Software Design

Game Design Documentation

Cataclysmic Cosmos

Robert Lockyer and Andrew Carter
4/3/2009

Table of Contents

Introduction

Cataclysmic Cosmos is an original game inspired by games of the tower defence genre as well as classic arcade games like asteroids. The game revolves around a single spaceship that is controlled by the player. It is the player's task to defend a planet from incoming waves of enemy ships. The player cannot attack directly, his ship is unarmed. However, he can build and relocate various types of turrets around the planet. These turrets will automatically orbit around the planet. This orbiting system is one of the main differences between Cataclysmic Cosmos and other tower defence games. *Figure 1* shows same initial concept art created in order to help illustrate game play mechanics.

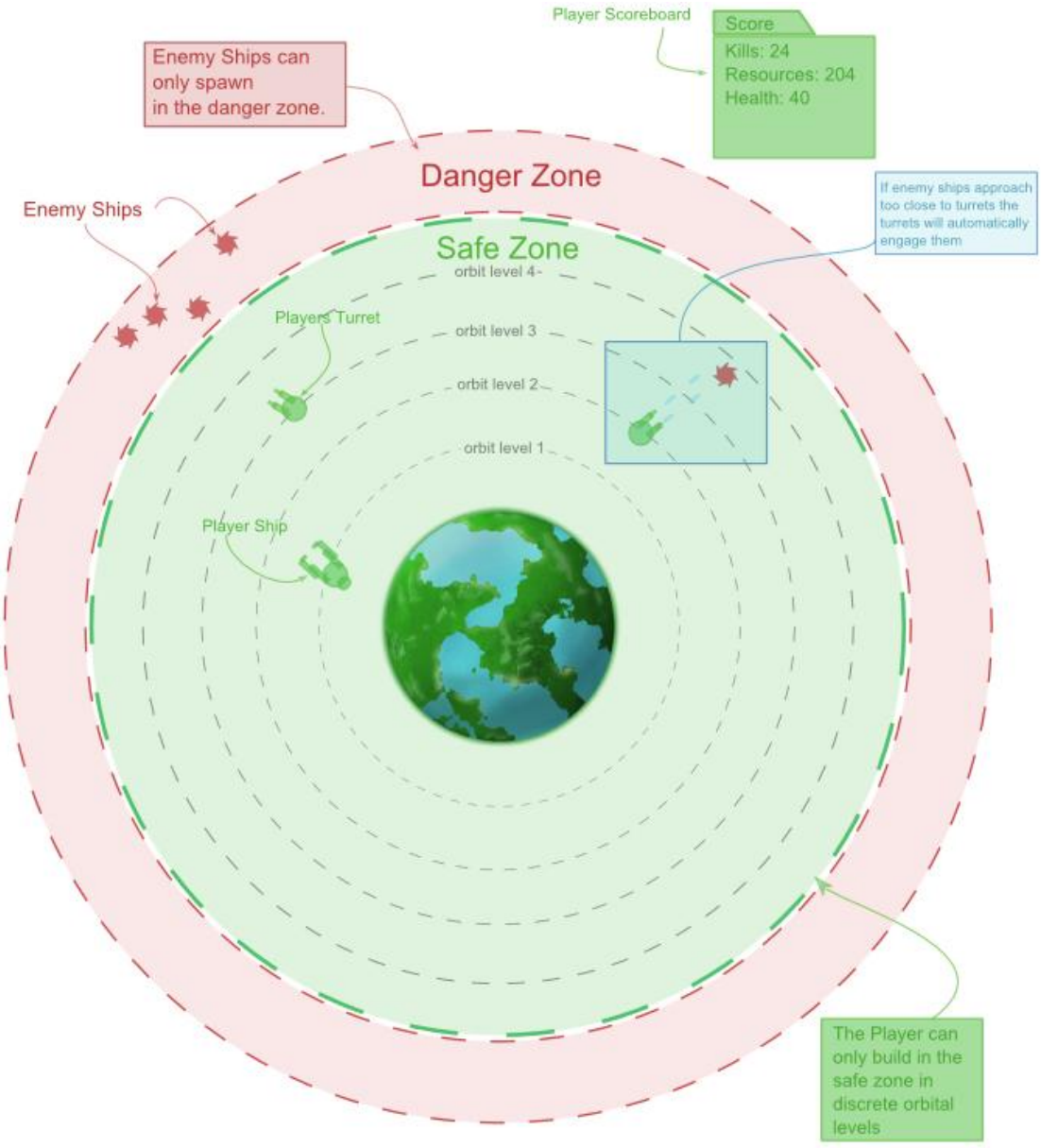

Figure 1: Initial Concept Art

Use Cases

Start-up

Use Case	"Game Start-up"
Subject	Cataclysmic Cosmos
Goal	To Start the Game
Actors	The Player, The Operating System
Precondition	The Player is interacting with an OS which has JRE 1.5.0 or greater installed
Steps	1. The player requests that the OS start the Game. 2. The OS starts the JVM, which launches the Game. 3. Game shows the player an intro screen 4. The player presses the spacebar 5. The Game draws its playing field and the game starts running
Post Condition	"The Game is Started"
Alternative Paths	1. After Step #3: If the player presses the 'Esc' Key or clicks the close button inherent to the application window the game terminates with post condition "Game Terminated" 2. After Step #3: If the player presses H in the intro screen they are taken to a help screen, from there the player can return to step 3 (the intro screen) by pressing BACKSPACE

Player Ship Movement

Use Case	"Player Ship Movement"
Subject	Cataclysmic Cosmos
Goal	To Manoeuvre the Player Ship in the Game World
Actors	The Player
Precondition	"The Game is Started"
Steps	1. The player presses a keyboard key(s) associated with a movement direction – 'W' for Forwards, 'S' for Backwards. 'D' for Yaw clockwise, 'A' for Yaw counter-clockwise. 2. The player begins, and continues to move in the direction(s) specified until the key(s) is released 3. The player ship continues moving along the current velocity vector
Post Condition	"The Game is Started"
Alternative Paths	1. The ship reaches the edge of the screen. It is instantly transported to the opposite edge.

Placing a Turret

Use Case	"Place Turret"
Subject	Cataclysmic Cosmos
Goal	To place a friendly turret in the game world
Actors	The Player
Precondition	"The Game is Started"
Steps	1. The Player moves the player ship into an orbital zone 2. The Player presses a key corresponding to the type of turret he wishes to create – i.e. 'T' for "plasma" turret type 3. The game checks to ensure that the player has enough money to create the desired turret 4. The turret is created and placed on the orbital path corresponding to the orbital zone the player ship is currently in
Post Condition	"The Game is Started" and "At least one Turret is in play"
Alternative Paths	1. During Step #2: If the player tries to place a turret outside of an orbital zone the attempt fails and the use case terminates with post condition "The Game is Started" 2. During Step #3: If the player does not have enough money to create the desired turret then the game informs the player of this fact and the use case terminates with post condition "The Game is Started"

Move Turret

Use Case	"Move Turret"
Subject	Cataclysmic Cosmos
Goal	For the Player to Move a Friendly Turret
Actors	The Player
Precondition	"The Game is Started" and "At least one Turret is in play"
Steps	1. The Player moves the player ship into the action range (defined radius) of a turret 2. The game provides visual confirmation that the player ship is within action range of a turret 3. The user presses the SPACEBAR to attach the turret to the player ship 4. The Turret stops its normal movement pattern and moves along with the player ship. It also stops its attack pattern 5. The player manoeuvres the ship to a location where he would like to place the turret (this location must be within an orbital zone) 6. The player presses the SPACEBAR to detach the turret 7. The turret begins moving along the orbital path associated with the zone the player ship is currently in
Post Condition	"The game is started" and "At least one Turret is in play"
Alternative Paths	1. During Step #6: If the player ship is not currently located in an orbital zone then the key press does nothing. The use case continues at Step #6.

Enemy "Dive Bomber" Ship attack

Use Case	"Enemy Dive Bomber Ship Attack"
Subject	Cataclysmic Cosmos
Goal	For the enemy ship to attack the planet under defence
Actors	The Enemy Ship (AI)
Precondition	"The Game is Started"
Steps	1. The game spawns a Dive Bomber ship on the perimeter of the game world 2. The Dive Bomber heads for the planet under defence along a straight line path 3. The Dive Bomber collides with the planet under defence and is destroyed, doing damage to the planet
Post Condition	"The Game is Started"
Alternative Paths	1. During Step #2: If turrets placed by the player do enough damage to the Dive Bomber during its trip to the planet under defence, then the Dive Bomber is destroyed prematurely, doing no damage to the planet

Plasma Turret Attack

Use Case	"Plasma Turret Attack"
Subject	Cataclysmic Cosmos
Goal	For the plasma turret to attack incoming enemy ships
Actors	The Plasma Turret (AI)
Precondition	"The Game is Started"
Steps	1. An enemy ship enters the attack range of the turret 2. The turret fires a straight shot directly at the ship with enough velocity to make a direct hit, damaging the enemy vessel 3. The turret continues to take shots with a regular frequency until the enemy vessel leaves its attack range or is destroyed 4. The turret then chooses it's next target by evaluating which enemy ship is closest to it and in its attack range 5. If it finds a new target it continues from step# 2, else it stops firing
Post Condition	"The Game is Started"
Alternative Paths	1. After Step # 1: If the player ship attaches the turret to itself then it stops firing. The use case ends with post condition "The Game is Started"

Design Outline

Our design is split into two main components, the model and view. The model holds onto all of the games state information. The view is dependent upon the model. This allows us to easily switch out the view if we ever decided to port the game to a new graphics API like Jmonkey. Another key aspect of our high level design is that we make use of the state pattern for controlling what the active scene is. This is useful because screens like the intro screen don't carry the extra baggage that a more complex scene like the playing scene has to deal with. We can develop these components in isolation and add new scenes with relative ease. Our implementation of the state pattern can be seen below (*Figure 2*).

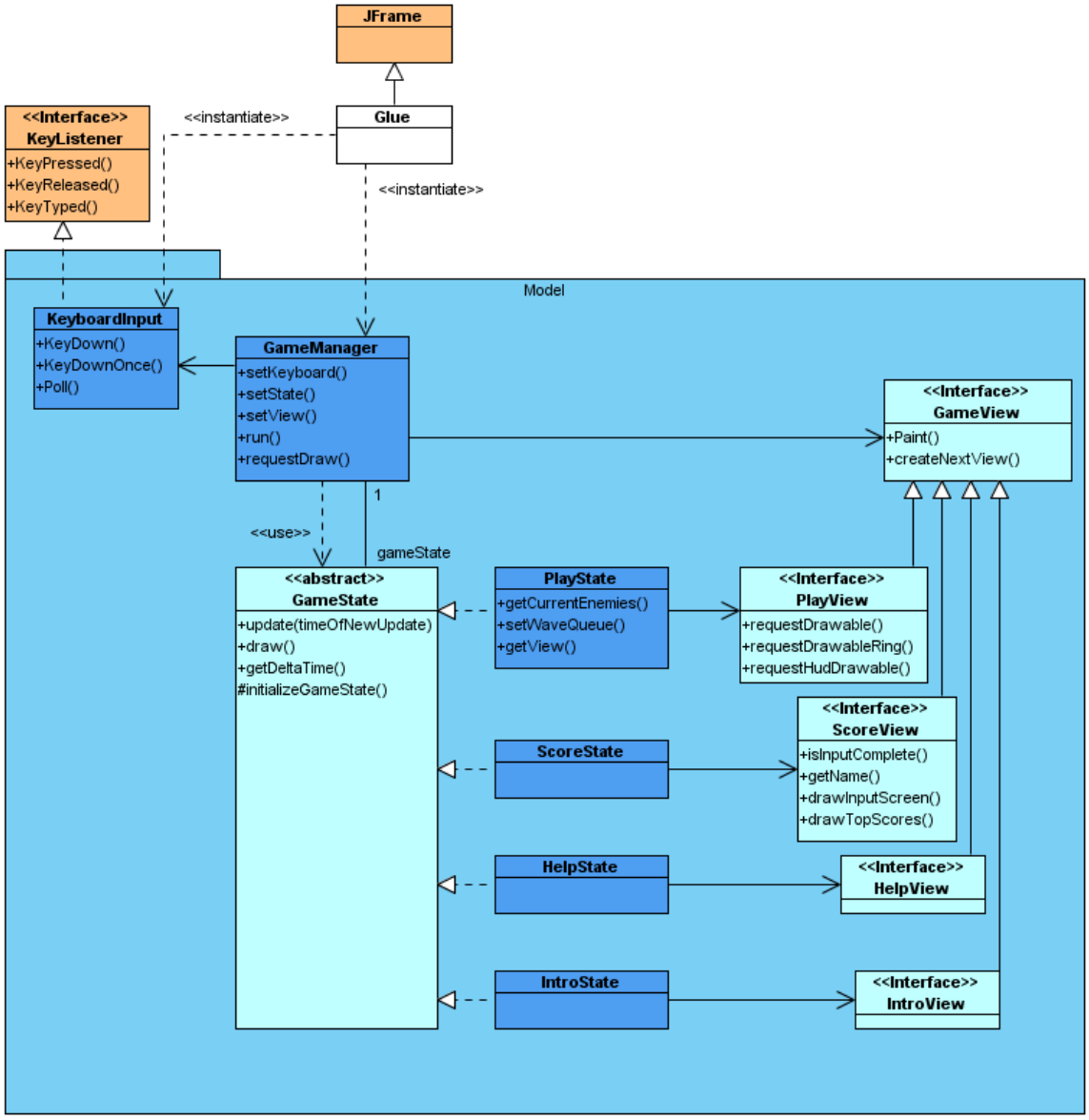

Figure 2: Model Package

The way that the java `Graphics2D` object works required us to design our interactions between the model and view in a very specific manner. The `Graphics2D` object is only valid for the duration of

the `paintComponent` method this means that the view must call the model from this method in some way in order to fetch fresh information. Since the view knows the model it can do this with relative ease. The view is known by the model through a series of interfaces shown below. The `GameManager` only knows various views as `GameViews`. But the specific states know them as their more specific derived interfaces such as `HelpView`, `IntroView` etc. These are the interfaces in the middle of Figure 3.

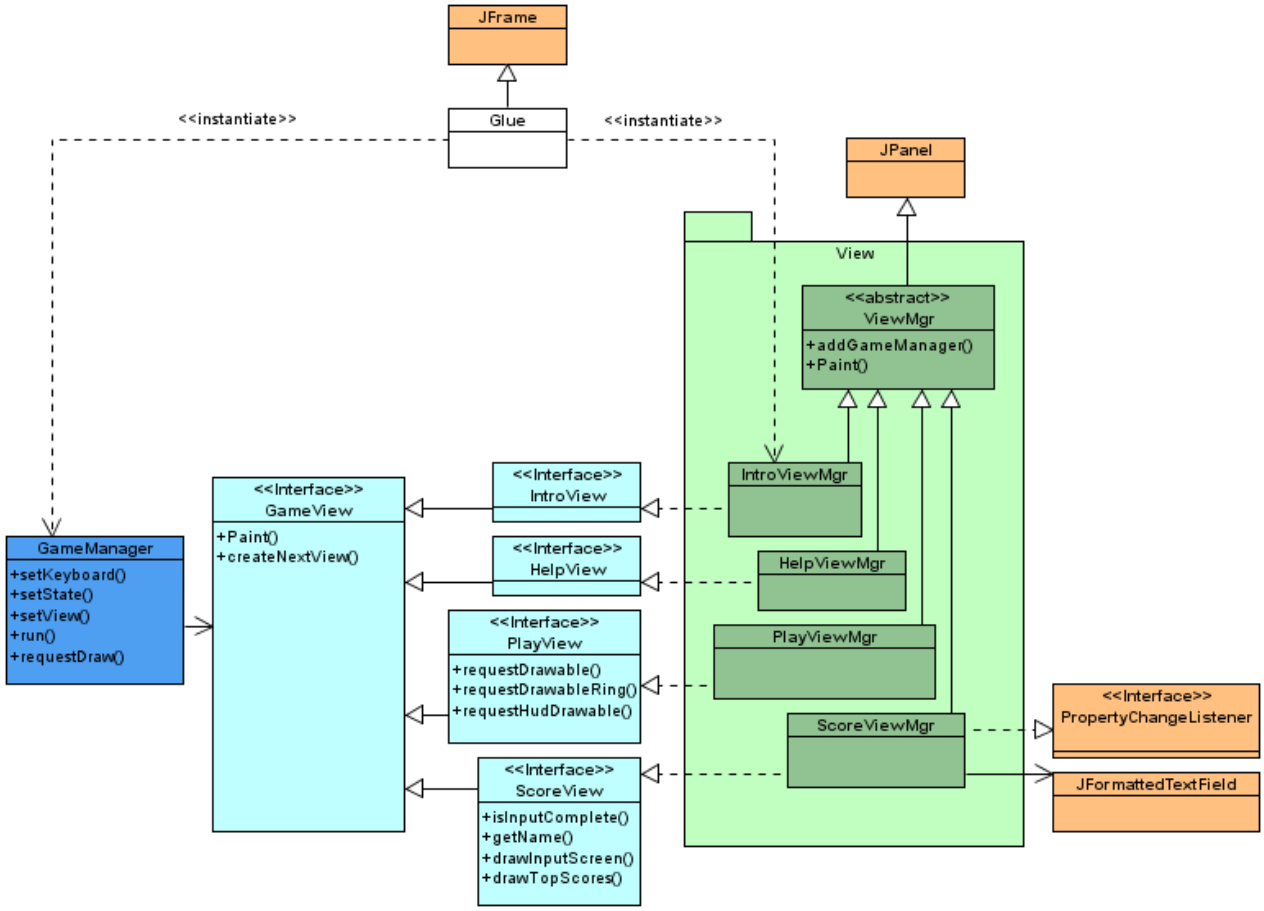

Figure 3: View Package

Package Level Design

As stated in the design outline the project is split into two main packages. These are the model and view respectfully. The view is considerably simpler than the model and therefore is not further divided into smaller packages. However the model is much larger and complex and therefore requires the greater organisation and compartmentalization that packages provide. The model is highly dependent upon `GameStates` and therefore we have decided to group these into a package under the names of `States`.

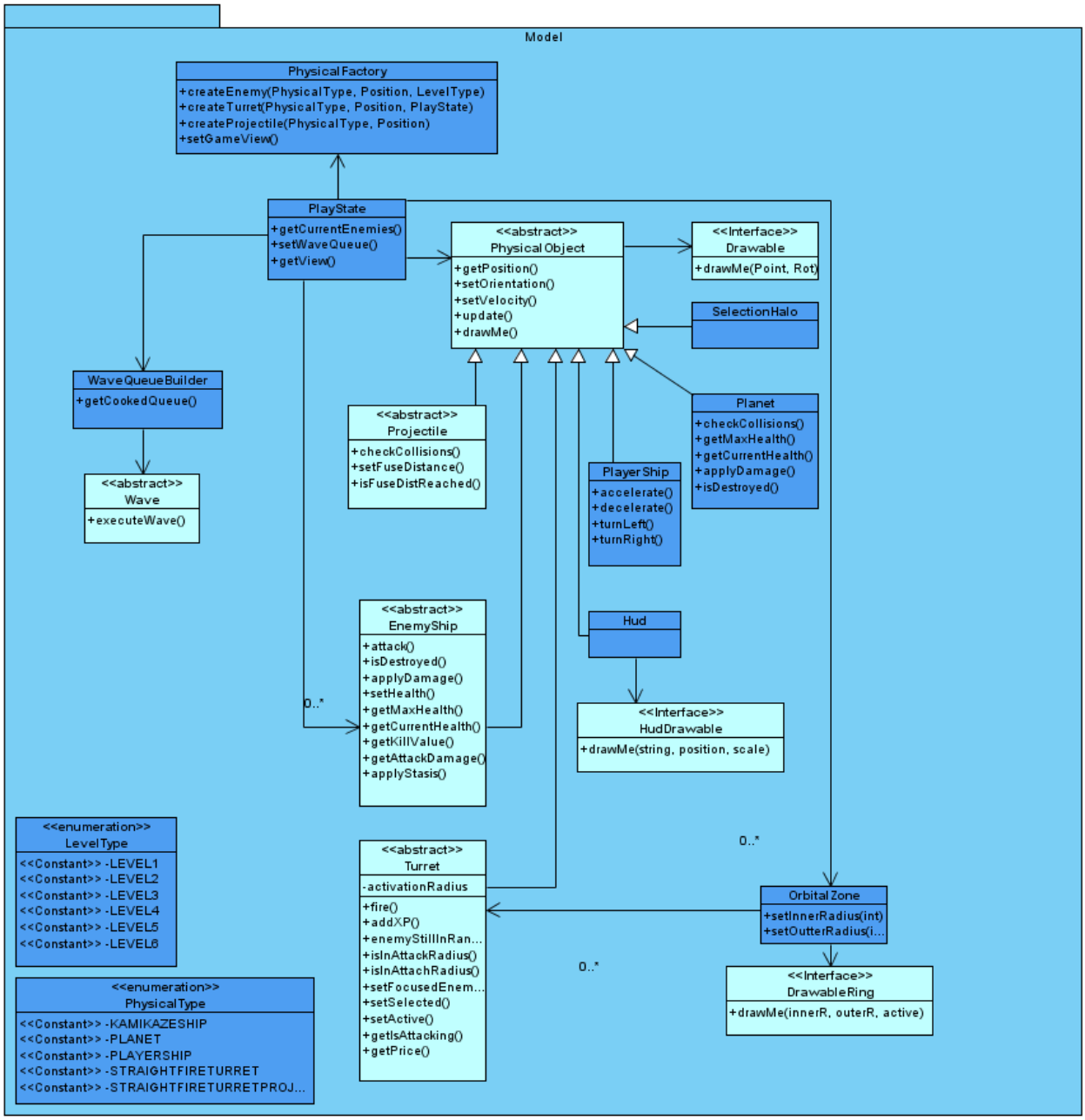

Figure 4:PlayStates Sub-Components

The most complicated of these states is `PlayState`. The layout of classes used by this state actually dictates the organization of the remaining packages that have not been mentioned yet. Therefore to understand the layout of these packages it is important that one has some idea of how the `PlayState` is set up. *Figure 4* shows the relationships of classes and interfaces related to the operation of `PlayState`.

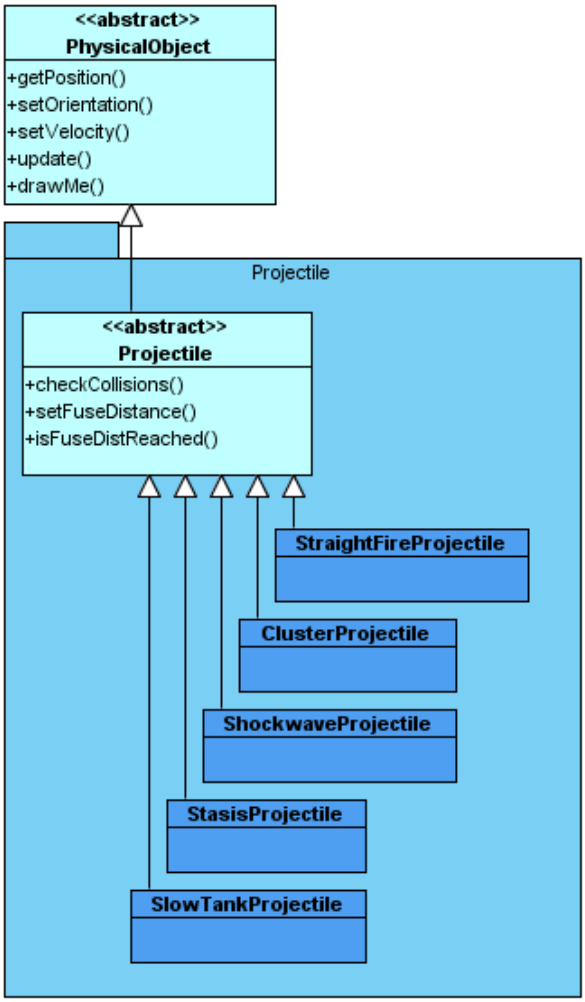

Figure 5:Projectiles Package

There are three important hierarchies of classes which we have logically grouped into packages. These are projectiles, Turrets and Enemies. Each of these is derived from abstract classes of the same name and therefore they share much in common. *Figure 5*, *Figure 6* and *Figure 7* show how these packages are arranged.

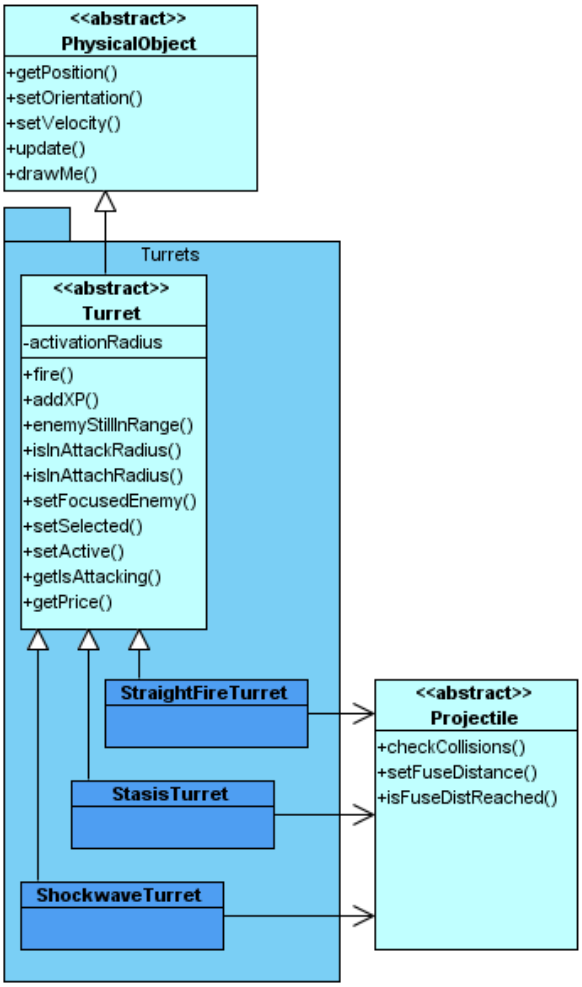

Figure 6:Turrets Package

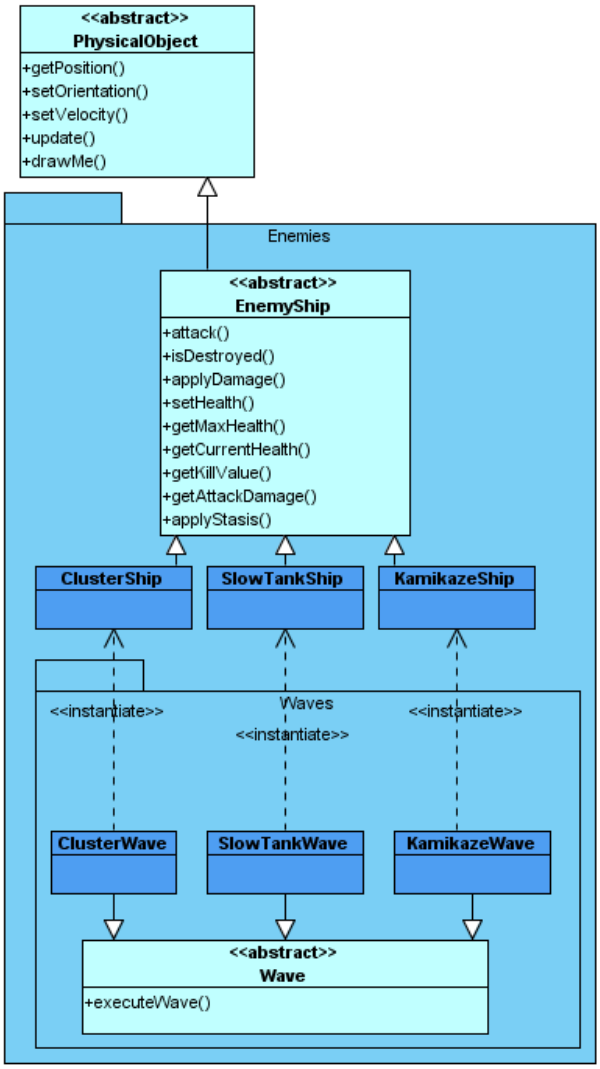

Figure 7: Enemies Package

Class Level Design

Game Manager

Top Level Game Model Object. Controls Update call Percolation among other things. The `GameManager` class is concerned with keeping track of time and holding onto the current state of the game. To keep track of this information we've implemented the state pattern. The three states that are designed so far are the `IntroState`, `PlayState`, and `ScoreBoardstate`. These states require pointers back to the `GameManager` so they can change the state when required.

ViewMgr

An abstract class, which all view managers should extend. Used to collect the common functionality between view managers, which seems reasonable to generalize. Also useful as a means from which to implement polymorphic code.

Vector2

A two dimensional, geometric Vector Allows for useful operations such as finding the distance between two points, getting an angle from a vector, performing dot products and more. Used extensively by `PlayState` components.

GameState

Acts as a state in the state pattern, The `GameManager's` state is controlled by what concrete `GameState` it holds at any given time. Swapping the old `PlayState` with a new one effectively resets. Other `GameStates` are included for an intro, scoreboard and help screen.

PlayViewMgr

Main View Component used to Visualize the Play State This `JComponenet` extends `JPanel` and implements `PlayView`. It is responsible for managing all the various view subcomponents for the Play State. In addition, by implementing `GameView` it is able to act as an ambassador to the model on behalf of the rest of the view.

PlayState

State Class which Orchestrates the cooperation of key gameplay objects Of the three main game states, the play state is by far the most significant. This is because it handles the core game logic. It creates different kinds of physical objects such as the `playerShip`, `OrbitalZones` and Planet and then specifies in its main loop when they should move along with some facets of how they should interact.

For more information on classes please refer to the included javadoc.

Key Interfaces

Making the view depend upon the model required the creation of several interfaces that the view classes could inherit. Each game state was given a corresponding interface which was derived from the abstract class `GameView` seen below. This allowed the `GameManager` to know the view states in a general way while the specific `GameStates` could know more specific details about their functionality.

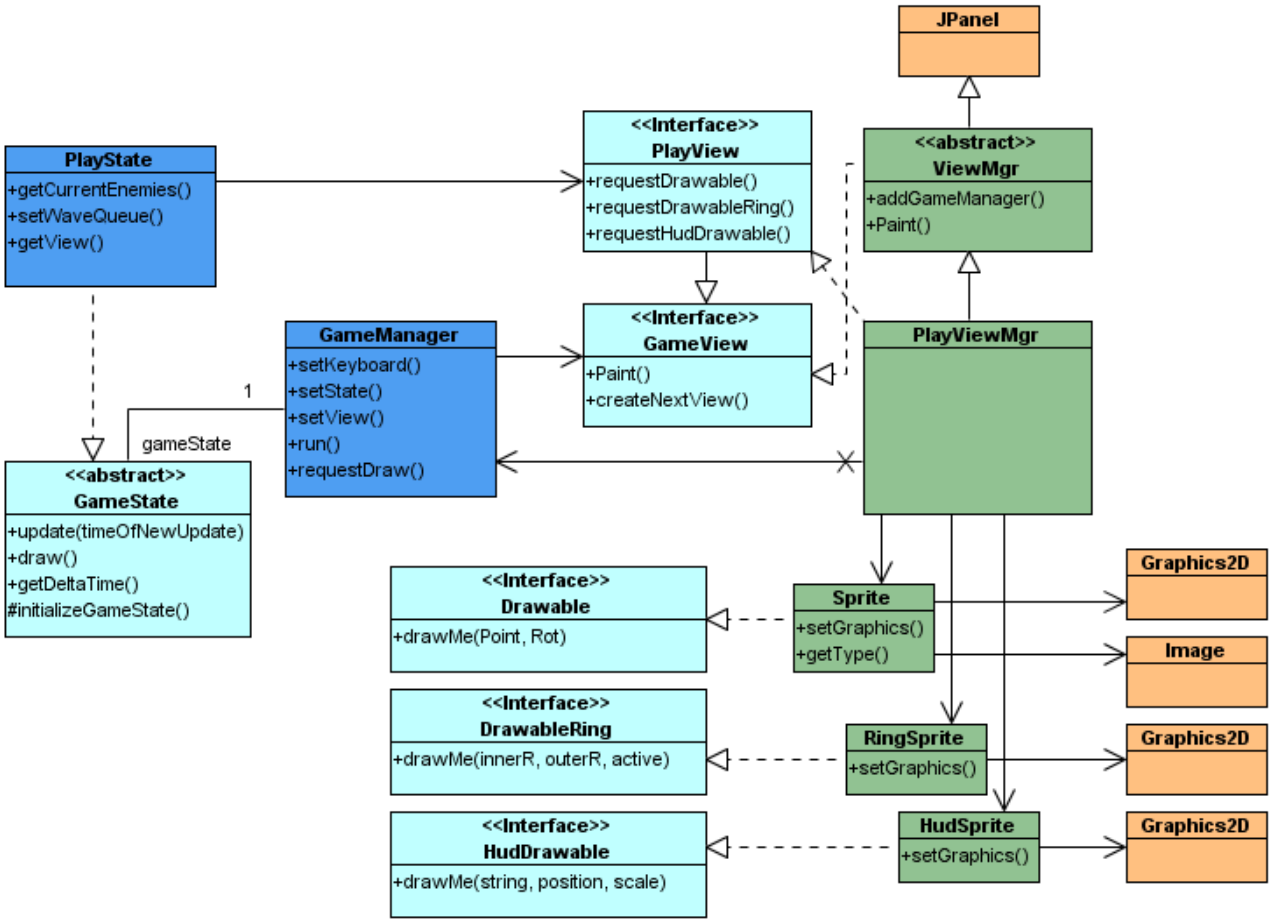

Figure 8: PlayState PlayView Interaction

Drawables were a useful interface for the `PlayStates` sub-components. Basically, they were a way of letting the model tell the graphics object how to draw those objects.

For more information on interfaces please refer to the included javadoc.

User Manual

Starting the game is as simple as double clicking the jar file. The player will be brought into an intro screen. From here they can either press the spacebar to start playing or press the H key in order to see a help screen (*Figure 9*). These options are displayed to the player so they won't need to read a manual in order to play.

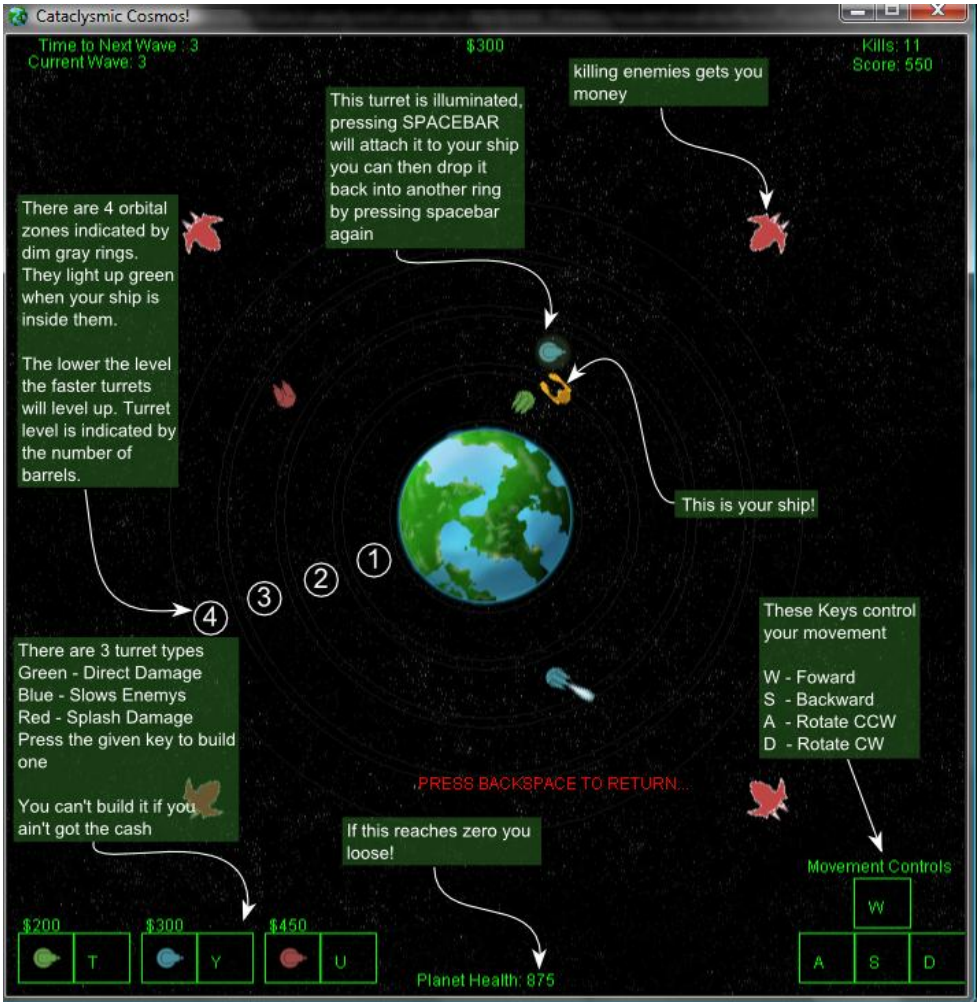

Figure 9: Help Screen

Movement

Controls

W- Forward
S – Reverse
A – Rotate Counter Clockwise
D – Rotate Clockwise

The player is represented by a yellow construction ship; its movement is not restricted by friction. This means the player will maintain their velocity unless they use their engines to change it. If the player

hits the edge of the screen they will be instantly transported to the other side. This can be a useful shortcut in some situations.

Turret Placement

Controls

T – Lay Plasma Turret
Y – Lay Shockwave Turret
U – Lay Stasis Turret

The idea of the game is to build and arrange turrets to defend the planet from incoming enemy ships. To build turrets the player must be within one of the four "orbital zones". These zones are indicated by gray rings which surround the planet. If the player is inside a ring it will light up green to indicate this.

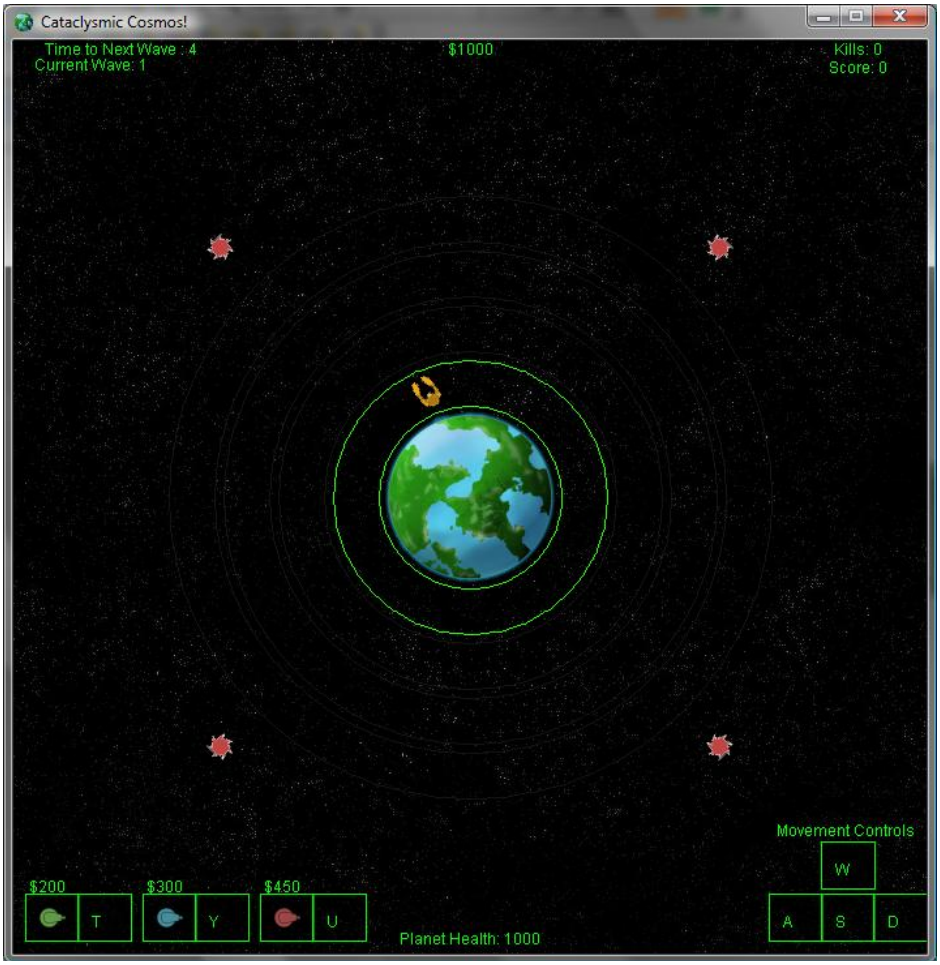

Figure 10:Orbital Rings light up green when the player ship enters them

Once the players inside of a ring he can place a turret by pressing the corresponding keys as shown in the controls guide. However, this will only work if the player has the money required to build the turret. If not those keys will result in no reaction. In the above figure if the player presses either T Y or U a turret will be created on the first level.

Turret Movement

Controls

SPACEBAR – Grab a turret/Drop a turret

Once turrets are placed they can be moved in order to take advantage of the bonuses from different rings. In order to do this the player must move close by a turret. Once they are in range the turret will light up to indicate that it is the active target. Pressing the spacebar at this point will grab the turret and lock it to the player's ship. Dropping the turret works much like placing one. You must be within a ring and pressing spacebar at this point will drop the turret there. No money will be spent to perform this action.

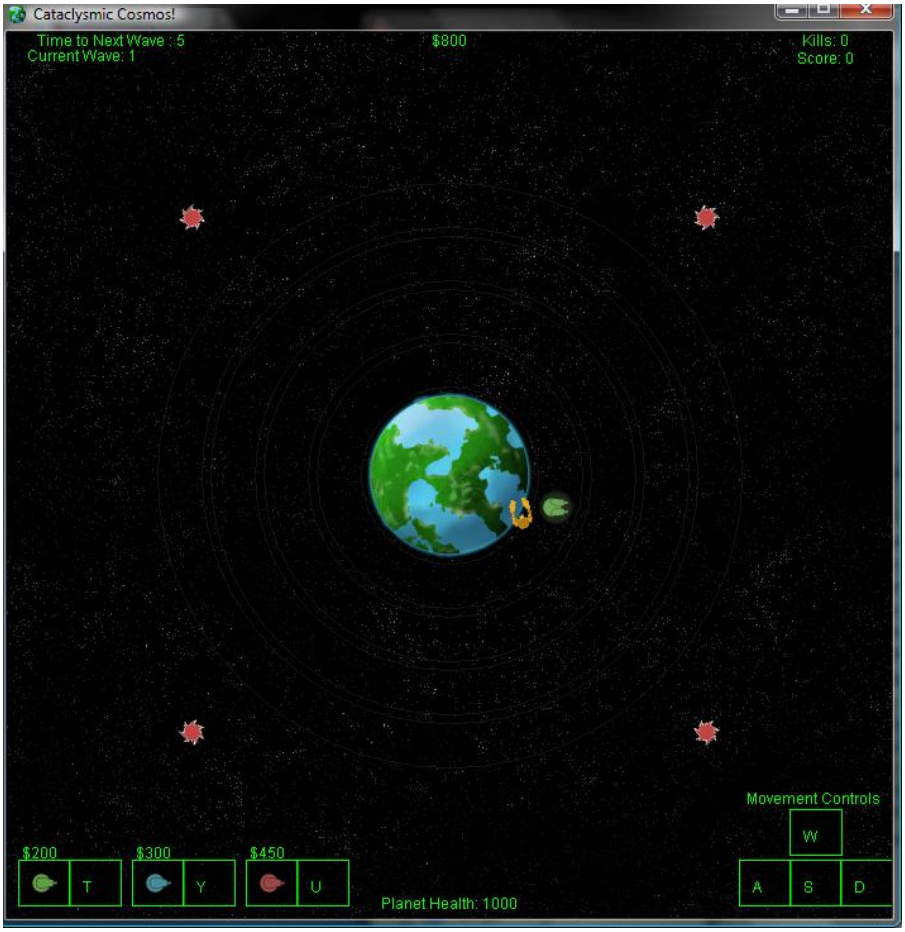

Figure 11: The player ship is in range of a turret

In the above figure (*Figure 11)* the player is within range of a turret. Pressing spacebar will attach it to the player's ship. There is a subtle glow effect on active turrets that shows up better during animation.

Other Controls

Controls

ESC – Exit the game

Game Mechanics

There are a couple of game mechanics that are not immediately obvious and so will be explained here in order to help new players understand the game.

Each turret has three different levels. These levels are indicated by differing graphical representations. These are shown below (Figure 12) for the stasis turret. The higher the level of the turret the more effective it will be.

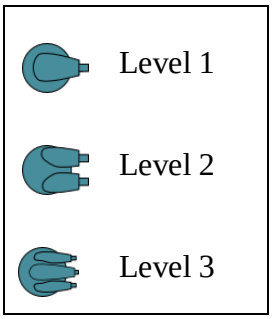

Figure 12: Turret Levels

There are four orbital rings within which the player's turrets can be placed. These rings incur bonuses to the turrets upgrade speed. The closer to the planet the faster the turret will be upgraded. Therefore it is often beneficial to place turrets close to the planet when first building them and then move them out as they reach level 3. The outside ring has the lowest experience bonus; however turrets in this ring get the most shots off since they are not blocked by the planet.

Enemies

There are currently three different enemy types. They are shown below (Figure 13). Their behaviours are generally related to their names. The slow tank is a slow moving unit which can take a lot of damage from turrets before being destroyed. It will approach the planet directly and begin firing on it when it gets within range.

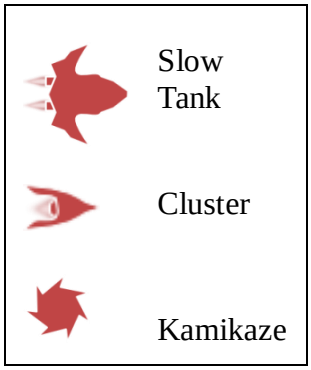

Figure 13: Enemy Types

The Cluster enemy is designed to attack in swarms and is relatively weak. This makes Shockwave turrets ideal against them since they can destroy multiple in a single blast. Kamikaze enemies are somewhere in the middle ground in terms of health and speed. Unlike the other enemies these approach the planet and dive-bomb it dealing damage and destroying themselves in the process.

Conclusion

The model/view split of our system is a useful and flexible design. However, something about how we implemented it results in a disparity between model and view updates. We believe this is a result of the `Jframe` running in a different thread then the model. At first we were receiving a lot of exceptions related to multithreading. Placing synchronization blocks around some of our vectors of `drawables` resulted in eliminating these exceptions. However, it did not fix the root of the problem. There are still occasionally hiccups where one thread waits on the other.

If we were given more time one of our first tasks would be investigating how this multithreading is operating and try and redesign our game to avoid or at least better control this issue.