CSC 413 Project Documentation

Spring 2019

Jonathan Julian

918164239

413.02

https://github.com/csc413-02-spring2019/csc413-tankgame-jjulian91

https://github.com/csc413-02-spring2019/csc413-secondgame-jjulian91

Table of Contents

1 Introduction

1.1 Project Overview

The projects were designed to give us an in-depth look at game development from a programming standpoint, without using a game engine. This challenged our ability to use java Swing, AWT, or JavaFX.

1.2 Technical Overview

These two projects were generally set up the same way. There was a single driver class which set up the game environment, instantiated different game objects, handled the painting of objects on the canvas, and oversaw the collision of objects as necessary.

Tank Wars required quite a bit more set up as there were more game objects to be created, more collisions to handle, and more objects to paint. The added difficulty in the tank game was instantiating and assigning controls to two separate objects. Additionally there was the challenge of adding a "camera" to each of these objects allowing for a split screen. Furthermore we were required to include a mini-map which showed the entire game-world in scaled version. Other objects required in Tank Wars were the instances of bullets, power ups, which were handled differently in Galactic Mail, as well as multiple lives, and a damage indicator.

As previously stated both games are generally the same in their set up. Both games have a main class which include an init() method which is responsible for setting up the game environment, setting up the player object(s), and initializing various hash maps which include shortcuts for the game environment. Once each of the games passes through the init() method it enters the game control loop. This is responsible for making subsequent calls to the paint method which updates the users screen. 99% of the work is done somewhere in this game loop, or a child function of the game loop.

Each of the games share the same file structure which is displayed in figure 1.1. This allowed for portability of the code with only changing thing like class names to make it make sense in the implementation. Every item within the GameObject package is a child of GameObject in some form or fashion, other abstractions were made to allow for grouping of items to ensure that there wouldn't be wasted loops when painting, checking for collisions, and allowed for special case handling of collisions. Each of the "Players" take (int) arguments which correspond to their controls which are passed through a keyEventListener assigned to the JFrame on which the game is drawn. Each of these keys are mapped to the listener and the listener then alerts the corresponding object of the key press.

GameObjects are all surrounded by borders which are drawn as rectangles in java. This allows for a simple way to check for collisions with use of the obj1.intersects(obj2). This call allows for a quick check of the sprites colliding. Using this method can consume more system resources than necessary, which is where it became imperative to create different abstractions for different types of objects. The general premise of collision checking was to check every object colliding with any other object. As the maps grow or the number of bullets increases there tends to be a bit of illogical collision checks. For example: Walls, bases, powerups, and home do not move, this means they are unable to collide with one another, therefore, the separation of moving objects and non-moving objects cut the number of iterations in the loop to less than 50% in most cases.

Once an object was collided each game handled the effect differently. Tank Wars required there to be a health bar and multiple lives for tanks, and I allowed “destructible walls” to be blown up by one bullet. In Galactic Mail one collision with an asteroid ended the game. When an asteroid collides with the mail carrier in Galactic Mail the object instantly dies, which then enters the gameEnd() method which checks for a high score from an internal .txt file. If the user score is a high score the ScoreTable class uses a JOptionPane to get user input and then places the users input and their score in the text file and updates the score board and prints it to screen. If the score is not a high score then the JOptionPane tells the user they are a “loser” and the previous table is provided. Since this project required us to use a jar and I didn’t believe the scope of the project was wide enough to include a database, the Scores.txt file will ONLY update the users name in the IDE. The idea that a JAR is self-contained, portable, and most importantly secure does not allow for the program to update files internally to the archive, and copying resources with the JAR defeats the portability of it I made the choice to not have scores update.

The Tank Game ends similarly where it tells the losing player that they have lost. This is not a score based application therefore there is no high score table needed. Although this is easily implemented by taking the damage produced by destructible objects and adding the damage done to a player score as well as the object taking damage.

The games also differed in the aspect of needing levels. Tank wars did not need multiple levels, but would be easy to implement. Being that the map file for TankWars generated by a comma separated value file, it would easy to write a script, as one of the TA’s did, to generate map files. These map files can be given to JOptionPane to give users the ability to select a map. Since there was a requirement for Galactic mail to contain multiple levels I omitted the CSV file and allowed the game to dynamically generate maps. This is done by taking the user’s last position and applying a “Home” base to it (which doesn’t subtract points from the user as we transition levels). Then it generates 5 bases in random locations checking for collisions along the way and regenerating as necessary. The complexity of the levels is included by taking the level number (n) and applying it to a loop creating n asteroids. To make the game playable I limited the number of asteroids to 5. The other complexity of Galactic mail is that each of the base generates a random speed of the ship. Since the requirement for the game was that it was more difficult to control the ship in space there is also a correlation to the level and the max speed the ship can move.

1.3 Summary of Work Completed

For each game we were required to design the flow of the game, design/create the objects, watch and handle collisions, design the powerups and game environment, and provide structure to the game. The game was built almost from the ground up, with the exception of basing the code off of the Tank Rotation Example and using inspiration from the wingman game provided.

2 Development Environment

Intellij 2019.1

Java 11

3 How to Build/Import your Project

Import instructions: If IntelliJ already has a file open, go to File - > Close Project. This will take you to another IntelliJ page and from there you can select checkout from Version control. With access to Git Repo go to the following link: https://github.com/csc413-02-spring2019/csc413-02-jjulian91 . You will be able to clone the repo from here. Once the repo is cloned you will be able to open this as a project with name of your choice following the prompts selecting MAVEN structure. . From here select the drop down in between the hammer icon and the play icon in the top right corner of the IDE. Select the plus sign and choose application from the list on the left pane then ensure the following settings:

1. Main Class: GameFiles.GalacticMail
2. Program Args: None
3. Working Directory: N/A
4. Use class Path : N/A
5. JRE: JDK 11

4 How to Run your Project

From Jar: Navigate to JAR folder. Double click.

From IDE: With the settings as described above you will be able to just click the play button.

5 How to Play:

TankWars:

Basic controls for left tank:

W – moves forward

A – rotates left

S – moves backwards

D –rotates right

SpaceBar – shoots

Right Tank:

Up Arrow – moves forward

Down Arrow – moves backwards

Left arrow – rotates left

Right arrow – rotates right

Right enter – shoots

Game play: Use controls to move tank around, Destroy the other player to win. Pick up an extra life if you need it to keep the game playing!

Galactic Mail:

Player controls:

Left arrow – rotates left

Right arrow – rotates right

Space Bar– Blasts off

Game Play: Aim your ship and blast off into space. Land on a moon to deliver their mail. Each moon that you land on will add 500 points to your score, the longer you stay on that moon the lower your score will go. Be careful, the gravity of each moon is different making your ability to blast off dynamic. This added with the asteroid belt in the galaxy makes this game difficult!

6 Assumption Made

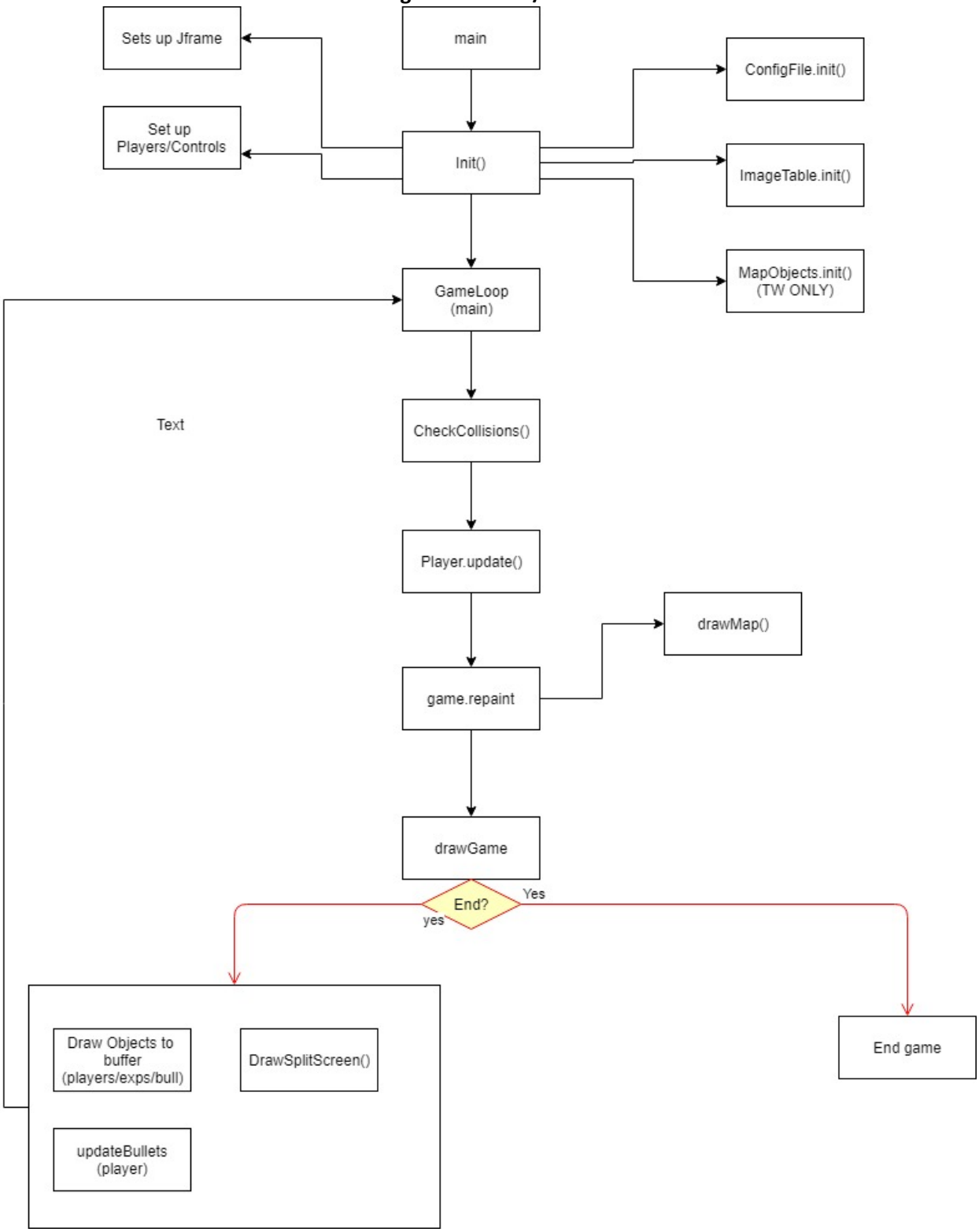
There weren't many assumptions necessary in these projects because the information was explicitly given. We were given free reign on what our power ups could be, how to handle damage, and the overall game environment. For Galactic Mail the assumption made was that there wouldn't need to be a "specific" map, being that every game play has the same levels/asteroid paths.

7 Implementation Discussion

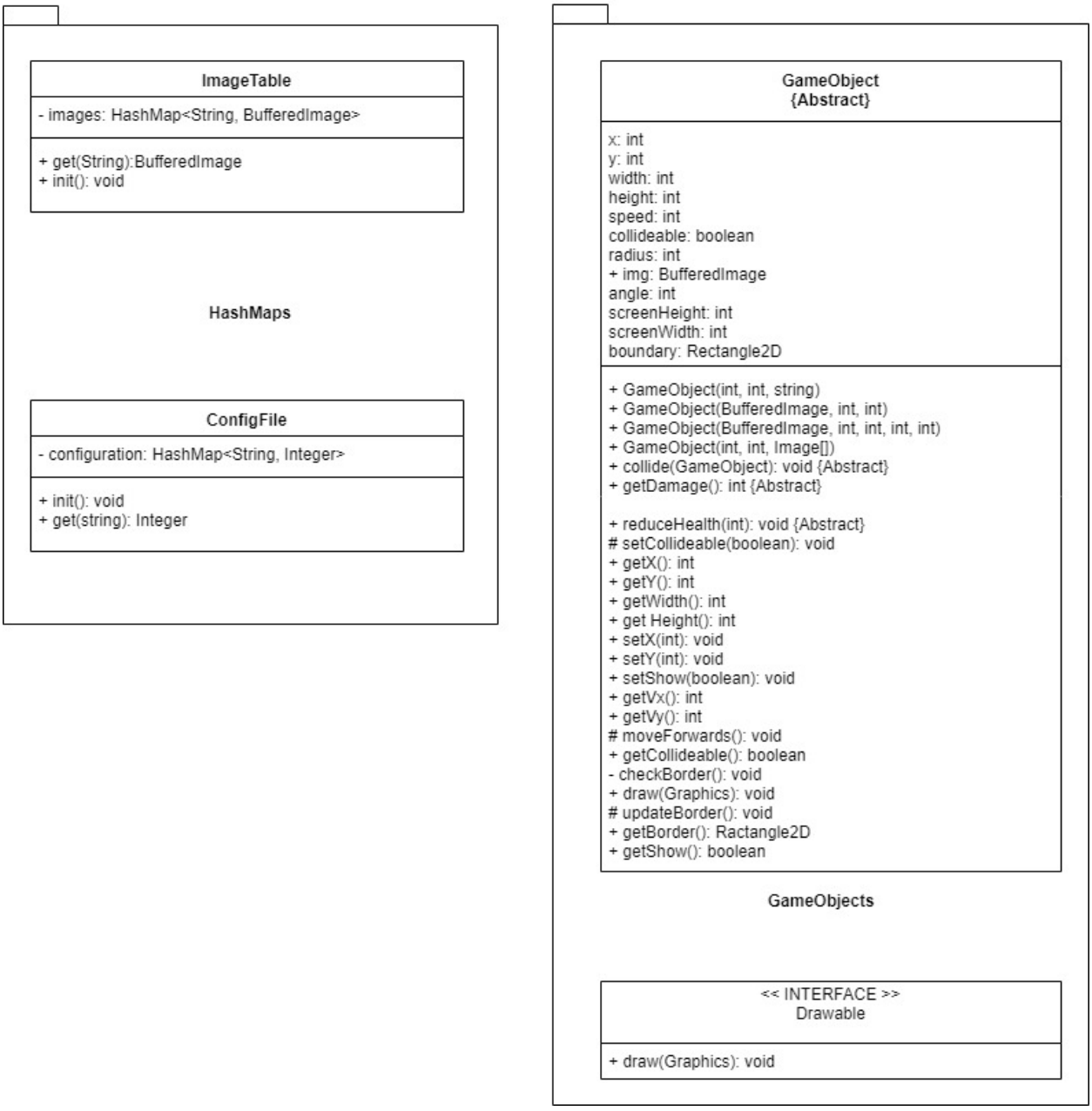
The implementation for both games drew focus from both the Tank Rotation Example (TRE) provided as well as pulling aspects and inspiration from the wingman game provided. Most of the implementation can be found in the technical overview.

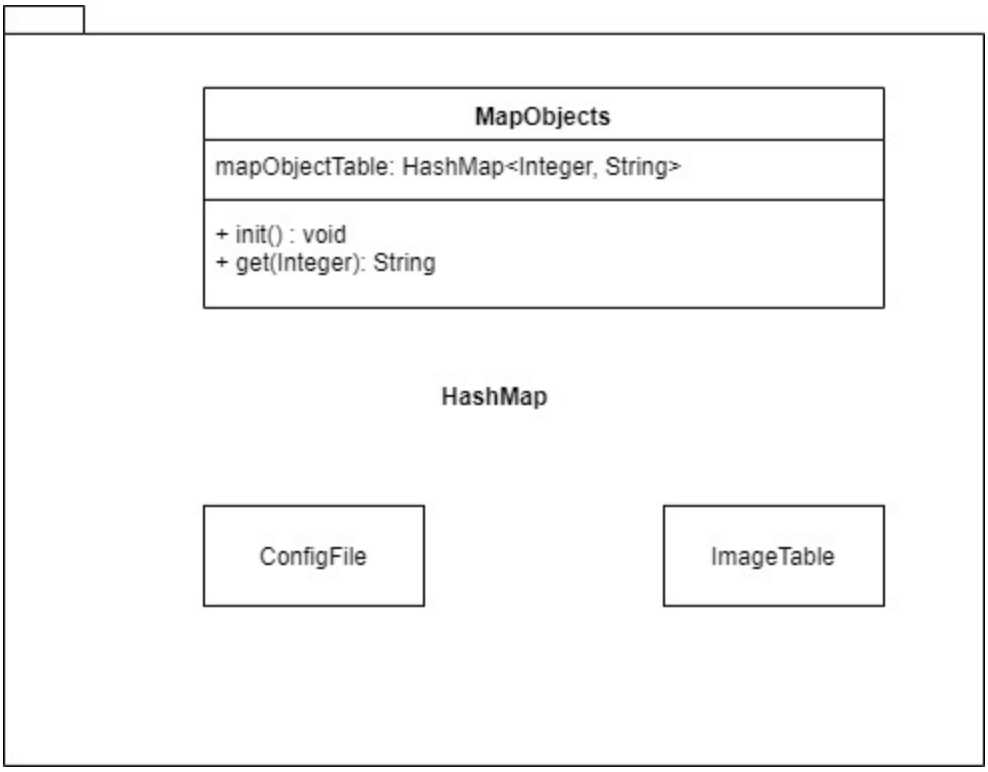
7.1 Class Diagrams

Shared Diagrams Classes/Game flow

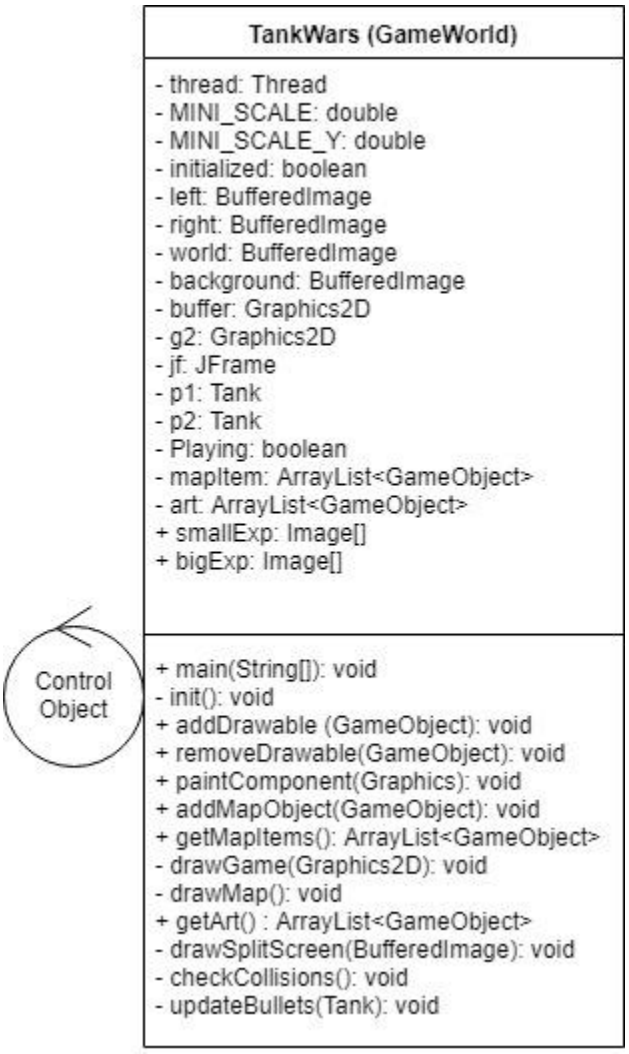


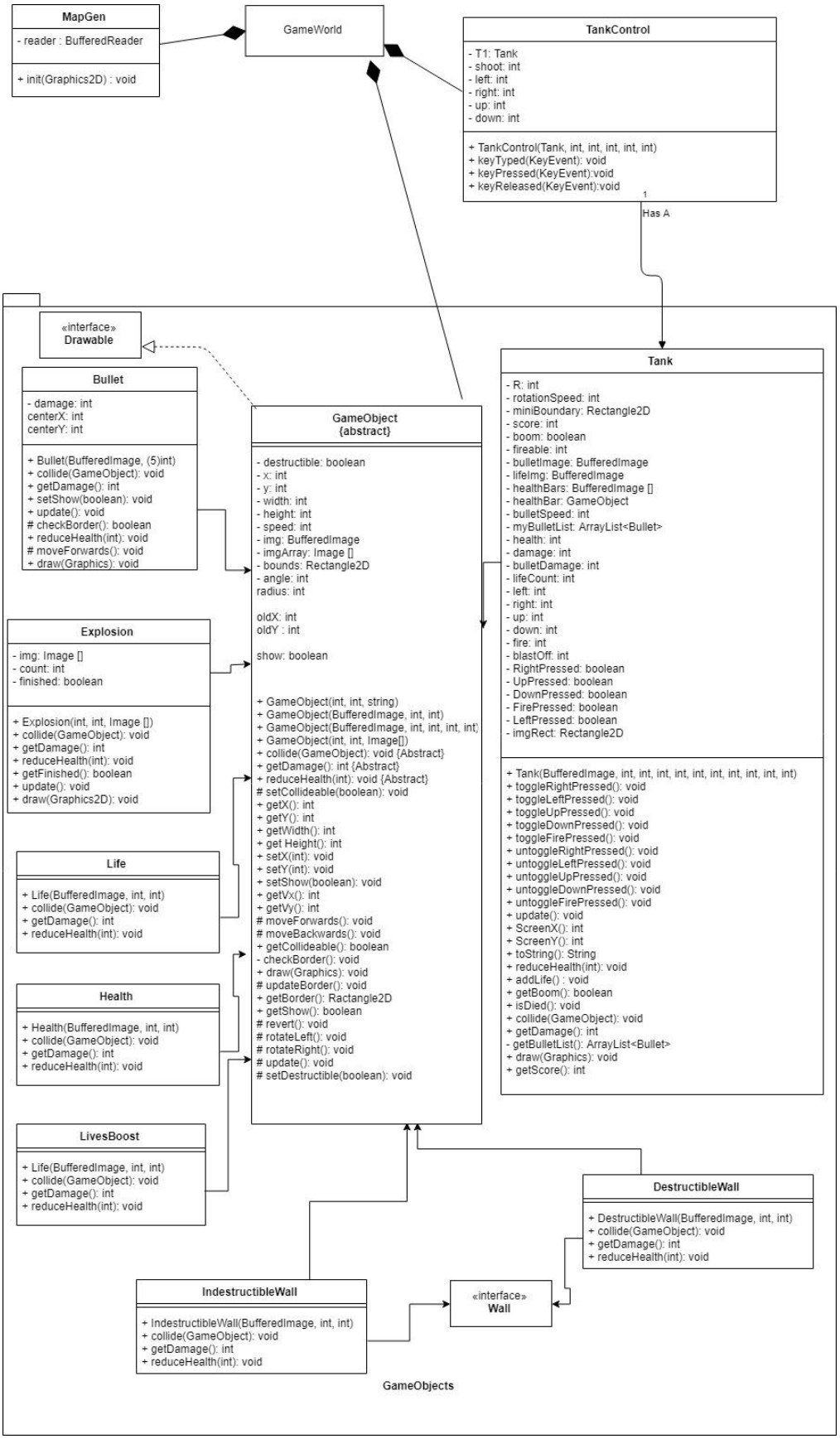
Shared Hierarchy (not all methods are used)



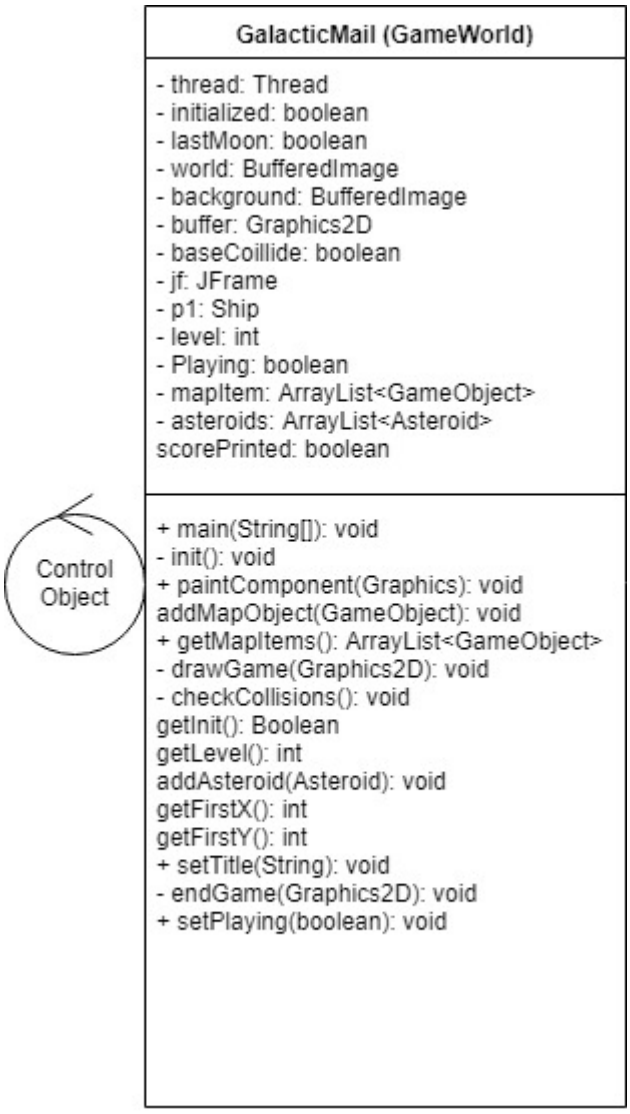

MapObjects
mapObjectTable: HashMap<Integer, String>
+ init() : void
+ get(Integer): String
HashMap
ConfigFile
ImageTable

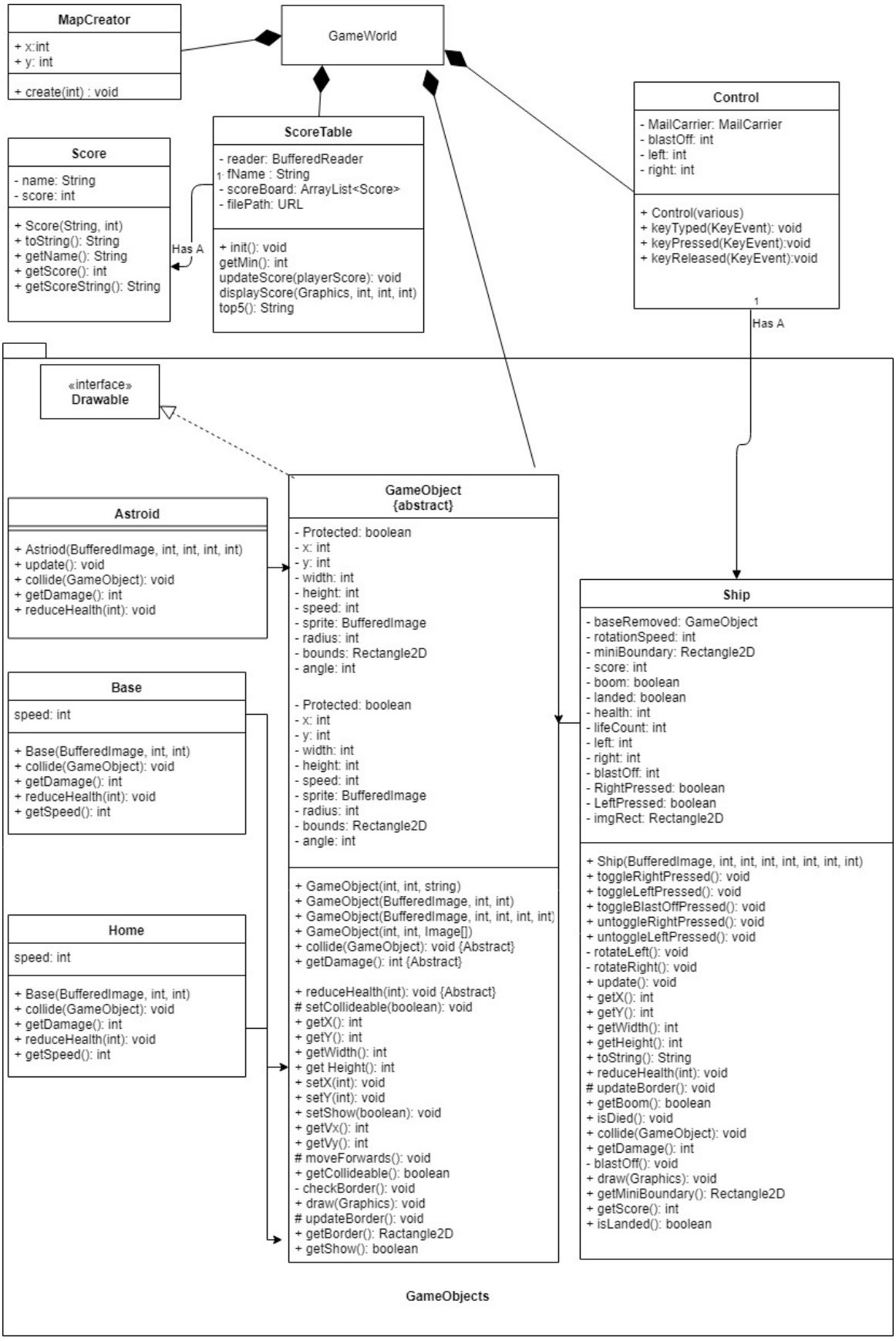
TankWars Diagrams




MapGen
- reader : BufferedReader
+ init(Graphics2D) : void
GameWorld
TankControl
- T1: Tank
- shoot: int
- left: int
- right: int
- up: int
- down: int
+ TankControl(Tank, int, int, int, int, int)
+ keyTyped(KeyEvent): void
+ keyPressed(KeyEvent):void
+ keyReleased(KeyEvent):void
1
Has A
«interface»
Drawable
Bullet
- damage: int
centerX: int
centerY: int
+ Bullet(BufferedImage, (5)int)
+ collide(GameObject): void
+ getDamage(): int
+ setShow(boolean): void
+ update(): void
# checkBorder(): boolean
+ reduceHealth(int): void
# moveForwards(): void
+ draw(Graphics): void
GameObject
{abstract}
- destructible: boolean
- x: int
- y: int
- width: int
- height: int
- speed: int
- img: BufferedImage
- imgArray: Image []
- bounds: Rectangle2D
- angle: int
radius: int
oldX: int
oldY : int
show: boolean
+ GameObject(int, int, string)
+ GameObject(BufferedImage, int, int)
+ GameObject(BufferedImage, int, int, int, int)
+ GameObject(int, int, Image[])
+ collide(GameObject): void {Abstract}
+ getDamage(): int {Abstract}
+ reduceHealth(int): void {Abstract}
# setCollideable(boolean): void
+ getX(): int
+ getY(): int
+ getWidth(): int
+ get Height(): int
+ setX(int): void
+ setY(int): void
+ setShow(boolean): void
+ getVx(): int
+ getVy(): int
# moveForwards(): void
# moveBackwards(): void
+ getCollideable(): boolean
- checkBorder(): void
+ draw(Graphics): void
# updateBorder(): void
+ getBorder(): Ractangle2D
+ getShow(): boolean
# revert(): void
# rotateLeft(): void
# rotateRight(): void
# update(): void
# setDestructible(boolean): void
Tank
- R: int
- rotationSpeed: int
- miniBoundary: Rectangle2D
- score: int
- boom: boolean
- fireable: int
- bulletImage: BufferedImage
- lifeImg: BufferedImage
- healthBars: BufferedImage []
- healthBar: GameObject
- bulletSpeed: int
- myBulletList: ArrayList<Bullet>
- health: int
- damage: int
- bulletDamage: int
- lifeCount: int
- left: int
- right: int
- up: int
- down: int
- fire: int
- blastOff: int
- RightPressed: boolean
- UpPressed: boolean
- DownPressed: boolean
- FirePressed: boolean
- LeftPressed: boolean
- imgRect: Rectangle2D
+ Tank(BufferedImage, int, int, int, int, int, int, int, int, int, int)
+ toggleRightPressed(): void
+ toggleLeftPressed(): void
+ toggleUpPressed(): void
+ toggleDownPressed(): void
+ toggleFirePressed(): void
+ untoggleRightPressed(): void
+ untoggleLeftPressed(): void
+ untoggleUpPressed(): void
+ untoggleDownPressed(): void
+ untoggleFirePressed(): void
+ update(): void
+ ScreenX(): int
+ ScreenY(): int
+ toString(): String
+ reduceHealth(int): void
+ addLife() : void
+ getBoom(): boolean
+ isDied(): void
+ collide(GameObject): void
+ getDamage(): int
- getBulletList(): ArrayList<Bullet>
+ draw(Graphics): void
+ getScore(): int
Explosion
- img: Image []
- count: int
- finished: boolean
+ Explosion(int, int, Image [])
+ collide(GameObject): void
+ getDamage(): int
+ reduceHealth(int): void
+ getFinished(): boolean
+ update(): void
+ draw(Graphics2D): void
Life
+ Life(BufferedImage, int, int)
+ collide(GameObject): void
+ getDamage(): int
+ reduceHealth(int): void
Health
+ Health(BufferedImage, int, int)
+ collide(GameObject): void
+ getDamage(): int
+ reduceHealth(int): void
LivesBoost
+ Life(BufferedImage, int, int)
+ collide(GameObject): void
+ getDamage(): int
+ reduceHealth(int): void
DestructibleWall
+ DestructibleWall(BufferedImage, int, int)
+ collide(GameObject): void
+ getDamage(): int
+ reduceHealth(int): void
IndestructibleWall
+ IndestructibleWall(BufferedImage, int, int)
+ collide(GameObject): void
+ getDamage(): int
+ reduceHealth(int): void
«interface»
Wall
GameObjects

Galactic Mail Diagram




MapCreator
+ x:int
+ y: int
+ create(int) : void
GameWorld
Score
- name: String
- score: int
+ Score(String, int)
+ toString(): String
+ getName(): String
+ getScore(): int
+ getScoreString(): String
Has A
ScoreTable
- reader: BufferedReader
1 fName : String
- scoreBoard: ArrayList<Score>
- filePath: URL
+ init(): void
getMin(): int
updateScore(playerScore): void
displayScore(Graphics, int, int, int)
top5(): String
Control
- MailCarrier: MailCarrier
- blastOff: int
- left: int
- right: int
+ Control(various)
+ keyTyped(KeyEvent): void
+ keyPressed(KeyEvent):void
+ keyReleased(KeyEvent):void
1
Has A
«interface»
Drawable
Astroid
+ Astriod(BufferedImage, int, int, int, int)
+ update(): void
+ collide(GameObject): void
+ getDamage(): int
+ reduceHealth(int): void
Base
speed: int
+ Base(BufferedImage, int, int)
+ collide(GameObject): void
+ getDamage(): int
+ reduceHealth(int): void
+ getSpeed(): int
Home
speed: int
+ Base(BufferedImage, int, int)
+ collide(GameObject): void
+ getDamage(): int
+ reduceHealth(int): void
+ getSpeed(): int
GameObject
{abstract}
- Protected: boolean
- x: int
- y: int
- width: int
- height: int
- speed: int
- sprite: BufferedImage
- radius: int
- bounds: Rectangle2D
- angle: int
- Protected: boolean
- x: int
- y: int
- width: int
- height: int
- speed: int
- sprite: BufferedImage
- radius: int
- bounds: Rectangle2D
- angle: int
+ GameObject(int, int, string)
+ GameObject(BufferedImage, int, int)
+ GameObject(BufferedImage, int, int, int, int)
+ GameObject(int, int, Image[])
+ collide(GameObject): void {Abstract}
+ getDamage(): int {Abstract}
+ reduceHealth(int): void {Abstract}
# setCollideable(boolean): void
+ getX(): int
+ getY(): int
+ getWidth(): int
+ get Height(): int
+ setX(int): void
+ setY(int): void
+ setShow(boolean): void
+ getVx(): int
+ getVy(): int
# moveForwards(): void
+ getCollideable(): boolean
- checkBorder(): void
+ draw(Graphics): void
# updateBorder(): void
+ getBorder(): Ractangle2D
+ getShow(): boolean
Ship
- baseRemoved: GameObject
- rotationSpeed: int
- miniBoundary: Rectangle2D
- score: int
- boom: boolean
- landed: boolean
- health: int
- lifeCount: int
- left: int
- right: int
- blastOff: int
- RightPressed: boolean
- LeftPressed: boolean
- imgRect: Rectangle2D
+ Ship(BufferedImage, int, int, int, int, int, int, int)
+ toggleRightPressed(): void
+ toggleLeftPressed(): void
+ toggleBlastOffPressed(): void
+ untoggleRightPressed(): void
+ untoggleLeftPressed(): void
- rotateLeft(): void
- rotateRight(): void
+ update(): void
+ getX(): int
+ getY(): int
+ getWidth(): int
+ getHeight(): int
+ toString(): String
+ reduceHealth(int): void
# updateBorder(): void
+ getBoom(): boolean
+ isDied(): void
+ collide(GameObject): void
+ getDamage(): int
- blastOff(): void
+ draw(Graphics): void
+ getMiniBoundary(): Rectangle2D
+ getScore(): int
+ isLanded(): boolean
GameObjects

8 Class Descriptions

Tank Wars:

Tank- This class held all the attributes for the Tank object for each player. It contained player location, health, bullet list, images etc. Tanks are an extension of GameObject. Some of these attributes are stored in the parent class.

Walls- There are 2 classes which share the Wall interface, which is a blank interface. Both destructible and indestructible walls are created by the MapGen class, and added to an ArrayList<Walls> to allow for faster looping of gameObjects in GameWorld while checking for collisions.

LivesBoost- This power up is also created by the MapGen and placed as an object on the map. These objects are also created under map objects for quick collision checks. Extends GameObject

Lives- Lives are created as objects and given to the Tank class for display. These lives are then displayed for each tank. Extends GameObject

Health- Heath is a class also assigned to the Tank and updated during the collision of the tank with a bullet. Each collision checks the damage and if the damage exceeds a threshold, then a new image is loaded to display health. Extends GameObject

Bullets- These are created by the tank object and stored in an Arraylist, updated through the tank and then drawn from their own DrawImage() method. Each carries their own damage, location and speed, this information is also stored in GameObject.

Explosions- These are created upon a collision of a Bullet and another Game object. The explosions have different sizes based on the type of explosion there is, small for damage, Large for destruction. Each collision creates one and adds to GameWorld's ArrayList of Explosions.

MapGen- This is used to create a map based on a Text file in the resources.

MapObjects—Hashtable used to define MapObjects while creating maps from file.

Galactic Mail:

Ship- This is just a reclassification of Tank from Tank wars, with a few methods and - Booleans changed for the particular implementation.

Asteroid- Asteroids are an adaptation of the BulletClass which is only created by the MapCreator Class(not the same as the TankWars class MapGen). This updates the asteroids location on a loop.

Home- This is a base which holds a special attribute that allows the ship to be landed without losing points. Created from the last location of the ship when transitioning levels or the start of the game.

Base- This object is used for scoring and moving on to the next level. As a player lands on a base the object adds 500 points to score.

MapCreator- This creates random maps based on the level. Each map has 5 randomly placed moons which check for collisions while creating the moon. Generates Asteroids and places them in a loop for drawing.

Score – objects held by score table.

ScoreTable—Class for retrieving file, reading file, checking for high scores, and updating file if needed.

Shared:

Control- This is what is used to listen on the JFrame to hear key strokes, this passes the information to the Tank class and handles the key stroke

GameWorld – This hosts the init functions of the game as well as the game loop used to control the drawing, collision checking and objects of the game.

ImageTable – updated images for the game based on images needed.

ConfigFile – Hashtable which contains information like map size, screen size, block size.

GameObject – basic information used for each object in the game, contains things like:

Coordinates, speed, radius, image, update function, etc.

9 Project Reflection

As with the other projects these games were fairly straight forward and not extremely difficult. The amount of information provided gave us a good basis for the game world/engine. The most challenging portion for me was the fact that I have no desire to do game development, which made it very difficult to "do my best". Total I spend about 25 hours on both games, which I am sure shows with the lack of additional powerups and only "checking boxes" to ensure I met the minimum criteria. The most problematic portion of the code was the actual painting of the game. Since we were given a working game which already implemented collisions, scores, moving pieces, etc. we had a solid foundation and plenty of resources to complete the project.

10 Project Conclusion/Results

These projects helped give us a look at dynamically/reoccurring updating programs. Since we are running a loop which updates every millisecond or so it gives us insight as to how quickly a well written program can run a few hundred lines of code. Overall, I feel as if the games did help me become a better programmer because I had to fight through something I dislike and achieve the desired end result. I could have done a better job in both games given that I had more of a desire to do it. After showing the game to my "test and eval" team, aka my kids, it allowed me to see that my logic was sound and didn't cause many bugs. It also gave me a little more pride in the work which allowed me to do a couple additional things in the tank wars game, like explosions. I think that the tank game could be cut

into a single term project. If students were given less resources up front and more time I feel it could benefit them more because they are forced to think about things a little more not just jump into the "example" which gives them almost the entire project only needing to sew it together. I do see how it would not yield all students a benefit because some will procrastinate.