Team E for Effort:

Tank Game & Galactic Mail Documentation

Andre Leslie ID: 916849302	**Joshua Lizak** ID: 915490880

Tank Game

GitHub Repository

https://github.com/sfsu-csc-413-spring-2018/term-project-team-e-for-effort.git

Project Introduction

For this assignment we were tasked with developing a solution for a simple two dimensional tank game. The caveat we were given was the task of making our program as object oriented as possible. With the knowledge in mind that we were to create a second game based off of the code for the original, we needed to make as many of the functions as portable as possible. The basis of this specific was to create a basic 2D game based around two tanks on a map. These tanks are able to move around, hindered only be each other and by destructible and indestructible walls. The tanks are able to shoot each other and the destructible walls. Once one tank destroys the other, the game is over. Upon the completion of the first game we ported some of the code over to our second game.

Execution and Development Environment

Written with NetBeans 8.2 using Java 8 / JDK 1.8 on macOS 10.13.4.

Scope Of Work

This game required the development of a game engine for displaying premade frames on the display, game objects, and a game clock to update items that move on their own. The game engine was to be tasked with initializing most of the game objects and then displaying them upon each frame call. A basic object was required for handling all objects used. Further from that, specific objects were created for things that required more information to be stored. Further from that we had to create a player view controller that takes in the full image of the map and breaks it down into two cameras that follow

the player's tanks around the map. Along with the player views we created a minimap creation class as well as the health bars.

Assumptions

The game demonstration that we were provided, Wingman, was not a strong example of portability that we could base our project off of. Due to this, we had to make many assumptions as to what the functions required of the game are. The methods by which the game engine and its subsequent game thread are ran were originally not made clear to us.

Command Line Instructions

Navigate to proper directory:

navigate to .../term-project-team-e-for-effort/Submission/Tank

To Compile:

javac tankgame/TankGame.java

To Execute:

java tankgame/TankGame

Implementation

Package Organization:

- Tank Game
  - TankGame
- Animations
  - Explosion
  - Sprite Loader
- Controllers
  - Collision Detector
  - Game Clock
  - Input Controller
  - Key Object
- Map
  - Game Frame
  - Game Menu
  - Game World
  - Wall List
- Player
  - Player view
  - Mini Map
  - Health Bar
  - Winner Display
- Objects
  - Basic
  - Bullet
  - Tank
  - Wall
- Resources
- Sounds
  - Game Sound

Order Of Operation:
In the main method, held within the TankGame.java, the GameFrame and GameWorld are both created. A Once they are both initialized, the game world is passed into the game frame which then displays the game world panel extension onto the game frame JFrame.

Upon initialization of the GameWorld class, it creates an instance of the Game Menu and an Input Controller. This game menu is an extension of the basic game object and is created then displayed by the GameWorld. The Input Controller takes all of the input from the keyboard and translates it into a special Key Object. These key objects are stored in an ArrayList that is then passed to whatever task requires input such as the GameMenu. The Game Menu allows the player to select which version of the game they would like to play (Networked or Local). Depending on the selection of the player, the Game Menu will set the GameWorld into either one of three states: Local, Host, or Connect.

Once the proper state of the GameWorld has been set, the GameWorld will produce the proper version of the player view and begin rendering out the game's assets. The GameWorld contains a thread which is responsible for running the game. Each time this thread updates, a new frame is produced.

The tank objects are created separately in the GameWorld and are set to respond to the previously created input controller. The tank object can take on two distinct states: TankOne or TankTwo. The state of the tank determines which side of the display the tank will be shown on as well as setting which keys control the tank.

The destructible and indestructible walls are produced in a class called WallList which is called by GameWorld. This Wall List class creates all of the wall objects, sets their destructibility, and adds them into an ArrayList. This array list is then passed into the GameWorld class for display on the screen.

Bullets, explosions, and collisions are created on demand depending on the conditions in the game. Bullets are saved within their corresponding tanks in order to determine ownership. Explosions are rendered onto the display when a collision with a bullet is detected. The Collision Detector is called by the GameWorld once each frame. The Collision Detector is passed all of the tanks, walls, and bullets and determines if any of them are colliding with each other. Upon any collision, tank movement is adjusted or an explosion is called.

If one of the tanks is destroyed, the Winner Display class is initialized and called by the Game World. At this time, the input to the tanks is suspended and a message is displayed on screen saying which tank has won.

Results / Conclusion

Development of this game was interesting for our team. One of our team members had already completed this class in the past and had previously done some work on this Tank Game. Because of this we had a small bit of insight on how to develop this game, however this did not mean that we had no obstacles to overcome.

Creating the basics of the game was easy and was completed early in the development process. Our challenges came with the extra refinement that we put into the game. Our team member's previous implementation of the game was not ideal to say the least and it required a lot of modification if it were to operate at the standards we had set for ourselves. May changes were made in the ways that object drawing was called and the encapsulation of many functions in the game. We also added other features that were not previously present in the other implementation, such as having game states and a menu system. Input was also adjusted for more encapsulation.

We initially set a goal to finish setting up networking for the game, however due to time constraints and complexity we were unable to finish this feature. We were, however able to finish setting up the UI for the networking. Upon startup of the game, the menu will prompt the user asking if he/she would like to set up the game as local or networked. If a networked option is selected, the menu will ask for an IP address or port number to create or connect to and then a single player view of the game will be displayed on the screen. We were unable to finish the actual networking backend of the game and so we have the UI which is, unfortunately, not functional.

Other than the networking, we accomplished all of the other goals we set for ourselves; The game runs smoothly and produces no errors, movement is smooth, explosions work properly, all game sounds work as intended, and everything is proper. The only issue we had with our final implementation would be further encapsulation of functions and generalization of functions to make them more portable.

Galactic Mail

GitHub Repository

https://github.com/sfsu-csc-413-spring-2018/term-project-team-e-for-effort

Project Introduction

For this assignment we were tasked with creating a space mail delivery game, using someone of the objects from the tank game we previously created. This game is also a 2D game where the player controls the movement of a spaceship with the arrow and spacebar keys. The ship is to have limited controls and can only use thrust to propel off of landed moons, while flying the player will avoid dangerous asteroids. Once the player had landed on a moon for delivery they gains points, but the longer the player hangs out on the moon the points will deduce. Once all deliveries are made the player moves onto the next level.

Execution and Development Environment

Written with NetBeans 8.2 using Java 8 / JDK 1.8 on macOS 10.13.4.

Scope Of Work

For the implementation of this game there were some classes already made in the previous tank game that were pulled for reuse such as the game clock, the basic object class, and the sprite sheet loader for the animations. Using the basic object as a base class for more complex objects we were able to create the player ship object in which the player would be able to control using the key listeners. The basic object was also extended for use with the moon and asteroids objects that were implemented into the game world and drawn to a buffered image to be displayed. The object position were updated and redrawn to the JPanel on every tick of the game clock in order to give the user a feel of continuous animation, which we calculated to run at 60 frames per second. We also implemented a game menu used in order for the user to select the state of the game whether that be running the actual game or viewing the list of high scores that we saved to a text file. For this particular game we also implemented a level system, in which after the user has made collisions (landed) on each moon the game world is re-initialized with an increase of asteroids to make the level more difficult than the previous one.

Assumptions

There were many assumptions made as far as the functionality of the game, as far as how the player ship was expected to move, given that the instructions expressed “limited” movement. As well as there were no specifications on the map layout or whether the layout would change on level change. Frame rates and overall object movements were left to assumption as well.

Command Line Instructions

Navigate to proper directory:

navigate to .../term-project-team-e-for-effort/Submission/Mail

To Compile:

javac galacticmail/GalacticMail.java

To Execute:

java galacticmail/GalacticMail

Implementation

Package Organization:

- GalacticMail
  - GalacticMail (Main)
- Animations
  - Explosion
  - Sprite Loader
- Collision
  - Collision
  - Game Ender
- Controllers
  - GameClock
  - InputController
  - KeyObject
- Map
  - Game Frame
  - Game Menu
  - Game Scores
  - Game World
- Objects
  - Basic
  - Asteroid
  - Moon
  - Player
  - Score
- Resources
- Sounds
  - Game Sound

Order of Operations:

Similar to the tank game, this game starts off in the Galactic Mail main class where the Game World and the Game Frame are initialized. The Game World is initialized into a menu state. In this state, a game menu object is created, an input controller passed into it,

and it is then drawn on the screen. On this menu, the player can choose to play the game, see high scores or quit.

If the player chooses to play the game, the menu will prompt the player for his or her name in order to add their name onto the leaderboard. After the player enters their name, the Game World is set to the playing state where the game will commence.

Our implementation of this game was able to bring over a couple key elements from our original tank game such as the sprite loader, input controller, game frame, basic object, and the game clock. Our original implementation of the tank game was mostly designed for that specific game type and wasn't totally compatible with the new assets and play style of the new game. What we were able to take from the original game were many of the implementation decisions that we made and the methods we used in order to create certain aspects of the game. For example, our game menu was easily adaptable to the second game, just with some light modification for size and selection requirements.

Much of the code for controlling player position, movement, rendering, input, and object updating was kept exactly the same as it was for the tank game. Some parts of our code did have to be changed to adjust for changes to differences with the resources required for the Galactic Mail game. Other than this, many aspects of our code was simply copied over to be used in the Galactic Mail game.

For this game, we had to implement a high score system that is to be viewed at the end of a game or by selection from the player. This was done using a Game Score class that would read from and add to a text file used to store all the player's names and the score that they received. When the Game World was set into the Scores state, the Game World would then create and start rendering the Game Score class onto the screen. Upon initialization, the Game Score class will read all the scores from the text file into an arraylist of custom score objects. From there the scores will be arranged in descending order using a modified version of the Collections.sort library. After sorting the six highest scores are drawn onto the screen. On the high score screen, the user can choose to go back to the main menu or quit.

Results / Conclusion

Overall the implementation of Galactic Mail turned out well, and it shows the importance of using good object-oriented implementations. So when code changes need to be made in certain objects the whole code-base does not fail. Also using well well defined objects

that have clearly defined functions make them easier for reuse which can make future projects much simpler.

It was possible for us to create an absolutely abstract version of the game engine and its subsequent components, however we did feel that this sort of abstraction could lead to issues with performance or problems with implementation of more specific features to the Galactic Mail game. In the end, we took only the most basic components from our games to use in both. Certain aspects of the game require specific functionality that couldn't easily be created using a highly abstract method.

This game does still have a couple bugs that we did not have time to work out. Occasionally the player ship will land in a position that is off of the selected moon but close to it. Launching still has some issues occasionally that we were not able to work out in time for submission.

Other than these issues, the game runs well and is easily playable over and over for the player to get the highest score possible.

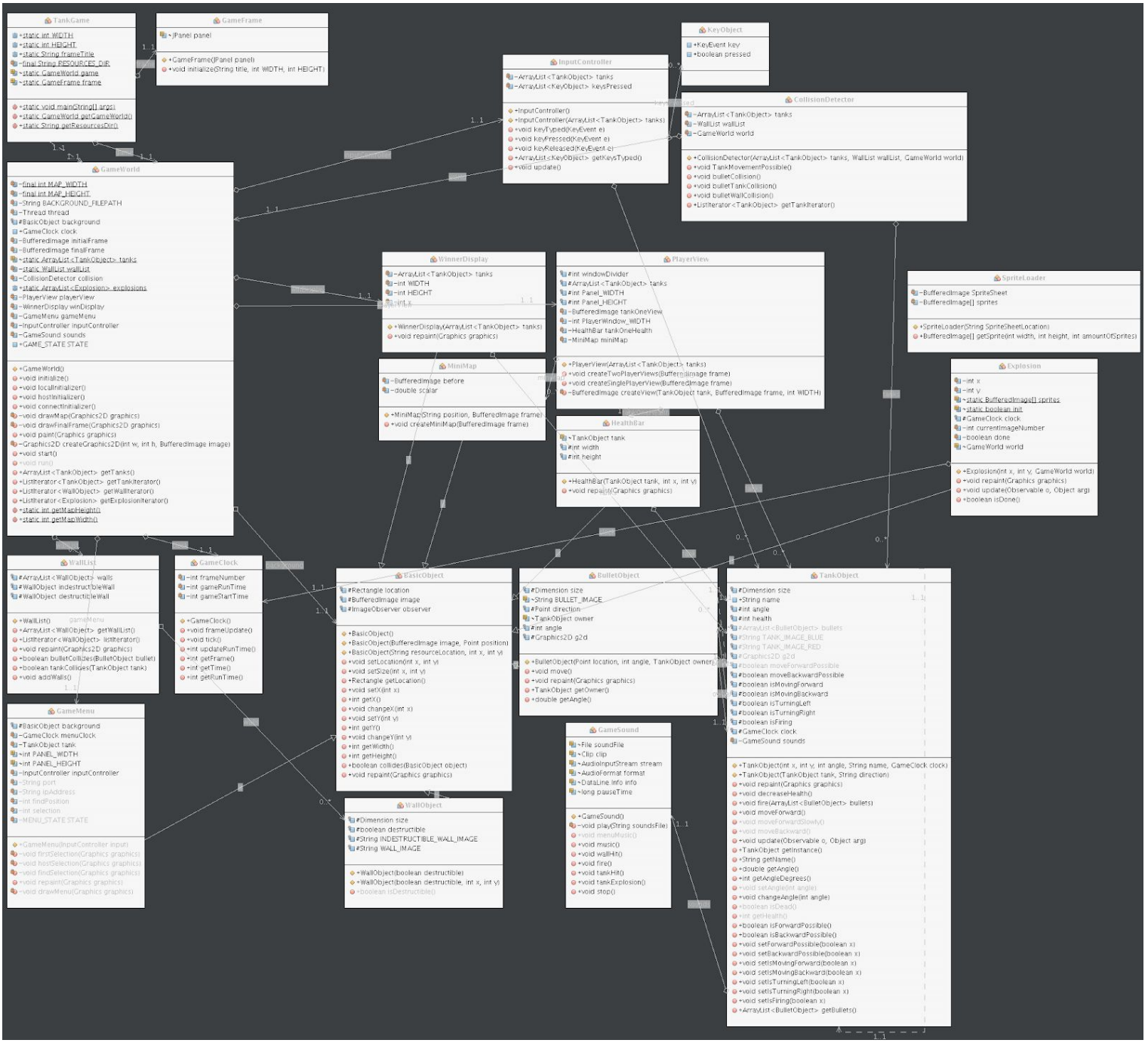
Tank Game & Lazarus Comparison

In the end, the tank game and galactic mail were two separate games with different features and goals, however their underlying game engine was very similar even if it was not absolutely the same code. The methods we used to create the first game translated over to the second game which made development of the second game very quick.

Our game engine was nearly identical across both game. The differences were in the options that the game could run in as well as with the objects that are used. The engine has different methods of initialization for the objects that are spawned on screen and thus the engine had to be modified in order to have it properly display the objects in their corresponding locations upon initialization. Game States were another differing factor in our game engine implementations. In the tank game, the Game World game engine has networking states that it can be set to and (although it wasn't implemented fully) it added to the complexity of the Game World class for the tank game. The Game World class in Galactic Mail was much more simplistic by comparison. It only had to render in one player object and some obstacles for the player to run into.

Due to each game having its own set of rules and features, we were not able to completely transfer over the code that we created for the first game. We did, however take the lessons that we learned on how to create certain aspects of the game and transfer over those methods to the second game which made development of the second game much faster and streamlined.

Tank Game Class Diagram



Galactic Mail Class Diagram