection point to determine how far the source of the boundary is from the intersection point, allowing us to use a simple repeating texture to determine what texture should be applied to each rectangle object. For example, if the point of intersection is 5 pixels from the source, we apply the fifth column of the sprite to the rectangle object, repeating this for each point of intersection.

While the above example is what I did originally, this quickly became inefficient as each frame required the application to retexture and recalculate 800 rectangle objects, and when we implemented sprite rendering, this doubled to 1600 updates per frame, absolutely destroying our performance. To combat this, I cut the number of rectangles down to 160, and did the same with the rays. This quadrupled our performance.

With the above change, I also had to change the texturing code, instead of each column of the sprite being directly associated with a point of intersection, each column could now be repeated up to five times, allowing the texture to resize based on the distance between the player and boundary, producing what is a pseudo LOD feature, saving performance if the player is a certain distance away. This feature is by far the thing I am most proud of, as the final product looks fantastic in motion, and is extremely efficient when compared to the first implementation.



The sprite rendering system is built off the above implementation, the differences are, the sprites render on a different layer of rectangle objects, allowing us to render both a wall and a sprite on any given section of the screen, and the sprites are dynamic, meaning they move within the world space, there are two kinds of sprites, billboarded, which rotate to face the player each frame, this is used for static objects like bushes, and rotating sprites, which rotate freely in a space, this was used for pickups.

# Technical achievements

## Andrew

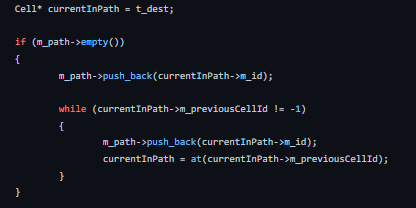
Throughout this project I have made many achievements in learning in topics, such as but not limited to ray-casting, pathfinding, map design, map editors, etc., The main technical achievement I want to discuss in this report Is Pathfinding:

Inserting image...The algorithm used, A\*, is an algorithm that allows the program to search for an optimal route for finding a goal from a destination, where the grid/graph is weight based. In my implementation, when there is a certain number of cells between the enemy and player, the enemy re-runs through the algorithm, to allow the enemy to re-find an optimal route. This is done so the player can’t zig-zag his way and make the enemy take longer to catch up with them. In terms of code, the enemy stores the route in their class, so each enemy has a different path that is easily traversable. This is stored in a vector of integers, for storing the ids of the cells that they want to traverse like this:

Text

Description automatically generatedThe enemies will also try to avoid billboard sprites, like bushes for example as they have a higher weight then a empty cell, which has a weight of 1. the overall code for this algorithm looks like this

The way I add to the path is the following, I get the goal path, and then work backwards by adding the ids of the cell to the path vector, and then set current cell to the previous cell and keep looping till it is the previous equals minus one, as this is the value for when the cell doesn’t have a previous cell. The path is cleared when a\* is run, and all cells previous value are set to minus one, and the cells traversal Boolean is set to false, so the grid is reset. This image shows the path being assigned after a\*.



Text

Description automatically generatedThe enemy traverses these cells by moving from the center of one cell to the next center, till it reaches the endpoint or like stated above gets too far away from the player, then a\* is rerun. This image illustrates traversal of the path and running of the a\*. This works by if the requirement for traversal is not calculated, the code runs as the Boolean foundNextPos is false, then sets to true after the calculations are done. It then runs the adding of velocity to the enemy and setting its new position. It also updates the position of the billboard boundary on the 2d map view. If the player is 1 unit away from the next cell, it reruns the code for calculating the next cells position and velocity. Then removes the current cell if that’s true. Next it checks if the path is empty and if so re-runs a\*.

The A\* code uses two functions that I have created which is at() and getCellPlayerIsIn(), which I will discuss here.

at():

is a function that will return a cell based on a given ID, so if I wanted the cell at column and row 2, I would give it ID 23, this is done because my cells are stored in a 2D array, so a conversion is needed to get a particular cell i.e the x and y coordinate are needed to access it. This function in code looks like :

Text

Description automatically generated

getCellPlayerIsIn():

is a function that can return a id of a cell that a point is located in mainly used for the player, i.e if a point it intersecting will cell return that cell’s id, this means I have to look through each cell, this function looks like this in code:

### NOTE: the name does imply that this would only get the cell that the player was in, but we refactored it, to allow us to use any point, as it was needed for other use cases, in the project like the enemy’s movement code, etc.

Text

Description automatically generated