Term Project

CSC 413 Section 2

Name: Kevin Baltazar Reyes
Student ID#: 916353599

https://github.com/csc413-02-fa18/csc413-tankgame-kay-rey

https://github.com/csc413-02-fa18/csc413-secondgame-kay-rey

Collaborated with: Abdi Mohamud, Amari Bolmer, Alex Wolski, and Ivan Briseno.

Introduction

The purpose of this project is to implement a two games, a tank game I named “Tanks For the Memories” and a game like Galactic Mail that I named “MoonGod.” These games will be written using Object-Oriented Programming.

Tanks for The Memories is a two player tank game that is controlled by one keyboard. It is split screen with a mini map in the middle. It also includes health bars for each tank, breakable walls, and health power up. The objective is to shoot the opposing player with the tank until their health runs out.

MoonGod is a single player space survival game. The lore of the game is a spaceship lost in space trying to get home. The spaceship finds that the moons offer guidance and shelter for the way home. The game includes asteroids, moons, the spaceship has health, progressively harder levels, and score. It is a game where the player controls a spaceship that doesn't stop and has limited mobility that has to fly from moon to moon. If the spaceship collides with the asteroid it will lose health and once health goes to zero then the game ends. Once docked on a moon, the asteroids cannot hurt the spaceship until the spaceship launched off the moon. The moon disappears once the spaceship launches off it, adding a point to the score and indicates that the moon has been landed on. Once all the moons are gone then the player goes to the next level which adds another moon and asteroid. The game gets progressively harder because every level has more moons the player has to land on and more asteroids that can hurt the ship.

Development Environment

The version of Java used in the project is Version 8 or more precisely Java 1.8.0_161. The IDE used for this project is JetBrains’ IntelliJ IDEA. The operating system used to write this program is the UNIX-based macOS High Sierra Version 10.13.6 (17G65).

Building Environment in IntelliJ

Both the projects can be built the same way.
Tanks For the Memories URL: https://github.com/csc413-02-fa18/csc413-tankgame-kay-rey
MoonGod URL: https://github.com/csc413-02-fa18/csc413-secondgame-kay-rey

1. Clone the project from GitHub using the URL
2. Open IntelliJ and click “Import Project”
3. Navigate to the cloned project

How to Run

Running Tanks For the Memories

1. Import the project in IntelliJ and have the project sidebar open where all the files can be seen
2. Navigate into the folder name ‘tanksforthememories’
3. Right click on TanksForTheMemories.java and click Run
4. Click Run to run the program in IntelliJ

Running MoonGod

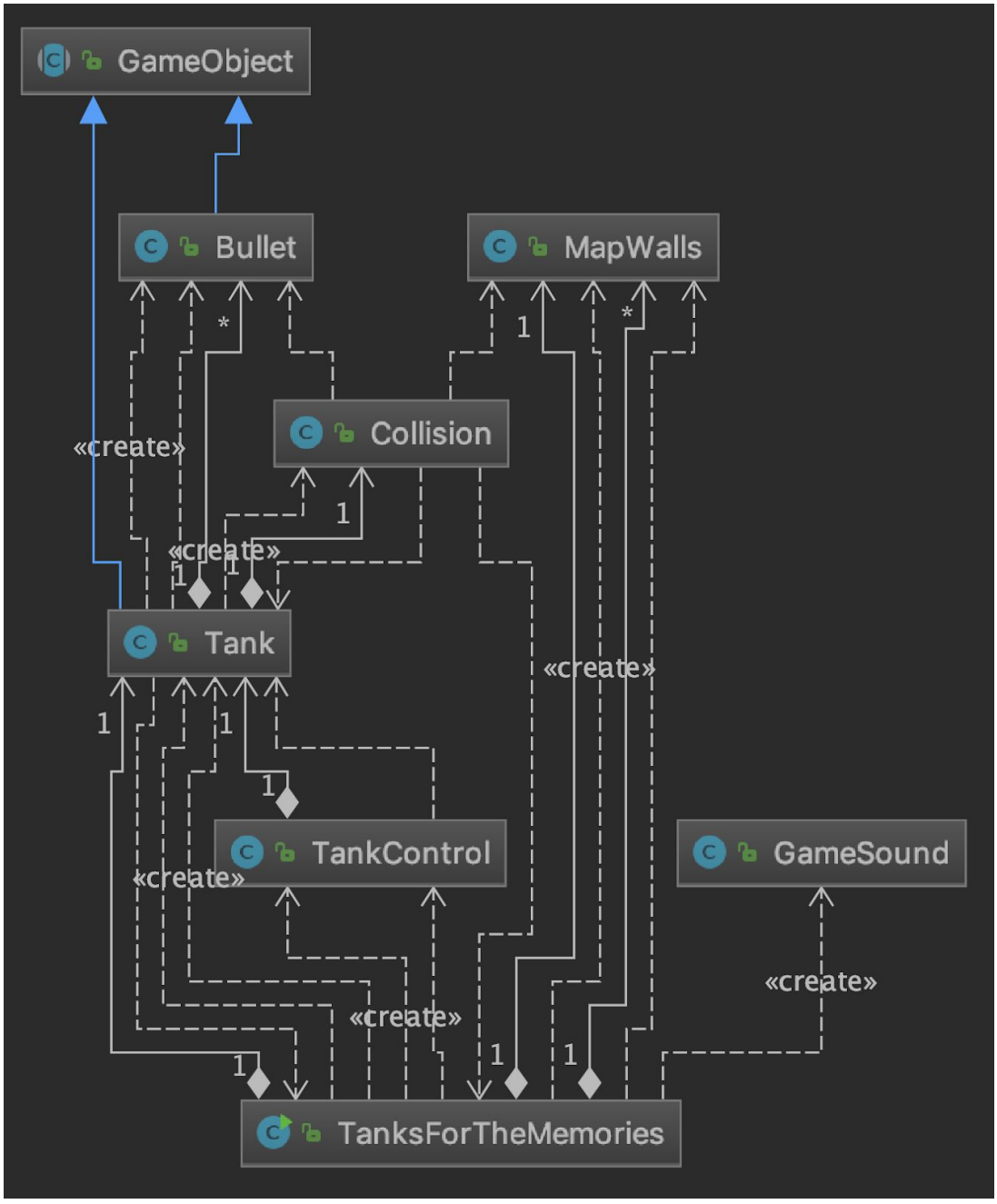
There are two ways to run the program.

1. Open the folder that was cloned from GitHub
2. Navigate and open the folder labeled ‘jar’
3. Double click the file labeled ‘csc413-secondgame-kay-rey.jar’ and the game will immediately start

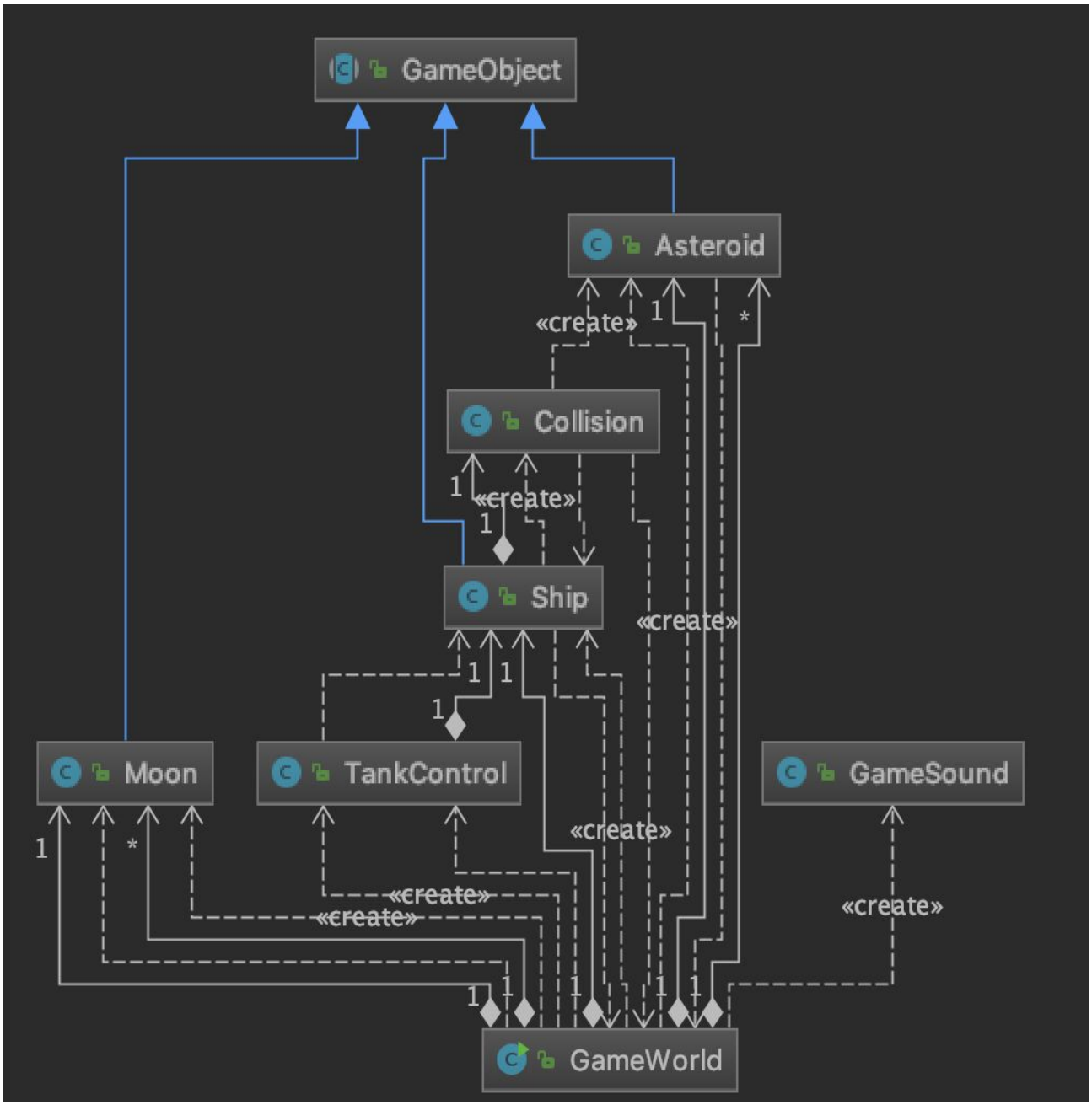
or

1. Import the project in IntelliJ and have the project sidebar open where all the files can be seen
2. Navigate into the folder name ‘moongod’
3. Right click on GameWorld.java and click Run
4. Click Run to run the program in IntelliJ

Tanks For the Memories Class Diagram



MoonGod Class Diagram



General Descriptions of Classes

General Descriptions of Classes Shared Between Both Games

There are a couple of core classes that were used in both games. These are very fundamental classes that enable the games to run while some made it easier to add to the game. Some of the classes shared had their name changed when implementing into the second game to make it clearer what the do. First is the GameWorld class, called TanksForTheMemories, in the first game. The GameWorld class is the main class that brings everything together and actually runs the game. GameWorld is responsible for making the objects as well as drawing everything. The GameWorld class is arguably the most important class in the whole game because it handles everything. An abstract class shared between both games is the GameObject class. This class is the building block for all game objects in the game.

Next class shared between both games is the Ship class which is called the Tank class in the first game, this class extends the GameObject class. The Ship class is what controls the ship object that the player is supposed to control. It handles the collision and appearance and overall all the operations that the player has to go through.

A class that is very closely connected with the Ship class and in both the games is the TankControl class. This class is in charge of reading all the inputs from the keyboard and passing those inputs to the rest of the program to tell the game what to do.

Collision is another class shared between both games. The Collision class is what handles collisions between game objects. The Collision class had to be heavily modified from the first game to the second game.

The last class shared between both the games is the GameSound class. This class is in charge of playing the theme music that is heard while playing the game.

General Descriptions of Tank Game Specific Classes

The classes that are only used in the Tank Game are the Bullet and MapWalls classes. The objective of the Tank Game is to bring the health of opposing player down to zero and the players do that by shooting bullets out of their tank and hitting the enemy tanks. MapWalls is the class used to create the map. I used MapWalls as a separate class because it let me create new maps easier.

General Descriptions of MoonGod Specific Classes

The classes that are used only in the game MoonGod are the Asteroid and Moon classes. These classes both extend the GameObject class and are essential to the MoonGod game. Asteroids randomly fly around the map and hurt the player if they touch one and the Moon object is used for the player to land on and progress to the next level.

Detailed Class Explanations

This section will go through every class in the game and give an explanation on what it does and how it works. Every class also has a picture of the constructor attached to the beginning.

Detailed Class Explanation Shared Between Games

This section focuses on the classes that are shared between both games as mentioned in the General Descriptions of Classes

GameWorld Class

The GameWorld class is arguably the most important class in the whole program. This class is what brings everything together. This is the class that contains the main method. The main method is pretty simple, it starts by making an instance of the GameWorld object and then calling the init function in the GameWorld class. What the init function does is it loads all the images and sounds that are going to be used in the game. The init function creates all the objects that are to be used in the beginning of the game including ships, asteroids, and moons. The init function also sets up the jframe and sets up TankControl so the keyboard can be used to control the players in the game. After the init function initializes everything, the program goes into a loop that doesn't end until the window the game is on is closed. Inside this loop all it does is update all the objects then repaints them. All the main method does is essentially update all the objects just using the update function for each object. This is the class that dictates the size of the display for the game as well as how many moons and asteroids the game starts with. All the game objects are stored in the class to make it easy to update everything since this is the main method. The only object not stored here are the bullets because those are stored with each tanks respectively. This is where everything is drawn as well. I use a buffer to put all the things that I want to draw. I draw everything from least important to most important because whatever is drawn last, is drawn over everything else. So I first draw the background image then I draw the map. After that I draw the objects excluding the ship. I draw the ship last because I want to make sure the ship is always visible. I print all the text including score and level here as well, I store that information elsewhere but call it in this class to print it out.

GameObject Class

```
public GameObject(int x, int y, int vx, int vy, int angle, BufferedImage img){
    this.img = img;
    this.x = x;
    this.y = y;
    this.vx = vx;
    this.vy = vy;
    this.angle = angle;
    this.width = img.getWidth( observer: null);
    this.height = img.getHeight( observer: null);

}
```

The GameObject class is an abstract class that every object in the game extends. This class is used as a building blocks to implement all the other objects. I tried to figure out what I needed in all classes and ended up having. The variables used in this class that every other object is going to use are x, y, vx, vy, angle, width, and height. X and Y keep track of where the object is. VX and VY is used to make the object move, it acts like the velocity. The angle dictates where the object is facing. The constructor also takes in a BufferedImage as a parameter. This image is used to graphically represent the object and the width and height of the picture itself. Other than the constructors, there are seven methods in the GameObject class that every object uses. There are getters and setters for the X and Y position of the object, this is important to keep track of where every object is. There are also getWidth and getHeight methods that return the width and height of the image used to represent that object. The last method in this class is the draw method. This method is a skeleton of how to draw each respective object.

TankControl Class

```
public TankControl(Ship t1, int up, int down, int left, int right, int shoot) {
    this.t1 = t1;
    this.up = up;
    this.down = down;
    this.right = right;
    this.left = left;
    this.shoot = shoot;
}
```

The TankControl class implements the KeyListener interface. KeyListener is used to receive keyboard events (keystrokes) and process them. The TankControl grabs these keystrokes and performs a different action depending on what keystroke is being activated. You pass in the Ship object that you want to control as well as which buttons are going to control it. There is an up, down, left, right, and shoot command that can be configured to a key. The TankControl class has a method that tells when one of the keys are pressed and another method that tell you when the key is let go or unpressed. These methods then tell the Ship object that the respective button has been pressed then its up to the Ship class to process that information.

Ship Class

```
Ship(int x, int y, int vx, int vy, int angle, BufferedImage tankImg) {
    this.x = x;
    this.y = y;
    this.vx = vx;
    this.vy = vy;
    this.angle = angle;
    this.tankImg = tankImg;
    this.tankMainImg = tankImg;
    this.isDocked = false;
    this.isLaunched = false;
}
```

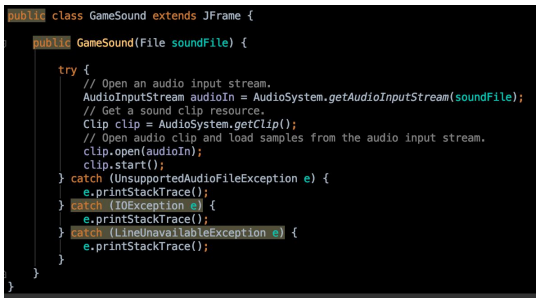
The ship class is what implements the ship that the player controls. In the tank game, this class is called Tank. This class extends the GameObject. A couple of important variables in the Ship class are the booleans isDocked, isOver and isLaunched. Other variables include rotation speed, health and speed. There are setters and getters for isLaunched, isDocked, X, Y, width, and height. The isDocked method also increases the score once it goes to false to signify that the ship has left a moon and acquired a point. The keystrokes from TankControl are passed to the Ship class and from there each button does a different operation. The left and right buttons change the angle to change the direction that the ship is pointing. This is the class that has the most methods because it is the class that has to perform to most actions. An important class is called isHit and this method is called in the collision class to indicate that the ship was hit by something that lowers its health. The isHit method first lowers the ship's health then checks how much health the ship has left and if the ship has zero or less health then the isOver boolean is assigned to true and when the GameWorld class sees that isOver is true then it will end the game. A Game Over screen is shown on the screen until the game is restarted. Then there is a drawHealth class that reads the health that the ship has and changes the color and size of the health bar accordingly. There is a checkBorder method that is used in the methods that make the ship move, the checkBorder method checks if a player goes outside the playing area and instead of changing the position, it keeps the position is bounds. The getRect method returns a rectangle based on the size of the picture of the ship. The most important method is the update method because that is what is constantly being called in the main method. The first thing the update method does is check for collisions, it will go to the collision class and check all the collisions for each tank. I check for collisions first because I don't want to move the ship before knowing if the ship is going through another object. Then update checks which button is being pressed and based on that it will call the corresponding movement method associated to that keystroke. Pressing up will make the moveForward method be called then pressing left will have the moveLeft method be called and so on.

Collision Class

The collision class is what processes when two objects intersect and cause a collision in game. This class changes from each game but they work the same way. To use this class you pass in a ship object that you want to check for collisions. In the method call the ArrayList with the objects that you want to check whether the ship is colliding with, for the tank game you want to check walls as well as bullets, in the second game you want to check for moons and asteroids.

The way that the collision works is that a rectangle is made for every object and collision checks if those objects collide. If an object collides the collision class will then check what object intersected the ship. Depending on the type of object, the collision will be different. The collision class is ran every time the ship object is updating, this makes it so that each ship in the tank game checks for any collisions.

GameSound Class

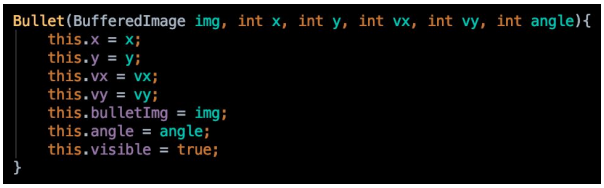


```
public class GameSound extends JFrame {

    public GameSound(File soundFile) {

        try {
            // Open an audio input stream.
            AudioInputStream audioIn = AudioSystem.getAudioInputStream(soundFile);
            // Get a sound clip resource.
            Clip clip = AudioSystem.getClip();
            // Open audio clip and load samples from the audio input stream.
            clip.open(audioIn);
            clip.start();
        } catch (UnsupportedAudioFileException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (LineUnavailableException e) {
            e.printStackTrace();
        }
    }
}
```

The sole purpose of the GameSound class is to play the theme music while the player is playing the game. The only parameter for GameSound is an audio file. That audio file gets loaded then immediately starts playing the song all the way through. The song is loaded in the init method of GameWorld and the GameSound method is called there too.

Detailed Tank Game Class Explanation

Bullet Class



```
Bullet(BufferedImage img, int x, int y, int vx, int vy, int angle){
    this.x = x;
    this.y = y;
    this.vx = vx;
    this.vy = vy;
    this.bulletImg = img;
    this.angle = angle;
    this.visible = true;
}
```

The Bullet class is used to create bullet objects which are Tank sub-objects. The Bullet objects are created by the tank anytime a player presses the fire button. There is a loop in the fire button which only registers every other button press because if you held down the fire button then it would keep shooting really fast. When the bullet is created, the tank that created it passes its own variables into the bullet parameters, this is because we want the bullet to be consistent. By giving the tank and the bullet the same variable values we guarantee that the bullet shoots out from the middle of the tank. When drawing the bullet, we draw the bullet first then update its coordinates, if I don't draw the bullet first then the bullet will look like it did not even hit the tank because it will think it hits the tank before drawing the bullet. The ship class has an ArrayList that stores all the bullets for that tank so when I draw the tanks, I go through both of those ArrayList and draw everything in the ArrayList then update. Bullets cannot be made without a tank as bullets are sub-objects for tanks.

MapWalls Class

```
MapWalls(int x, int y, BufferedImage wallImg) {
    this.x = x;
    this.y = y;
    this.img = wallImg;
}
```

The MapWalls class is how the level is drawn out in the tank game. There are two types of walls, breakable and unbreakable walls. The way that I draw out that map is I use a 2D array and to distinguish whether it was going to be an empty space, unbreakable wall, or unbreakable wall I numbered them 0, 1, and 2 respectively in the array. I used a 32x32 2D and each value in the array represents one block. I chose to do it this way to make it easier to change maps and levels. It was easy being able to just change a number on an array to change the type of wall that the MapWalls constructed. The MapWalls class is only called in the init of the GameWorld class when init is loading everything. The class is called once to make all the walls then stored in an ArrayList to be able to manipulate later if the tank breaks a breakable wall.

Detailed MoonGod Game Class Explanation

Asteroid Class

```
Asteroid(BufferedImage asteroidImg) {
    this.x = getRandomNumberInRange(50, 1150);
    this.y = getRandomNumberInRange(50, 900);
    this.vx = getRandomNumberInRange(-2, 2);
    this.vy = getRandomNumberInRange(-2, 2);
    this.angle = 0;
    this.asteroidImg = asteroidImg;
}
```

The Asteroid Class is a very similar class to that of the bullet class in the tank game. The asteroids represent obstacles for the player as they are trying to get to the next level. The only parameter the asteroid class has is the image that is used to represent it. All of the values are essentially random, I randomly create this to make different experiences every time anyone decides to play the game. The asteroids fly around randomly and when at an edge of the screen the asteroid goes to the other side to make it seem like the whole screen is connected. This style of movement true for all the objects in MoonGod. If an asteroid hits the ship, it does damage to the ship then the asteroid object gets erased and a new one is created at a random location.

Moon Class

```
Moon(BufferedImage moonImg) {
    this.x = getRandomNumberInRange(50, 1150);
    this.y = getRandomNumberInRange(50, 900);
    this.angle = 0;
    this.moonImg = moonImg;
    this.vx = getRandomNumberInRange(-2, 2);
    this.vy = getRandomNumberInRange(-2, 2);
}
```

The Moon class is almost identical to the Asteroid class. The moon class acts like a safe haven for the ship to land on. When created, the moons will spawn at random locations in the game. The moons do not move though so when the moon gets updated, it will only spin at a stationary position. When a ship lands on a moon it gets docked, once the player undocks from the moon, the moon will disappear and add a point to the player's score.

Assumptions Made

My biggest assumption and underestimation were the collisions. The problem seemed easy when first thinking about it but once I started to implement the collisions I saw all the issues that it brings. At first, I thought all I would have to do to implement collisions is to observe if two objects intersected with each other. I did not realize that was just the tip of the iceberg. Out of the whole project, I would say collisions took the most time out of everything to make. In Tanks for the Memories, I essentially had to manually go through every object and check if it collides with the tanks. It was very tedious adding another object because if I have at add another object, I would also have to add it to the collision watch list manually. If not, then no other object could interact with it. My abstraction was not optimal.

My biggest assumption for the second game 'MoonGod' was how to construct a level. I did not plan very well when implementing the game. I worked very incrementally only adding things when I realized I needed them. It made me flow better while working on it but in the end I realized I needed to do a lot of clean up because it was obvious that a lot of the aspects of the game did not work well with each other. The biggest issue in the end was creating a new level, while making the level I had not thought about when the level would finish and how to start a new level.

In the end, the project would've been doner faster and cleaner if I had planned ahead instead of implementing parts when I need them.

Conclusion (Reflection and Results)

In conclusion, this project was a lot of fun for me. I just wanted to make some games that I would be proud of and along the way I feel like I got a deeper understanding of Object Oriented Programming. The tank game was very hard to start up and understand how everything worked. I had to learn a lot of new things the get the tank game to run which got in the way of me working on Object Oriented Programming. Once I had to turn in the tank game I started to understand how everything worked. By the time I had to work on the second game, I feel like understood how to make a game easier. I had a better grasp of Object Oriented Programming as well which made my code a lot cleaner and helped when creating new objects using inheritance.