Team 22

John Mendoza & Naylin Min

CSC 413: Software Development

T, TH: 7-8:15

Spring 2018 Semester

Tank Game: https://github.com/csc413-02-sp18/csc413-tankgame-Team22
Super Rainbow Reef: https://github.com/csc413-02-sp18/csc413-secondgame-Team22

Introduction:

For our semester project, we coded two games from scratch. For the first game, we implemented a 2-player Tank Game. For our second game we got to choose the game to implement, and we choose to do Super Rainbow Reef, a brick breaker game.

For the Tank Game, the idea is for two tanks to try and destroy each other and get rid of all their lives. The requirements are to have 2 players, with both controlling their own tank. The two tanks should be able to move up and down and to rotate left and right. The movement and the rotation of the tanks should be smooth. The tanks should be able to shoot bullets which would do damage to the other tank. The game should also include a split screen for the players as well as a mini-map. Both players should have health bars and life count, and both should decrease properly. The map includes breakable walls and unbreakable walls as well as a power up of our choosing. The collision between tanks, walls, powerups, and bullets are to be handled accordingly.

For our second game, we choose Super Rainbow Reef. In Super Rainbow Reef, *BigLegs* had invaded Rainbow Reef and the duo of *Pop* and *Katch* must stop them to save their home. The idea of Super Rainbow Reef is for *Pop* to defeat the *Big Legs* on the map with the help of *Katch*. However, the *BigLegs* are often hiding behind the coral defenses. Much like a typical brick breaker game, Pop bounces off *Katch* and off the coral defenses(bricks) and the *BigLegs* on the map. If *Katch* fails to catch *Pop*, *Pop* will fall out of the level and lose a life. The requirements of Super Rainbow Reef are that *Pop* would move realistically. If *Pop* bounces off the left-end of *Katch*, *Pop* should bounce toward left side, while bouncing off the right-side would send Pop right. Left and right arrow keys would move *Katch* horizontally to catch *Pop*. There should also be “gravity” in the game as well, where *Pop* falls back towards *Katch* each time he is in the air. The game is to have several levels, containing a number of *BigLegs* to defeat. Breaking the coral defenses (bricks) as well as defeating the *BigLegs* will accumulate points. There are to be special blocks that give the player extra rewards, but these special blocks and the bricks don’t have to be destroyed in order to finish a level. If *Pop* and *Katch* defeat the *BigLegs* in all the levels or if *Pop* falls off the map with no lives left, then the game will end.

For both our projects, the execution and development environment were the NetBeans8.2 IDE. We also pushed all of our changes to our github repo, so that both of us could have the latest version.

We were able to complete the majority of the requirements for the tank game, but we were not able to get a fully functioning split screen. For Rainbow Reef, we were able to add the majority of our desired goals – different bricks with different scores, a main menu, sounds , and a high scores table – in addition to the basic requirements of the game.

Build and Run Instructions:

To build both games, simply open the projects in the IDE of your choice. Next, for Rainbow Reef, set the working directory to the RainbowReef folder, and for the Tank Game set the working directory to the tankGame folder. Then to both compile and run, hit the play button in your IDE.

Assumptions

For the tank game, we assumed that the game did not need a score, sounds, or a menu, because it was our first game, and we focused on developing the core gameplay features of the game. As a result of these assumptions, the tank game does not feature any of the aforementioned features, but instead we ensured that we did our best to hit the core requirements. For Rainbow Reef, we assumed that our game would not only need the core features of gameplay, but also those additional quality of life features such as a main menu, sound, score, and multiple levels, so we added those the game.

Rules & Controls:

Tank Game:

Each player gets 3 lives to begin the game. The goal is to defeat the other player by shooting at tank until their health bar is at zero. If a player loses all 3 lives, the game will end. Player 1 controls: Up - move up, Down - move down, Left - rotate left, Right - rotate right, and Space - shoot. Player 2 controls: W - move up, S - move down, A - rotate left, D - rotate right, and Q - Shoot.

Super Rainbow Reef:

The main goal is for *Pop* and *Katch* to clear the coral defenses and defeat the *BigLegs* in all the levels. Secondary goal is to get the high score for the game. *Katch* controls: Left - move *Katch* left and Right - move *Katch* right.

Class Diagram:

Tank Game Class Diagram:

Super Rainbow Reef Class Diagram:

Shared Classes:

Even though the two games are fundamentally different in game play, controls, and rules, there are a lot of overlapping ideas and classes that are used in both games.

The classes that were used in both games are Background class, GameEvents class, GameObject class, Controls class, Game class, and Window class. **Background class** handles the background image on both classes. **GameEvents class** extends Observable and handles all the synchronizing of Observer objects. **GameObject class** is the base class used in both games. GameObjects have commonalities such as coordinates, images, states, and methods for rendering and updating. GameObject implements Observer so that all the subclasses from GameObject get "observed" by our GameEvents class. GameObject class takes in x-y coordinates and bufferedimage, since all the objects will have an image associated with it and their x-y coordinates of the object. **Control class** handles key presses for both games and determined the controls of the game. **Game class** is our main class that put all the pieces together. In this class, the game loop is handled as well as any threads that we might use, such as for soundplayer. The game class is a JPanel in which rendering and updating of objects takes place via ArrayList data structures. For Rainbow Reef, the game class also handles controlling the state of the game, and each GameObject uses composition by taking in the current instance of the Game Class as an argument, so that events from GameObjects can manipulate the state of the game. **Window** is the frame in which the Game panel is placed.

Tank Game Classes:

Classes that are specific to the Tank Game are Tank class, Bullets class, Wall class, and Powerup class. **Tank class** handles both player tanks. It takes in x-y coordinates, a bufferedimage, and tank ID in the constructor. The ID is used to differentiate between player one and two's tank. The tank class also extend GameObject class and implement Observer. In this class, the movement of the tank, firing bullets, collision with other tank, power up effects, and live and health checking is handled. In our **bullets class**, the constructor takes in x-y velocity, damage of the bullet, and the angle of the tank, on top of x-y position coordinates and bufferedimage. In this class, collision between bullet and tank is handled. Our **wall class** constructor takes in x-y coordinates and a bufferedimage. This class handles both breakable and unbreakable walls. This class also handle wall collision with tanks and bullets. **Powerup class** is similar to our breakable wall from our wall class. It handles collision with tanks and the respawn rate of the power up once it is collected.

Super Rainbow Reef Classes:

Classes that are specific to the Super Rainbow Reef game are Player class, Pop class, BigLeg class, Bricks class, SoundPlayer class, MainMenu class, MouseInput class, and TiledMap class. **Player class** handles *Katch's* movement and updating. **Pop class** handles *Pop's* movement from collisions, and it defines the behavior of collisions with bricks, power bricks, and katch. Players have no real control of *Pop's* movement, instead, the movements are determined by collision between *Pop* and other game objects such as bricks, *BigLegs,* or *Katch*. **BigLeg class** is for creating BigLeg objects. The class constructor take in x-y coordinate and a bufferedimage. **Bricks class** handles all the breakable bricks in the game. We used a hashmap to hold all the bufferedimages, and depending on which key we used to call the bufferedimage of the brick from the constructor, they would be assigned different brick IDs. These brick IDs determine the score of the brick or effect of the brick. **SoundPlayer class** is used for our background music. **MainMenu** and **MouseInput classes**, work together to get our title screen working. MainMenu class display the title page with options while MouseInput class handles the mouse clicks to pick the options. **TiledMap class** is used to read out .txt file of our levels. We used a .txt file with characters to build our levels. TiledMap class takes in the file name and build the map based on the symbols in the text file, and it uses a 2D array to arrange them in the coordinate space.

Team-reflection:

This was one of the toughest projects to get things rolling because it was our first time coding a big project from scratch. One of the hardest and most time-consuming part of this project is understanding Java swing components and how to use them. It took us about 2-3 weeks just to get a tank on a window that moves. We started out with Canvas to get the tank displayed and moving. However, we ran into a bunch of issues using it. One of which was the tank would leave a trail of the image as it moves. From the advice given in class, we decided to move on from Canvas and switch to JPanel. Getting the tank to move smoothly and understanding how it worked was also difficult at the start. Once we got the tank displayed and moving properly, the project progressed a lot smoother. Collisions were handled using rectangle at first. However, we found that rotating rectangles, such as the tanks, and using the built-in rectangle intersect method were returning strange results. We decided to switch the hitbox for tank into a circle to make rotations smoother. Another issue we ran into was split screen/mini map. We ended up using the strategy of building one bufferedimage with all the game objects in it and using subimages and scaling to get the split screens and minimap working. Another time-consuming part about our first project was playing with the x-y coordinates and trying to display the objects in the correct locations.

For our second game, we had a better understanding of rendering with swing components and trying to figure out how to manipulate the Java graphic library for rendering. We were able to reuse a lot of the code in our first game. We used the same strategy for border checking, collision, and overall class diagram. However, there were still issues that had to face. One of which was collision between *Pop* and the bricks. Instead of breaking one brick and bouncing off it, *Pop* would keep going and breaking all the bricks in the column. Another issue we encountered was Pop getting stuck in bricks bouncing back and forth. To solve these issues, we changed the behavior of Pop-Brick and Pop-Katch collisions to ensure smooth and fun movement.

Summary of work:

Naylin: I help code collisions logic and power ups for the tank game. Helped debugged collision glitches on both games. I also help implement bullets and handling of it. Helped implement “zones” for *Katch* for our second game but we ended up not using it. I also helped with high-score keeping and display logic. We met up about 2-3 times a week to work together and talk over issues and how to fix them.

John: I helped with the data structures, updating, and rendering for the first game. I also added small tweaks when needed to the tank game. For the second game, I also did the same, and I wrote the menus and sound.

Project Conclusion:

This project taught us much about creating GUIs, rendering, and the observer pattern. It also introduced us to threading in Java and creating music and sound. This project also solidified our understanding of data structures, and when and how to apply them. It also introduced us to the project design process on a large scale. Designing the projects from the ground up was difficult, because we would often have to go back and change our ideas or the code. We overcame these challenges by taking a more careful approach in design for the second game, and we really tried to think about how to create implementations before going in and coding them. This project will help us greatly in any future game or GUI development we do, and it also will help us in designing future projects we take on, because we now know what it is like to design a large project with little direction.