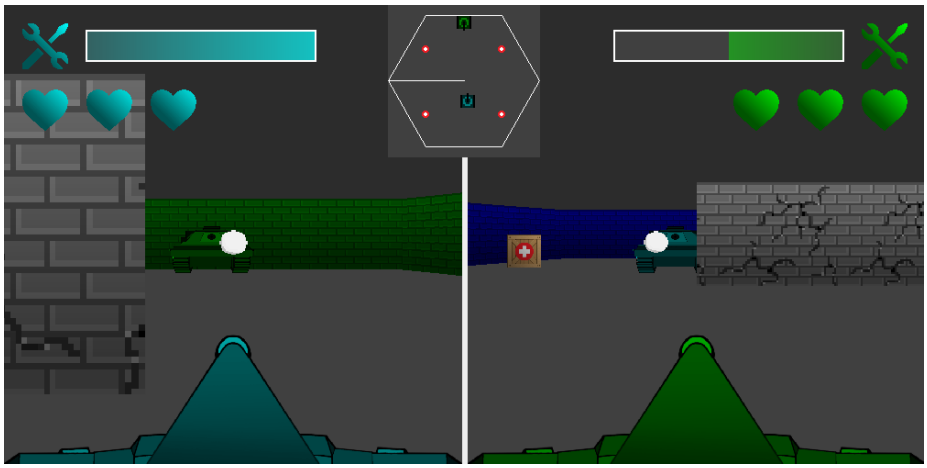
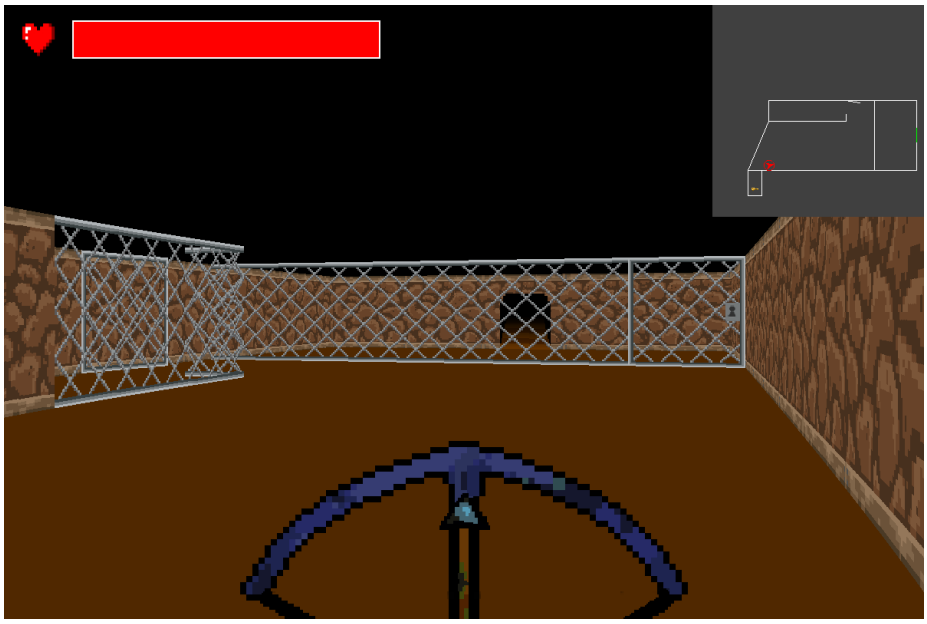
CSC 413 – Term Project Documentation

Tanks 3D and Dungeon Crawler

Alex Wolski
CSC 413 Section 2
Fall 2018



https://github.com/csc413-02-fa18/csc413-tankgame-AlexWolski



https://github.com/csc413-02-fa18/csc413-secondgame-AlexWolski

Table of Contents

1 Introduction

1.1 Project Overview

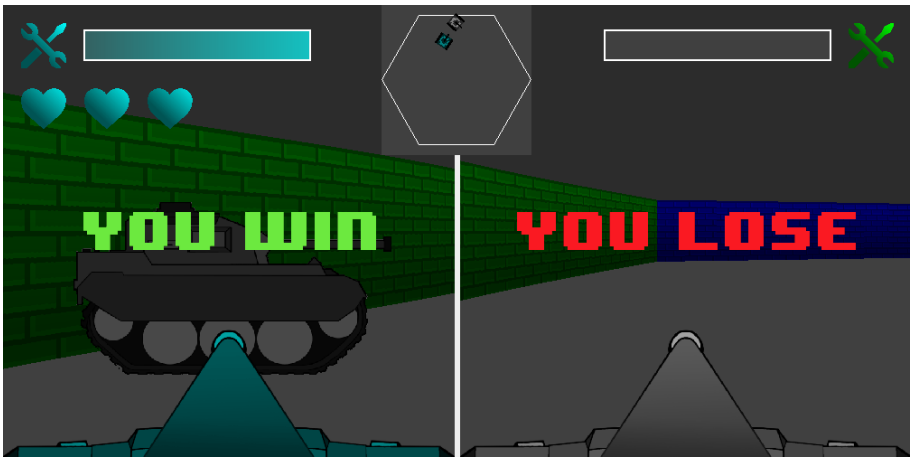
This project consists of two Psuedo-3D games in the style of Wolfenstein 3D and Doom. Using the "ray-casting" technique, these games projects a 2D world to a 3D image.

Wolfenstein 3D
Source: https://maniacsvault.net

Doom
Source: https://www.destructoid.com

1.2 Introduction of Tanks 3D

Tanks 3D is a competitive two player game. The screen is split into two to show each player's perspective. Each player controls a tank which can move and shoot bullets. The objective is to destroy the other player's tank. This is done by hitting the opponent tank with bullets until their health is depleted. Each time a tank's health is depleted, they lose a life. Once a tank loses all of its lives, that player loses and the other player wins. There are health crates scattered around the map that players can restore their health.



1.3 Introduction of Dungeon Crawler

Dungeon Crawler is a single player puzzle game. The objective is to overcome the obstacles and reach the exit. The obstacles include locked doors and blocked entrances. In order to overcome these obstacles, the player must find keys and weapons. Each time the player reaches an exit, they progress to the next level. After clearing the final level, the player wins.

2 Development Environment

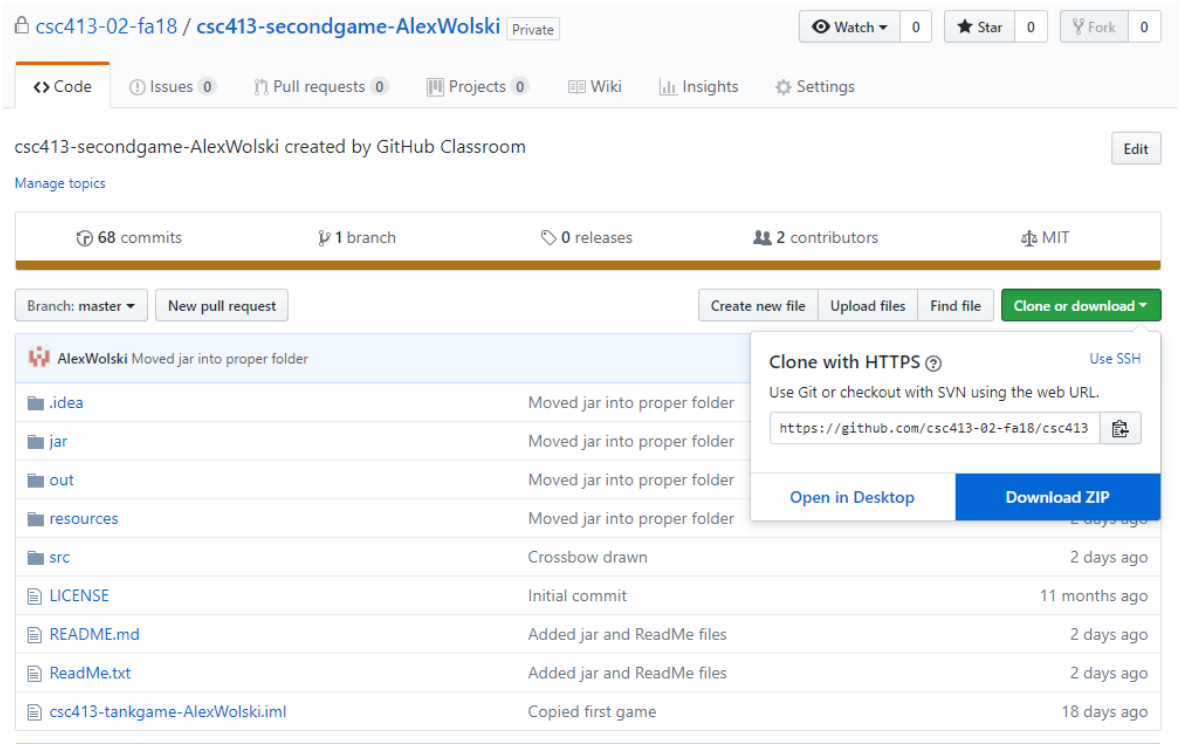
This project was developed using the **IntelliJ IDEA 2018.2 IDE** along with **JDK 1.8.0_171.** No external or special libraries were used in this project.

3 How to Import the Projects

To import the games, you need to first download the code or clone it through git.

A direct download option is available on the git hub page for either game:

https://github.com/csc413-02-fa18/csc413-tankgame-AlexWolski
https://github.com/csc413-02-fa18/csc413-secondgame-AlexWolski



Alternatively, the games can be cloned using git. Use this console command to clone the first game:

git clone https://github.com/csc413-02-fa18/csc413-tankgame-AlexWolski

Or this command for the second game:

git clone https://github.com/csc413-02-fa18/csc413-secondgame-AlexWolski

4 How to Build the Projects

Before building the jar file, make sure you have both the necessary:
https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

To build the games, you could import the project into the IDE of your choice and use it to generate the jar file.

Alternatively, you can build the jar through the command line. If you are on windows, first navigate to inside the project folder using the **cd** command.

To build the first game, run the command:
cd <project folder\src\Tanks3D>

For the second game, run the command:
cd <project folder\src\DungeonCrawler>

Then, include the JDK directory to the list of program paths:
path "<path of jdk 1.8.9>\bin"

If you are on a 64 bit machine and installed the 171 update of the jdk, run this command:
path "C:\Program Files\Java\jdk1.8.0_171\bin"

After that, compile all of the classes using this command:
javac *.jar

Finally, create the manifest and jar files using this command:
jar cvfe <Name of jar>.jar GameManager *.class

5 How to Run the Projects

5.1 Launching the Games

Before running either game, make sure you have JRE version 1.8.0:
https://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html

To start either game, you can simple double click on the jar file.

Alternatively, you could run the jar file through the command line. On windows, open the command line and use the **cd** command to navigate to the folder containing the jar file and resource folder. If you did not build your own jar, you can use the provided jar in this directory:

<Project Name>\out\artifacts\<Project Name_jar>

Then run this command:

java -jar <jar file>

5.2 How to Play

Tanks 3D Controls

Player 1:

W = Move Forward
S = Move Backwards
A = Turn left
D = Turn Right
Space = Fire

Player 2:

Up = Move Forward
Down = Move Backwards
Left = Turn left
Right = Turn Right
Space = Fire

Tanks 3D Objective

Avoid getting hit by the other player.

Try to hit the other player with bullets until they run out of lives.

Dungeon Crawler Controls

W = Move Forward
S = Move Backwards
A = Move Left
D = Move Right
E = Interact
Space = Fire weapon

Mouse left/right = Look left/right

OR

Left/right arrows = Look left/right

Dungeon Crawler Objective

Exit the level by reaching the stairs.

Collect keys to unlock doors in your way.

Collect weapons to break down blocked pathways.

6 Assumption Made

I made a lot of assumptions in the design of the games. The requirements provided by the client were brief and left a lot of the decision making to me. So, I had to make a lot of assumptions as to what the client would want.

The biggest most important design assumptions I made was that the client would be happy with a pseudo-3D game. Judging from the instructions, the client may have expected a 2D game. However, I believed that a 3D game would immerse the player more and give a better overall experience. I also assumed that I would have enough time to develop a 3D game instead of a 2D game.

Another big design assumption I made was with the mechanics and theme of the second game. The client provided me with a list of game ideas they would like to have made. But I wanted to make a more original game and assumed that the client would be happy with the results.

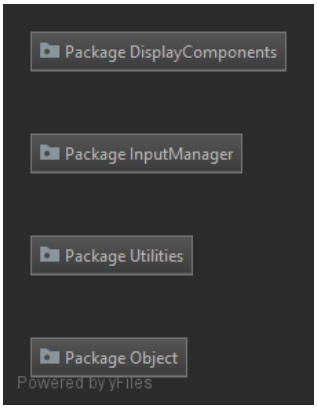
There were also several assumptions I made during the implementation of the game. I assumed that the game may be updated in the future and have content added or changed. To prepare for this, I made the games as organized and well documented as I could. The programs have high cohesion with many separate classes to perform certain tasks. This makes it easy to identify and modify classes. I also commented a lot of the code to make it easier to people to understand it in the future.

The last assumption I made was with the level design. Currently, neither of the games have a level editor. The levels are hard coded and loaded at start up. However, I made the assumption that new levels and a level editor would be added in the future. To accommodate this possibility, I designed the game objects in a dynamic way. Each wall object can take a texture file, coordinates for it starting and ending position, and booleans that determines its behavior. This allows for high customizability for the wall and prepares it for use in a level editor.

7 Class Diagrams

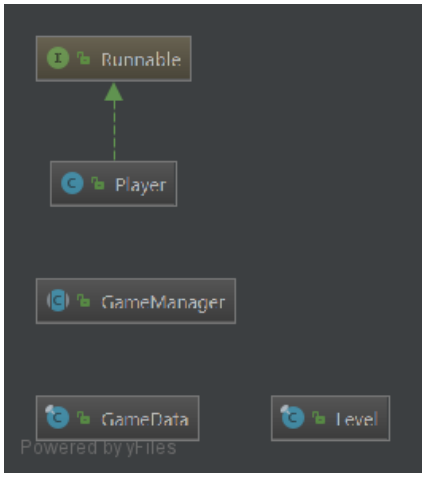
7.1 Tanks 3D Class Diagram

Packages:



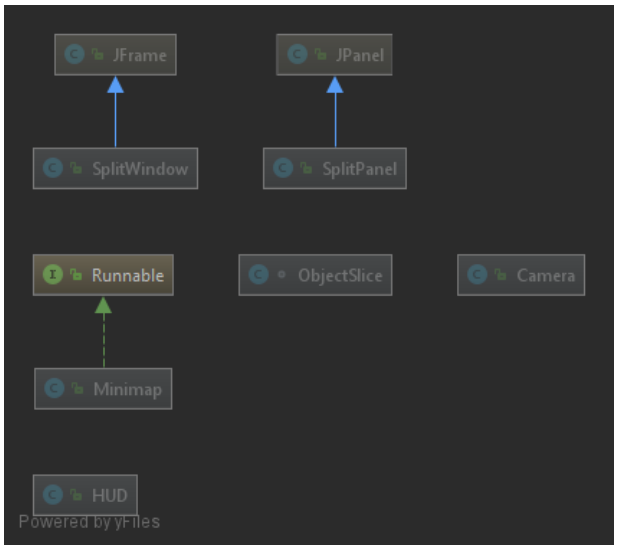
The Object package contains the wall and entity classes. The Utilities package is for tools non-specific to this game. It contains a math class and an image manipulating class. The Input Manger package is for classes relating to processing the keyboard inputs. Finally, the Display Components class contains all of the classes involved in rendering the frame and pushing it to the screen.

Tanks 3D:



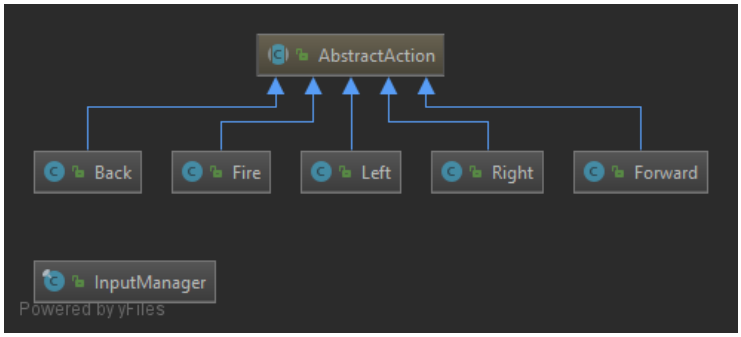
These are the classes in the base package. Player implements Runnable so that each player's screen can be rendered in a different thread. GameManger is the entry point for the program and controls the entire game. Level loads and stores the level, while GameData stores important information to be shared among all of the classes.

Display Components:



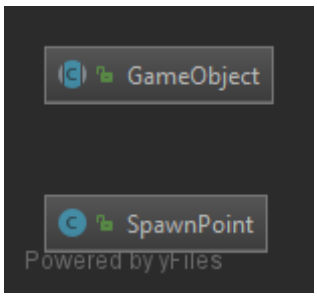
SplitWindow extends JFrame while SplitPanel extends JPanel. These two classes work together to manage the display and push the completed frame to the screen. SplitPanel contains a separate buffered image for both players and the minimap. All three are rendered at once using threads and displayed. Minimap implements Runnable so that it can render while both of the players' screens are redering. Camera has ObjectSlice objects which contains the data needed to draw a column of pixels.

Input Manager:



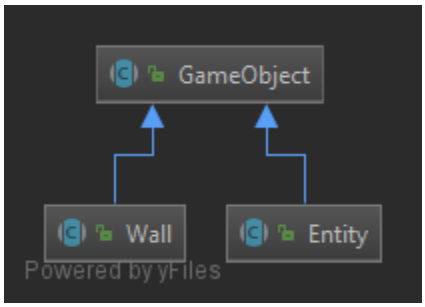
InputManger binds all of the needed keys to AbstractAction classes. These classes are called every time their corresponding key is pressed, and call a method in Player.

Object:

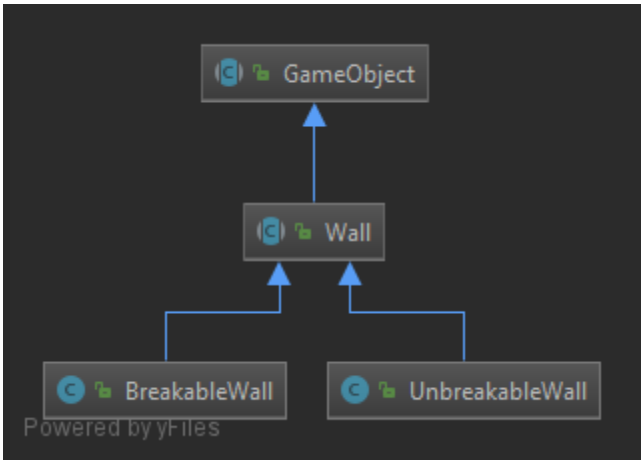


The two objects are GameObject (which includes walls and entities) and SpawnPoint. SpawnPoint is an object created when the level is loaded, and stores the information on the state each player is spawned in.

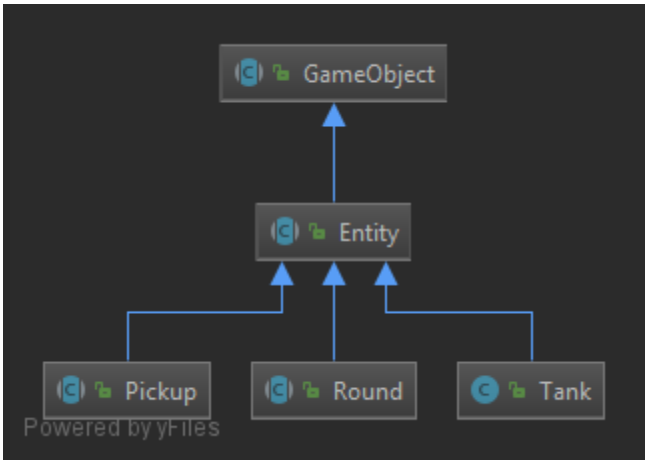
Game Object:



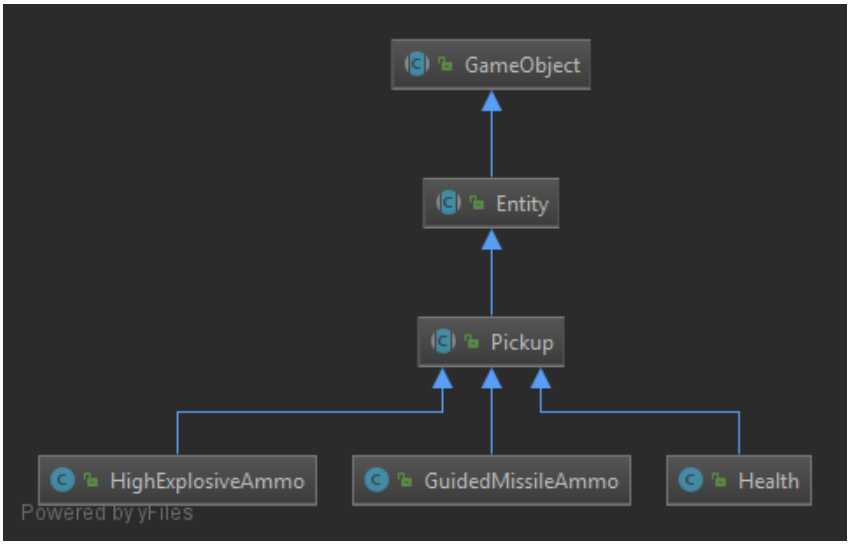
Wall:



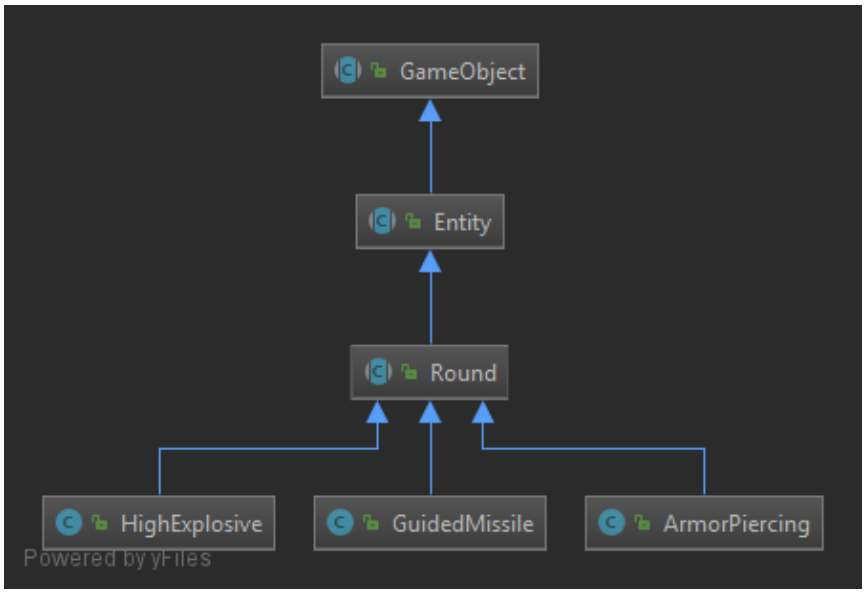
Entity:



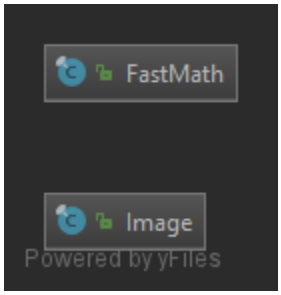
Pickup:



Round:

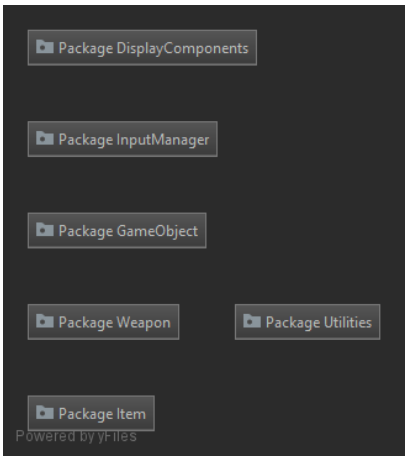


Utilities:



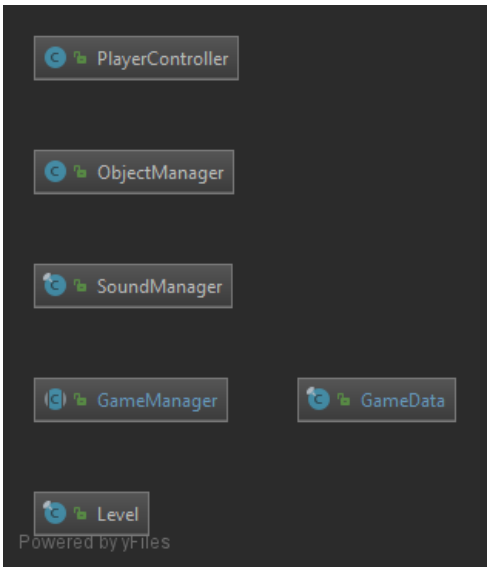
7.2 Dungeon Crawler Class Diagram

Packages:



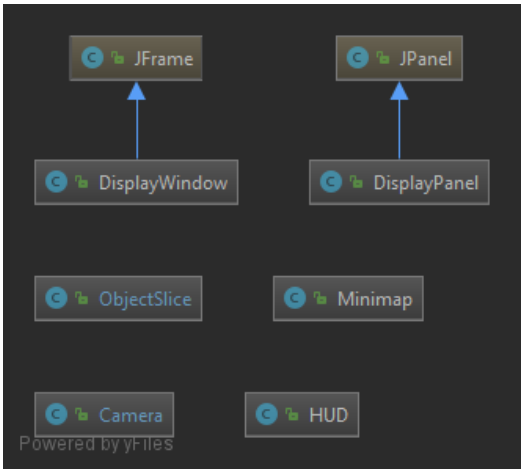
The additional packages in Dungeon Crawler is the Weapon and Item packages. These contain items that players can pick up and weapons the player can use.

Dungeon Crawler:

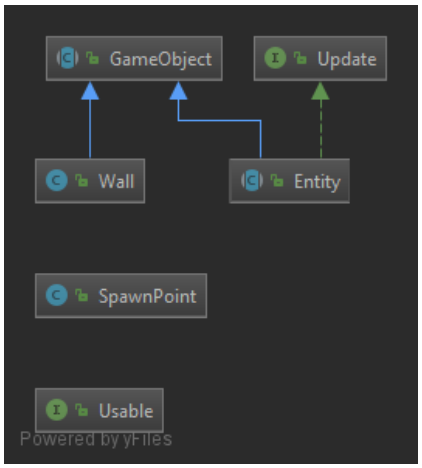


The SoundManager class loads and manages the audio effects in the game.

Display Components:

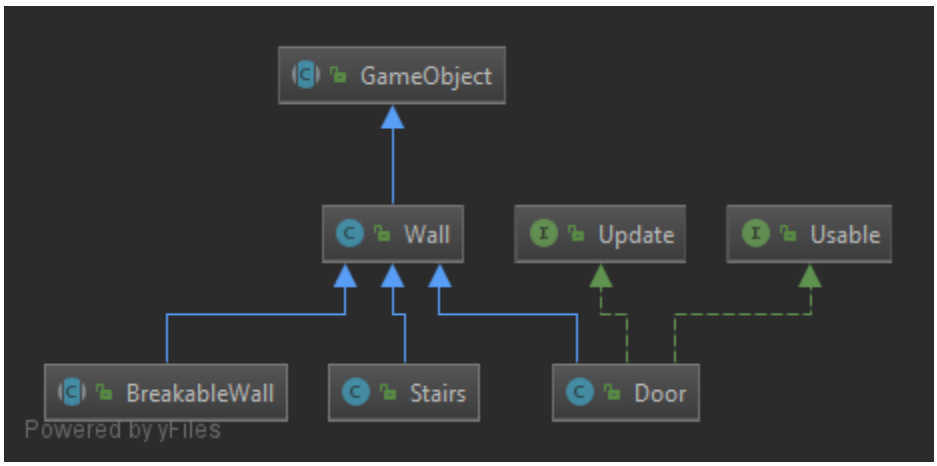


Game Object:



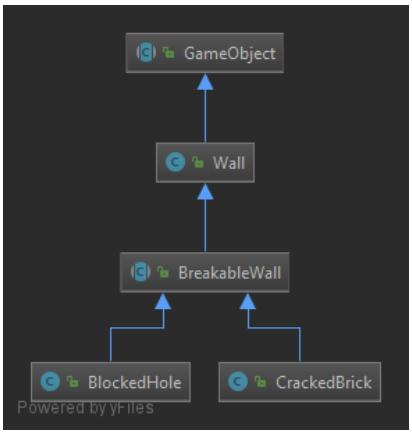
Similar to the GameObject package in Tanks 3D. But this package is effectively a combination of the Object and GameObject packages from Tanks3D. The Usable interface is for objects that the player can interact with. And the Update interrace is for objects that require a constant update in the game loop.

Wall:

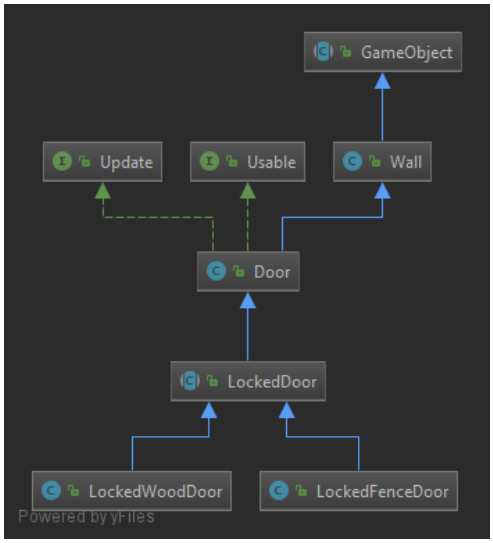


Door implements Update and Usable since the player can open the door, then the door is updated every frame as it opens.

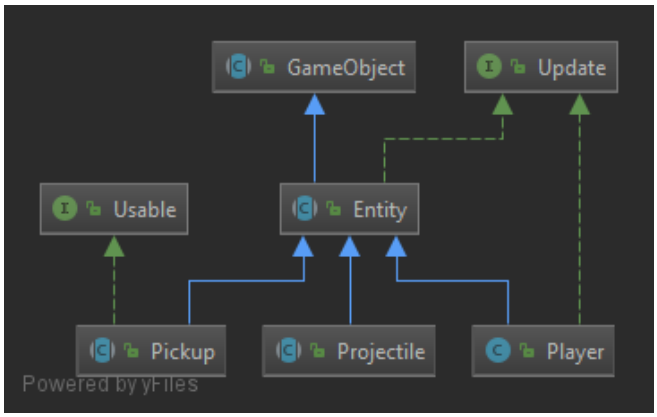
Breakable Wall:



Door:

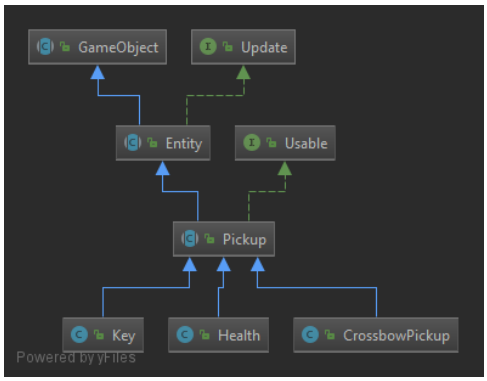


Entity:

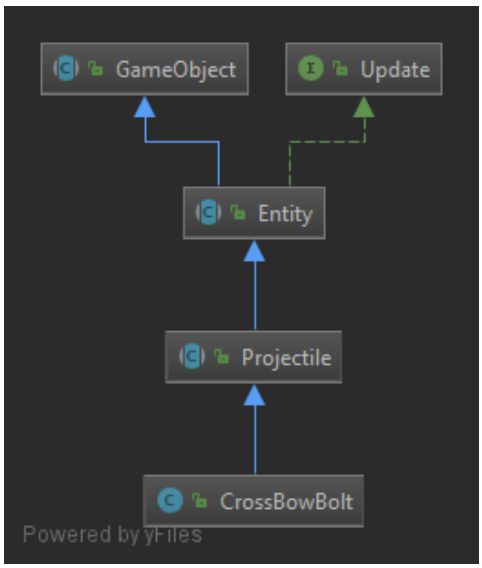


Entities implement Update because all entities have a speed and could move each frame. Player implements and overloads Update because in addition to movement, Player needs to manage some game logic. Pickup implements Usable because the player can interact and pick up any Pickup.

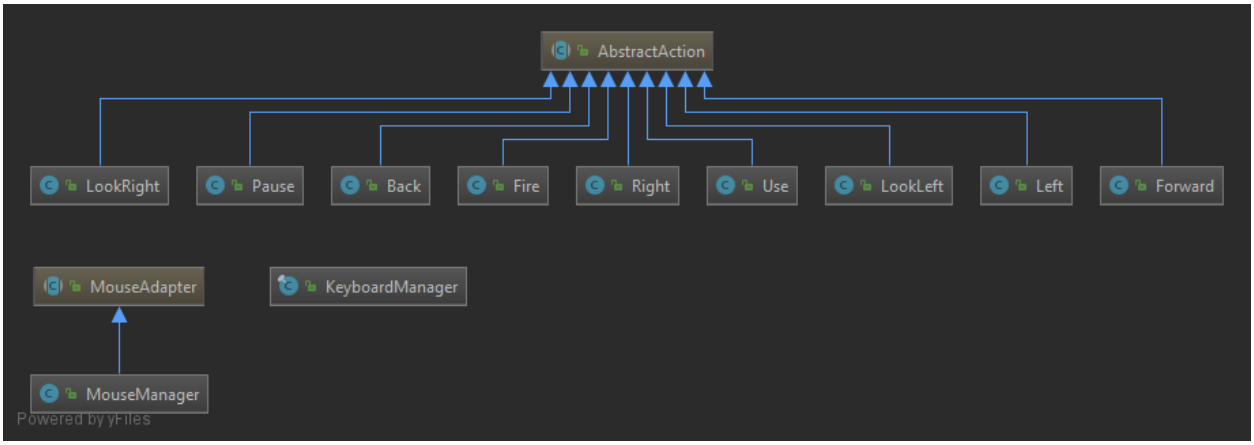
Pickup:



Projectile:

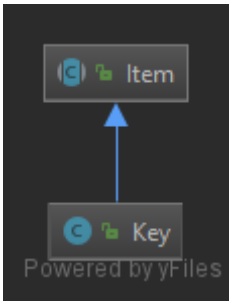


Input Manager:

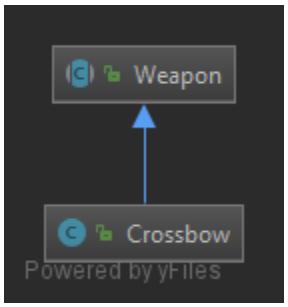


This package is similar to the InputManager package in Tanks 3D. The main difference is that this package also includes inputs from the mouse and the corresponding action classes.

Item:



Weapon:



8 Shared Classes

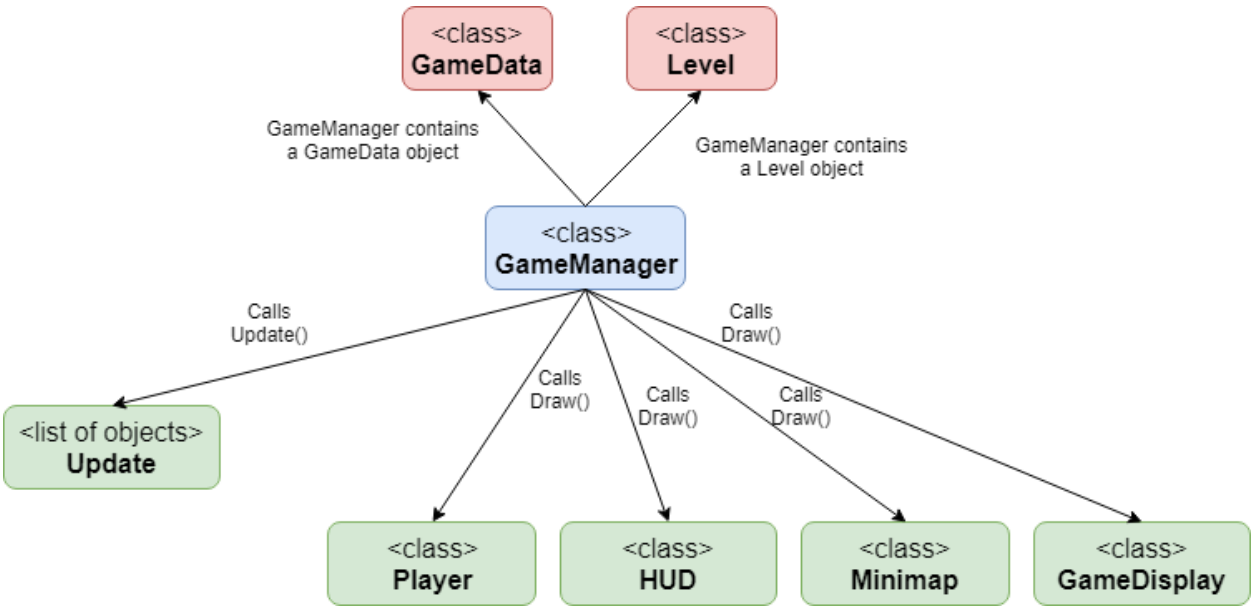
GameManager:

The core of the program is the **GameManager** class. This class serves as the entry point to the program and manages the entire game.

Each frame, the **GameManager** class will calculate the time since the last frame: deltaTime. This is passed to the update method of any objects that requires an update. After all of the objects are updated, the **GameManager** draws the frame. First, it passes a buffered image to the draw method of **Player** to draw the world. Then, the **HUD** object's draw method is called to draw the player's current stats. Finally, the draw method of **Minimap** is called to display the minimap. To push the frame to the screen, the **GameManager** class passes the buffered image to the **DisplayWindow** object. Then, the loop begins again.

To manage all of the data, the **GameManager** class contains a **GameData** object. This contains the player, camera, level, hud, minimap objects. It also has a list of wall, entity, usable, and updatable objects. This object passed to the **Entity**'s update and all of the draw methods to provide the objects the needed information to perform their tasks while maintaining encapsulation.

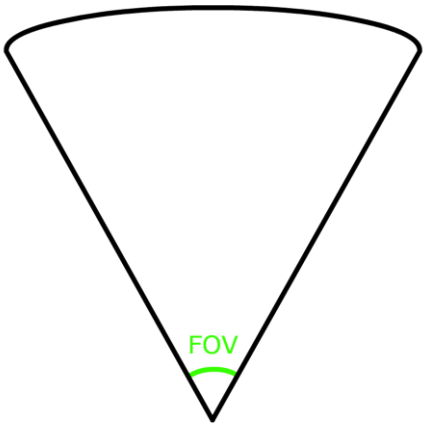
The **GameManager** is also responsible for instantiating all of the objects, changing the level, and handling when the game is over.



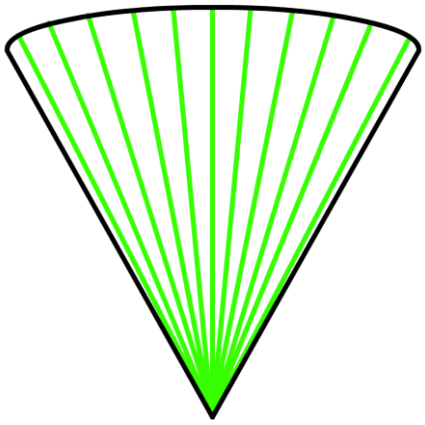
Camera:

The class that handles the projection of a 2D world to a 3D image is the **Camera** class. The draw method of this class draw the world to a buffered image at its position and angle.

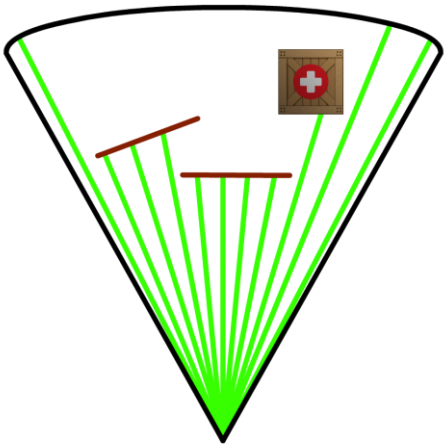
The camera class uses “ray-casting” to render the world. Each camera object has a pre-determined field of view, or FOV.



This FOV acts as a bound for the rays that the camera casts. There is a ray cast for every column of pixels of the game display. And the rays are cast starting from the left bound of the FOV to the right bound.



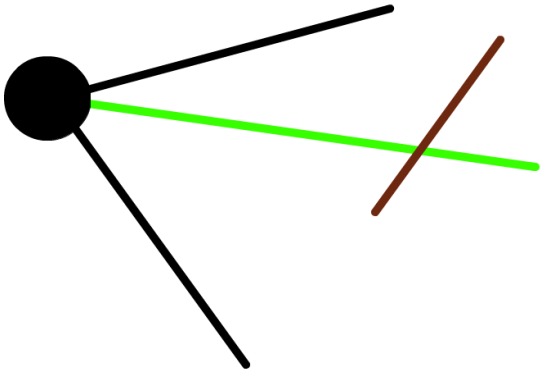
These rays collide with wall objects and entities. The rays can determine how far away the object is and where on the object the ray hit.



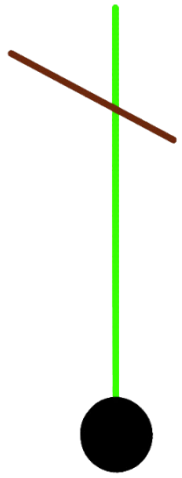
Using this information, the camera can determine what part of what texture to display at that column and how high the column should be.

The rays could be implemented using traditional line intersection equations and the use of the distance formula. But these methods are expensive. Because rays are colliding thousands of times per frame, they need to be efficient to minimize the performance impact.

The algorithm for the rays used in my games relies on the fact that the player's angle and the angle of the ray are known. This image shows the camera, the camera's FOV, one of the rays, and a wall.

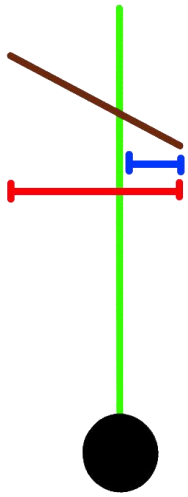


The angle of the player and the ray are known, so it is trivial to rotate the wall to in front of the player.



Then, it is easy to determine if the ray hits. If one point of the wall is to the left of the y-axis and the other is to the right of the y-axis, the ray hits.

This method also makes it easy to calculate where on the wall the ray hits. The ratio of how far along the wall the ray hit is the x coordinate of the point to the right of the y-axis divided by the difference in the x-coordinates. To find the column of the texture that will be applied to the column of pixels the ray represents, the width of the texture can be multiplied by the calculated ratio.

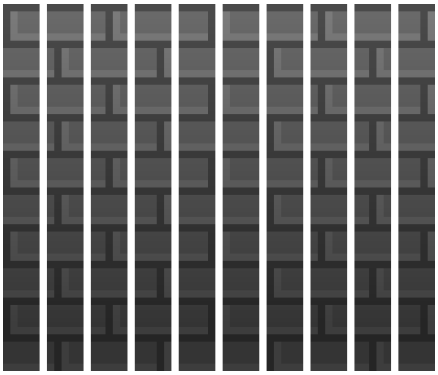


Finally, the distance can be found by fitting a line to the two points of the wall and finding the y-intercept. The distance is multiplied by the cos of the ray angle to remove the fish-eye effect caused by this method of ray casting.

This same method is used to detect entities. The only difference is that entities are represented by a wall with a width of the diameter of the entity's hit-circle. And this wall is positioned so that it faces the camera.

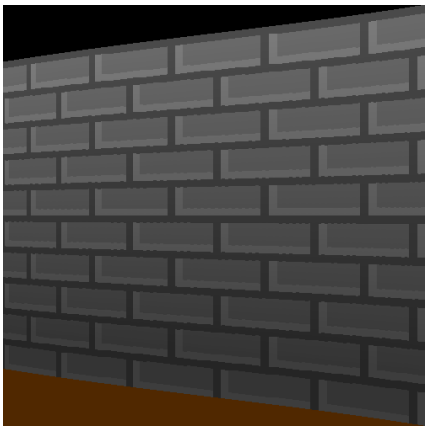
Once the needed location of the texture is found, it can be drawn to the screen. Here is one of the textures used:

The rays will scan through the word and detect collisions with walls and entities. Normally, multiple rays will hit a wall and select several columns of that texture.



Then each slice of the texture is scaled depending on its distance to the camera. The formula is simply the height of the wall in-game divided by the distance.

This is an example of a transformed wall in the game:

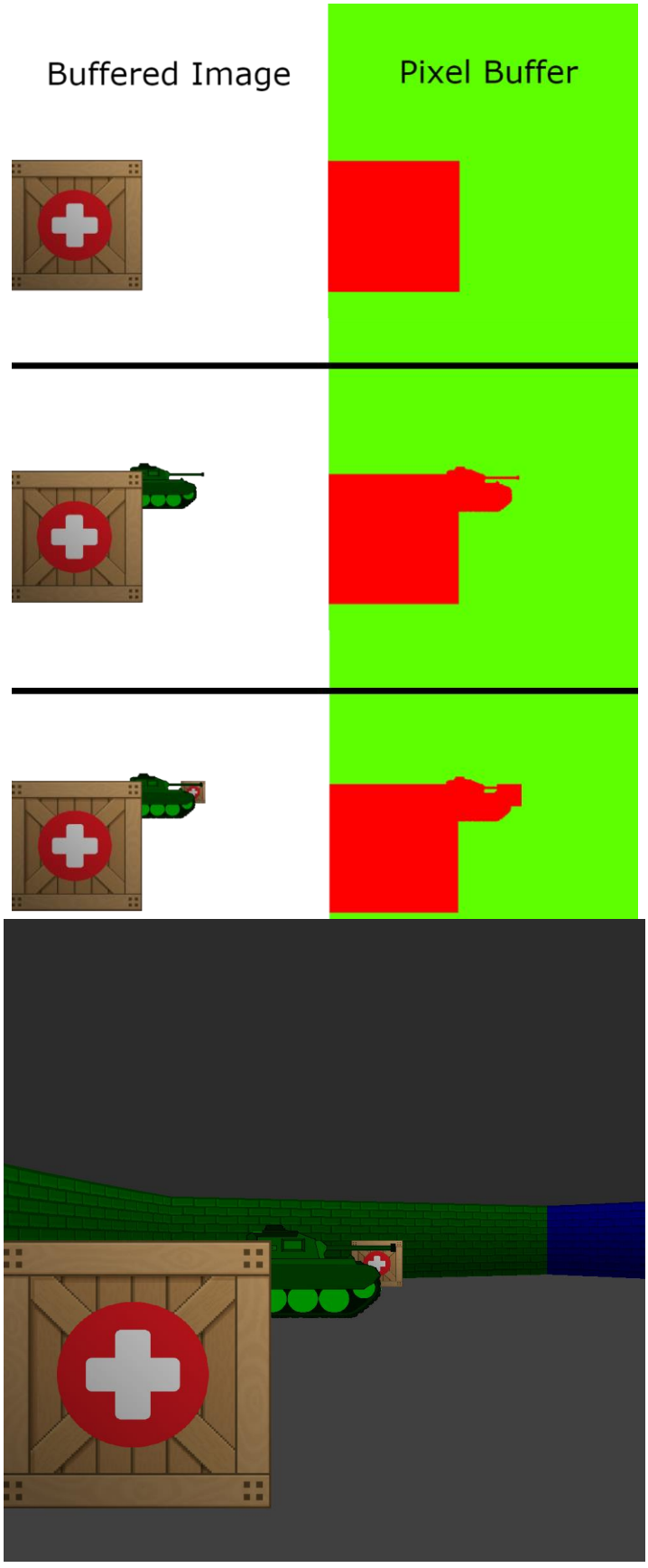


The camera also has culling to prevent unnecessary drawing of pixels.
A rudimentary algorithm for managing overlapping walls is the “painter’s algorithm,” in which the furthest walls are drawn first. Then the closer walls are drawn on top until all of the walls are drawn. The problem with this method is that drawing pixels are resource intensive. So, drawing pixels that are soon overwritten is a waste of performance.

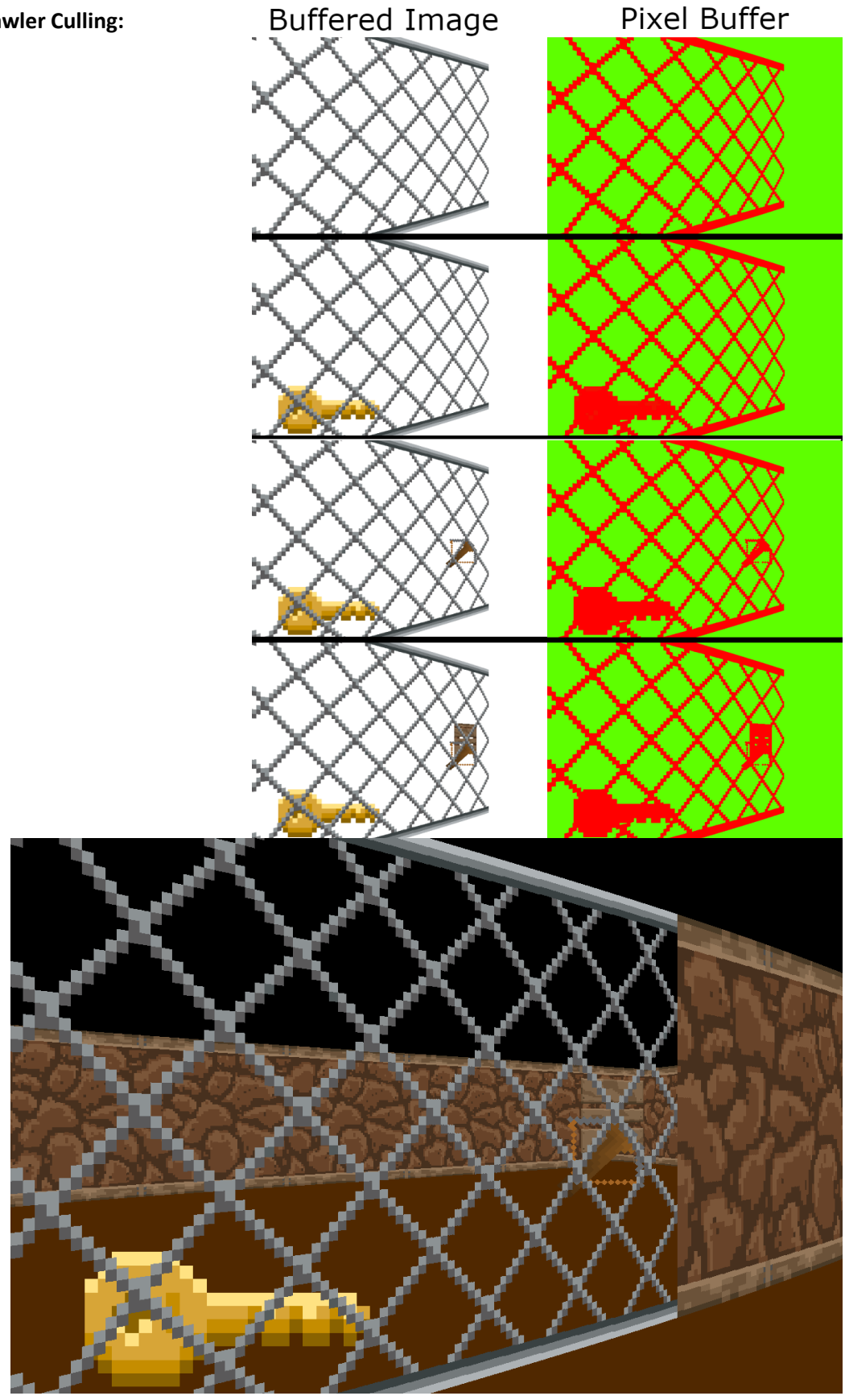
To avoid this issue, the camera class has a pixel buffer. This is a Boolean buffer that determines if each pixel can be written to or not. The objects are drawn in order from furthest to closest. Every time a pixel is drawn, it checks if the corresponding boolean in the pixel buffer is true. If it is, the pixel is drawn and the boolean is toggles.

In Tanks3D, the culling was only applied to the entities, since all of the walls were opaque. But in Dungeon Crawler, some of the walls are see-through. So the culling procedure applies to them too.

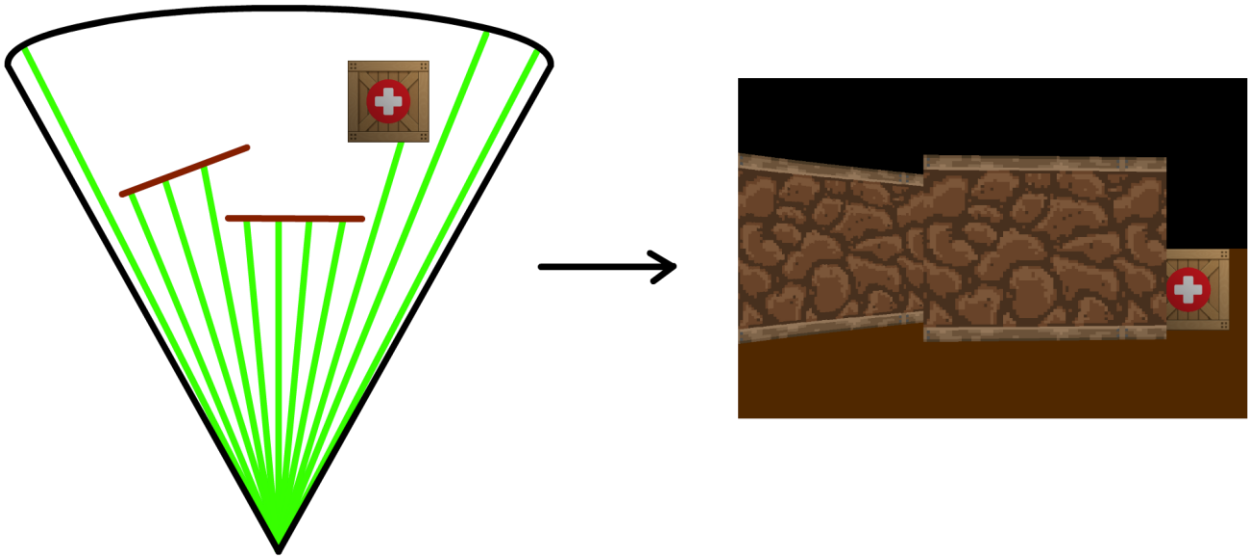
Tank 3D Culling:



Dungeon Crawler Culling:



The end result is an impressive 3D projection of a 2D world



Here is an animation demonstrating how the world is rendered column by column:

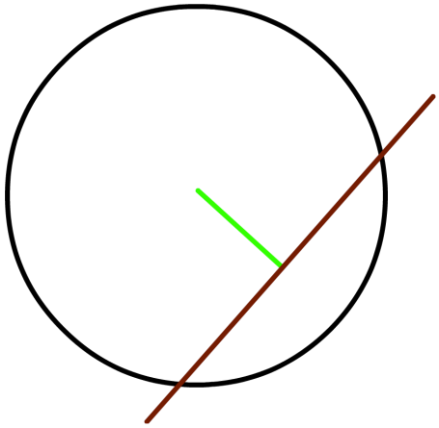
https://i.imgur.com/MTjJSO1.gif

Entity:

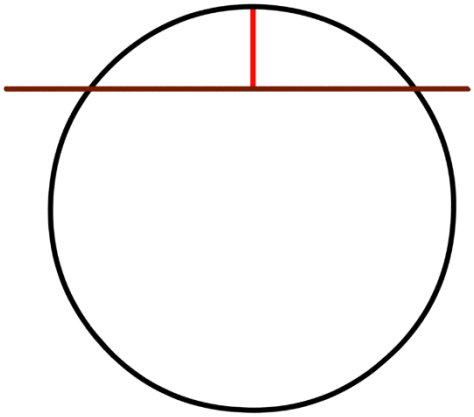
The entity class manages the objects in the game represented by a single point and sprites. Each entity has a direction its facing, an array of sprites for the different angles the entity can be viewed from, and a speed. The entity class implements the “update” interface, so every entity is updated each game tick.

The entity class is also responsible for the collision detection. Walls don’t move, so every entity checks if they collide with a wall or another entity each tick. Again, traditional methods of collision detection using the distance formula would be costly. So the Entity class relies on the atan2 function and rotations.

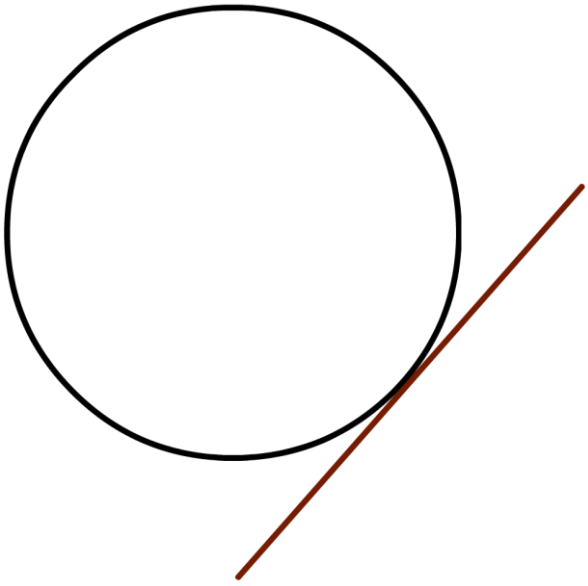
For collisions with walls, the entity first calculates the angle at which it will be moved back. This is the angle perpendicular to the angle of the wall.



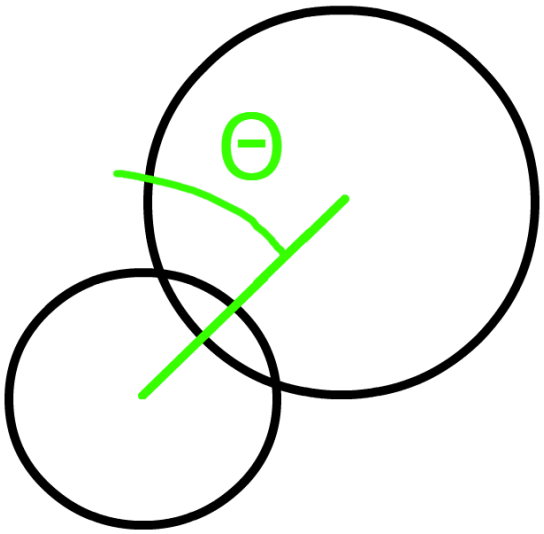
Then using this angle, the wall is rotated to in front of the entity. Then it is easy to calculate the distance that the entity needs to move: the hit-circle radius of the entity minus the y position of the wall.



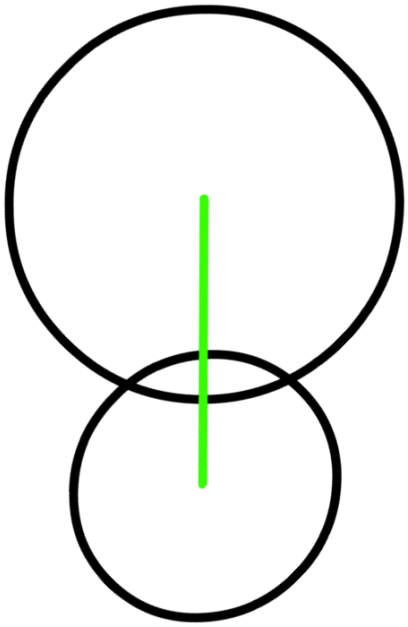
Finally, the entity can calculate the distance it needs to move using the calculated angle and distance.



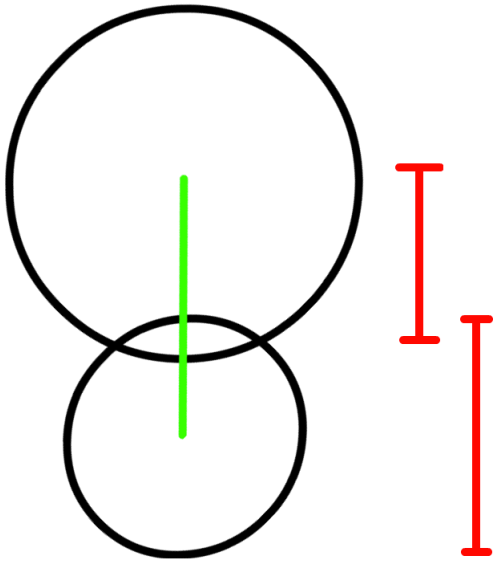
A similar process is used for collisions between two entities. First, the angle between the two entities is calculated:



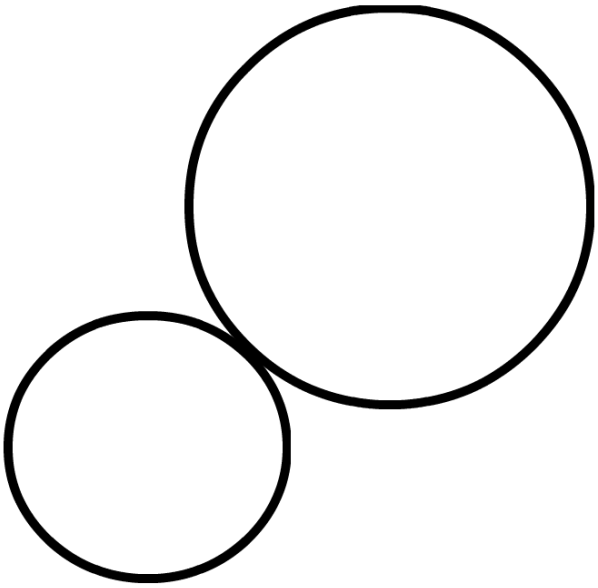
Then the second entity is rotated around the first so that they are aligned along the y-axis.



The distance is now easy to calculate using the hit-circle radius and centers of both entities.



Finally, the first entity is moved based on the angle and the distance.



Fast Math:

This was an important class in optimizing my game. The FastMath class contains a lookup table for sin and cos. This allowed for much faster rotation computations. In addition, it contains other methods to perform calculations faster than the regular Math class can.

Image:

Again, this class contributed to the performance of the games significantly. The provided classes for editing buffered images are quire slow. So this class provides a better and faster alternative.

These are the three major classes shared in both games. There are many other shared classes, but those were discussed in the Class Diagrams section.

9 Tanks 3D Classes

There are no classes specific to Tanks 3D that weren't used in some way in Dungeon Crawler. However, many of the classes were changed.

10 Dungeon Crawler Classes

There were several classes added to Dungeon Crawler that weren't in Tanks 3D:

- **SoundManager**

  Loads and stores all of the audio files that will be used in the game. Any class can give **SoundManager** an audio file name and **SoundManager** will play that file.

- **Item**

  Items are objects that players can pick up and carry with them. The only **item** in Dungeon Crawler is a key.

- **Weapon**

  Weapons are objects that players can pick up and use to fire projectiles or do mele damage. The only **weapon** in Dungeon Crawler is a crossbow.

- **ObjectManager**

  This is an important class that remedies some of the issues in Tanks 3D. In Tanks 3D, entities would modify sensitive array lists directly and cause concurrent modification errors. To solve this, **ObjectManager** takes requests to add or remove entities from the array lists. It then waits until a safe time to perform those actions.

- **MutableDouble**

  To reduce the number of parameters passes needed, **MutableDouble** acts as a wrapper class for a double. This allows the double's address to be passed and remove the need for it to be passed as a parameter.

- **Update and Usable Interface**

  These interfaces help organize classes to be stored in one array list. Each only has one method that the inheriting class needs to implement.

11 Reflection

This was a very ambitious journey that I embarked on. It took much longer than I had expected to complete these projects. I also encountered many problems that I wasn't expecting. However, I struggled through those sticky situations and learned a lot. I was able to meet all of my expectations for Tanks 3D. And I am satisfied with the progress I made on Dungeon Crawler as well. I wasn't able to complete all of features I had in mind for it, namely particle effects and lighting effects. However, it exceeded my expectations in aesthetic and game play.

Thanks to the dynamic approach I took in developing Tanks 3D, many of the aspects of Dungeon Crawler were easy to make. The levels only took a few hours to make and the assets were easy to add to the game. However, I spent a lot of time on unexpected challenges. Changing the rendering algorithm support see-through walls was particularly hard. Especially since it came with a performance impact. I also increased the resolution for Dungeon Crawler, making performance an even large issue. A lot of time was lost trying to optimize the game and bring it up to a playable frame rate. However, I think that effort was worth it since the game runs very smoothly now. I am glad that I prioritized the performance over the particle and lighting effects, since the effects would have ruined the framerate and playability of the game.

One of the regrets I have is spending so much time on the assets. I spent many hours finding art, editing images, finding sound effects, and editing the audio. These gave the game a consistent tone and made both the graphics and sounds satisfying. However, these improvements were not necessary. After optimizing the game, I should have focused on the particle and lighting effects.

12 Conclusion

This was an extremely fun project for me. I definitely plan to keep working on it into the future. Over this winter, I plan to implement the particle and lighting effects. As a long-term goal, I would like to make this game an online cooperative game and add a story.

These two games taught me a lot about program design. Unfortunately, I did not work with anyone on this project and missed out on those teamwork skills. However, I am glad that I was able to work on this project individually because I was able to set a much higher standard for myself. Through the development of this project, I better understood the importance of flexibility of code. Using this approach, I was able to use the first game as a starting point for my second game. I also learned the importance of judging the difficulty of a task and prioritizing tasks accordingly. This lesson was learned the hard way with features I was unable to complete before the second game was due. However, I now know to be more careful of this in the future. I will be sure to give myself more time than I think I need and to not promise too much. However, I will continue to set the bar high for myself and continue to improve my programs past what I have promised.