CSC 413 Term Project Documentation

Summer 2020

Jarett Koelmel

920135967

413.01

https://github.com/csc413-01-SU2020/csc413-tankgame-jkoelmel

Table of Contents

1 Introduction

1.1 Project Overview

This application was part of a term project that focused on the practical side of object-oriented programming (OOP) and using these skills to implement a basic video game. At its heart, the key was encapsulating basic types of resources that would be reused in multiple places and in various ways so that amending or adding new types of objects to the project would become a painless evolution.

While the scope of the game was given basic outlines to follow regarding implementing a two player 2-D tank game, we were given free-reign to pursue other types of games and projects. As such, I have implemented a procedurally-generated maze game which pits two players against each other as well as an AI player.

1.2 Introduction of Game Concept

My game, Labyrinth Game, has two players with the ability to shoot fireballs that are stuck in a maze filled with gold coins, the goal of the game is to collect more coins than the other player while dodging the ghost AI that follows the closest player while coins are still on the map.

There are randomized power-ups placed throughout the maze at each start and each playthrough is different as the maze is generated at random over a grid of 25 x 14 or 350 rooms. The AI speeds up with each coin picked up until making contact with a player, at which point, it will respawn somewhere in the maze at random and its speed set to the default speed, only to build again with consequent coins picked up.

Contact with the ghost or another player's fireball, reduces player health. There is only one life, so when the health bar is drained, that person dies. There are, however, health pickups on the map in random and limited number, to help remedy this situation.

2 Development Environment

a. For development, I used Java version 14.0.1, but any Java version > 8 should be compatible with the LibGDX library used to create this game.
b. IDE used was IntelliJ Ultimate 2020.2
c. My library dependencies come from the LibGDX family of game development libraries. For game assets: textures, sprites, backgrounds, and audio, I either made them by hand, used open source content from opengameart.org, or used that same open source content and tailored it to fit my needs.

3 How to Build/Import your Project

a. Clone the repo to your computer. Open your IDE, for this example, I will be using IntelliJ to explain:
    1. Click Open…

2. Select either the entire repo directory folder, or the project's build.gradle file that is one sub-directory down. (Not in the core or desktop folders).
3. If building from Gradle file, you may need to install plugins to help manage the filetype, IntelliJ's can be found under settings -> plugins and searching for Gradle.
4. If opening the directory as a whole, navigate to the build.gradle file discussed above and open in the IDE. If you have the correct plugin, Gradle will either begin installing dependencies or prompt you with a small Gradle icon (elephant) in the corner of the editor.
5. If wanting to test directly from the IDE via the DesktopLauncher class, make sure that the working directory is set to the repo directory, or you may receive errors along the lines of asset or file not found.
6. Once all files are indexed, dependencies installed via Gradle, and settings checked we can package the JAR via the terminal, either Windows CLI or IntelliJ will work.

b. Because this project uses Gradle to manage dependencies and packaging, we will be using the command line to create the fat JAR that holds all of the dependencies and assets required to run the game as a standalone file. From the terminal in the correct directory enter:
   1. gradlew desktop:build
   2. gradlew desktop:dist

   You should see build folders populate the IDE Project explorer under core and desktop directories. The JAR files, core-1.0 and desktop-1.0 respectively, are held in those folders under the libs folder. The JAR is now packaged and can be moved to different folders and be renamed without issue. The included JAR in the jar directory was generated in this way.
c. Navigate to directory that holds the JAR and run "java -jar {filename.jar}. The default attached to the repo is named Labyrinth-Game-1.0.jar, but as discussed earlier, the distribution default by LibGDX from the Gradle settings is desktop-1.0.jar and that also should be able to run independently if desired.

4 How to Run your Project

The game can be ran via the command line as explained in Section 3.c or found in the file explorer on your system and ran via the JAR icon like any other application.

The rules of the game are simply to survive the maze and clear the map of gold coins. There are two players and an AI controlled enemy in the maze. Both players have the ability to shoot fireballs at each other, the walls, and the ghost. Hitting the walls will destroy half of the wall, meaning escaping dead ends requires two hits. Hitting other players reduces their health and stuns them temporarily, not allowing them to move or shoot. Hitting the ghost, stuns it briefly before it resumes tracking the players again.

The controls for the game are listed in the instructions screen and is accessible from the main menu. Controls:

Player one is controlled by the arrow keys and shoots with the right-CTRL key.

Player two is controlled by WASD and shoots with the SPACE key. During the game, you can pause with P, return to the main menu with ESC, restart with a fresh map with R and mute the music with M. The ghost is controlled by the computer in full and does not allow for player control.

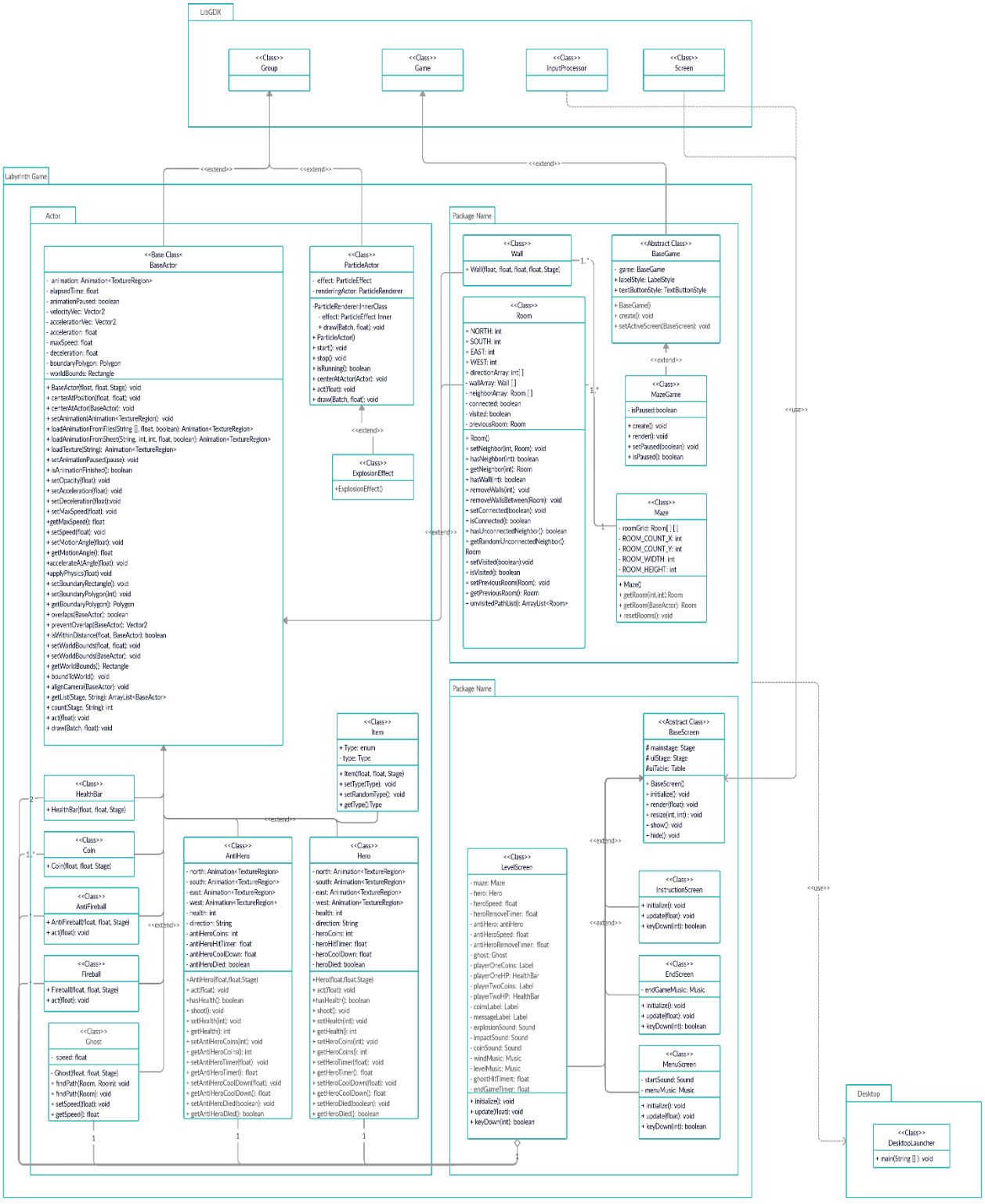
If both players die, the game is over. If one player manages to finish, the person with the most coins, alive or not, is declared the winner. Because the game consists of 350 rooms, each with a coin, there is a chance of a tie. From the game over screen, you can choose to play again with the ENTER key or quit the game with ESC. The menus allow for mouse input as well for clicking the buttons to navigate through the menus.

5 Assumptions Made and Implementation Discussion

During the design and development stage of my game, I assumed the crux of approaching the program successfully would be proper implementation of the rules of object-oriented programming, inheritance, and polymorphism. Templating as much as possible so that large sections of code are not repeated unnecessarily and class variables are not accessible to an unreasonable degree.

The entirety of my implementation of this game relied on building base classes upon which I could build on top of and customize as needed generate all objects, actors, and screens required to create a fully functioning game. Using the LibGDX library, some abstract classes and interfaces provided a barebones framework to draw upon. Extending those classes, I implemented a plethora of custom methods to handle object creation, editing, user input events, and collision handling. These methods will be discussed in greater detail in Section 7.

6 Labyrinth Game Class Diagram



7 Detailed Class Descriptions and Explanations

Actor Package -

BaseActor Class:

The foundation of all actors in the game are built upon the BaseActor class which extends the LibGDX Group class. This class is one of the largest sections of code in the game but allowed for some of the classes built upon it to be rather simple. The fields of the class contain basic information that is universal to all types of actors I would be implementing in the game.

The constructor of BaseActor, calls the Group constructor and then proceeds to initialize all fields to generic values which will set the baseline for all instances of this class. As BaseActor is not abstract, it can and is instantiated in cases where additional or specific functionality does not need to be implemented.

BaseActor contains a large array of public methods used as accessors and mutators. Many of them are self-explanatory in nature. But I shall explain the more complex ones here:

setAnimation: This takes an Animation object as a parameter and passes it to the calling actor. Its size is set based on the values of the animation passed in, and is centered on the actor accordingly. If there is not already a boundary established for collision, one is created for it by calling setBoundaryRectangle().

loadAnimationFromFiles: Rather than using an Animation object, a String array of filenames can be used to generate an Animation object which is the return type of this method. The frame duration is how long each frame is displayed while the animation is running in seconds. The boolean loop is straightforward and sets whether the animation plays once for loops while active.

loadAnimationFromSheet: This is similar to the last method, but instead of multiple files, it relies on a standard sprite sheet that contains all the relevant positions of the animation. The rows and columns are used to divide the input file into a grid that will be used for individual frames of the animation. Nested for-loops populate a Texture array that is then used to create the Animation object.

loadTexture: This is a simple method that co-opts the loadAnimationFromFiles method to set static textures but as Animation objects allows for translation and adjusts to be made easily.

After the animation and texture methods section, there are a handful of accessors and mutators that handle the physics of the actors. The applyPhysics method handles the regular updates at a specific delta time, dt, which is the time between rendering frames.

Next is the collision methods, these require establishing the boundary geometry for actors in the game.

setBoundaryRectangle: This is a simple method that draws a rectangle around the actor in the size of its width and height. It provides a short-circuit evaluation method when checking for overlap between actors in-game.

setBoundaryPolygon: This method creates a more accurate boundary around the actor especially when the actor's texture does not display to the edges of the actual texture. The polygon can be created using any number of sides desired but for simplicity sake, I have used 8 in my implementation to create an octagon around the actor for collision detection.

overlaps: This method checks whether the polygons of the calling actor and the actor passed in as the argument overlap. There is a short-circuit evaluation condition in the method that checks the rectangles of the actors first, in the event there is not a boundary rectangle for one or the other, the polygons are checked and a boolean is returned.

preventOverlap: This is generally called after an if-statement returns true that there is overlap between actors. It relies on a MinimumTranslationVector object that is a 2D vector which can offset the actors by the minimum amount required to prevent clipping during collision events.

boundToWorld: This allows for actors that call it during their update stage of running to not escape the bounds of the world set by the mutator: setWorldBounds. This is important because the exterior walls of the maze are destructible but the player still cannot leave the map.

getList: This method takes the name of a Stage object, and a className of Actor to find and returns an ArrayList of all current actors in play on that stage. This is used many times for checking updates on item and coin count as well as handling collision events and consequential actions to take upon those instances.

The two final overridden methods are act and draw and required to instantiate the actors in the game world. They are overridden from the Group class but have multiple cases of being overridden by the classes that extend this class as well.

AntiFireball Class:

This is the actor for the AntiHero's weapon, the blue fireball, which calls the constructor from the BaseActor class and sets the default speed of the projectile. Because of the instantiation of non-default physics values, the act method is overridden and applyPhysics is called in its own implementation.

AntiHero Class:

The AntiHero class holds all the fields and methods for player two in the game. It has a different texture and animation applied to it to help differentiate it during play. There are fields to track all pertinent information about this player during the game:

health – tracks the current player's health and get represented by another actor for the UI called HealthBar

direction – this is used to track which animation should be called depending on which way the player is facing and/or moving

antiHerocoins – a running tally of all coins collected by this player

antiHeroHitTimer – this is a float value that gets set when the player is hit by the other player's fireball and prevents them from moving or firing until it is back to 0

antiHeroCoolDown – this is used to prevent spamming of the player's weapon. After shooting this timer is set to 0 and it must reach 2 seconds before allowing the shoot command to be used again.

antiHeroDied – a simple boolean value to denote whether or not the player is still alive. Mainly used for end game detection.

After initialization of default values and texture application for the four main walking directions, there is an implementation of act which constantly detects the status of the actor and adjusts the animation accordingly.

The shoot method handles the instantiation of a new AntiFireball object, sets the proper rotation of the texture and animation and then sets the direction of motion to be the same as the current actor's motion or facing direction.

The remainder of the methods in this class are simply accessors and mutators to handle changes to the actor outside this class.

Coin class:

This is a simple class that only has a constructor which calls the BaseActor constructor, sets the texture for the coin, and creates a collision polygon. The population of coins on the map is handled in another class which will be explained later.

ExplosionEffect Class:

This class is an extension of the ParticleActor class I created to handle the explosions that occur when two fireballs hit each other or the fireball hits a wall and destroys part of it. The particle effect used was made in the LibGDX particle editor application and imported into my assets.

Fireball Class: Much like the AntiFireball class covered above, all of the fields and methods are the same with the primary difference being the texture and animation applied. It was required to create another class for this in LibGDX because of how the collision would be handled via the getList BaseActor method. Having two instances of the same class type would have created many issues in correctly identifying which player's weapon was interacting with what on the screen simultaneously. So, while it may seem like repeated code, the alternative would have been more unwieldly to code and much less elegant of a solution.

Ghost Class:

This class holds the logic for the enemy on the map. The ghost is initialized with a default speed and animation. The bulk of the code is dedicated to the two findPath methods. It is overloaded because of the need to continue pathfinding when one of the two players have died, something that is detected via the boolean values in the AntiHero and Hero classes. The findPath methods use breadth first search to find the shortest path to the closest player when called. Then the action queue for the Ghost actor is populated a few steps at a time so that the path is constantly updated for player movement and destruction of the map.

HealthBar Class:

This class provides a simple object and texture that can be scaled according to the int value of the respective player's health.

Hero Class:

This is an almost identical class to the AntiHero class discussed above and just like the issue with the fireballs, having a discrete class type for the other player made the logic of collisions and actions based on which player was doing what in game became much simpler to implement when simply extending a new class which holds a different texture and a separate set of differently named fields.

Item Class:

This class holds an enumeration of the two types of power-ups to add to the map. In the constructor for the object, after the BaseActor constructor is called, a helper method is called which sets the type of the new object randomly to one of the enumerated types. Then collision boundaries are set and the object is ready to be added to the map, a function done in another class yet to be discussed.

ParticleActor Class:

This class holds an inner class which is responsible for setting up the particle effect to be animated on the screen. The particle itself is really just a small white block that is used as a spawning point for the effect made in the particle editor. The act method handles updating the particle effect and disposing of it properly when completed. The draw method simply calls the superclass's constructor but is necessary to function properly.

Game Package-

BaseGame Class:

As an abstract class, the BaseGame class outlines the styling of fonts, labels, and buttons that will be used in the actual game. For this game, there is only one extension of this class which is by the MazeGame class.

Maze Class:

The Maze class handled the construction of the procedurally generated maze. It tracks the total time it takes to generate a new maze which is printed to the console when ran from within an IDE. The roomGrid is comprised of a 2D array of Room objects. To fill as much of the screen as possible and leaving room for the UI at the top of the viewport, the grid is 25 Rooms wide and 14 Rooms tall. The Rooms are first created, then relationships are established between each of the rooms in the four cardinal directions. Then a while loop handles ensuring each Room in the maze is connected to another in at least one direction. The branch probability setting is set to a median float value of 0.5 but can range between 0 and 1. Adjust this value has a direct effect on the type of maze generated. Lower probability of branching means that there are longer passages with fewer off shoots, higher probability means that there will be more short passages, more often. The next variable that affects maze generation is the wallToRemove setting. This dictates how many more Wall objects to remove from the maze. A higher number in this setting can make for rather open maps, a lower number makes a more traditional maze with little to no

clearings. After testing, a number of 50 seemed to be a decent balance between rigid maze and open passages. The final methods are overloaded accessors and a reset function.

MazeGame Class

The MazeGame class is simple extension to the BaseGame class which relies on the super-class's methods heavily but also relies on the screens implemented in the Screen package. The only field is a boolean value that allows for the render engine to be paused by the user when the Pause button is pressed. The easiest way to check various screen implementation for debugging is to change the value of the setActiveScreen method call in this class as it is the entry point for the game. This way, checking the game over screen does not require playing the entire game every time you make a change.

Room Class:

Even though the Room and Wall classes are technically extensions of the BaseActor class and uses those methods for collision and proper object creation and disposal, the logical place for these classes was in the Game Package because of their close relation to the Maze class and generation. Each Room is created with a Wall and Neighbor array and the connected and visited boolean values to false. The rest of the methods in this class are simply accessors, mutators, and the public methods for the logic that is required by the Maze class's procedural generation and Ghost path-finding.

Wall Class:

The Wall class is rather simple, it just creates an object that is a thin line with a texture applied and some basic collision boundaries. They are then populated by the Room Class in the Wall array that surround each grid box. The Maze class handles removing the proper walls and some extra to create an actual maze.

Screen Package-

BaseScreen Class:

The BaseScreen class is an abstract class that outlines the pertinent parts needed to create the various screens of the game. There is a mainstage that handles the actors that are directly part of the game or menus. The uiStage handles the user interface in-game. The uiTable helps in formatting the uiStage via a standard set of methods provided by the Table class from LibGDX. The uiTable is directly inserted into the uiStage during construction of

the screen. Because this class implements both the Screen and InputProcessor classes from LibGDX, there are a handful of methods that must be present but not necessarily implemented. When necessary, screens that extend this class will utilize them. Render, show and hide methods are required at a minimum to create a functional screen that can be extended by concrete classes.

EndScreen Class:

This is the class that holds the game over screen and allows for the players to restart a new game or quit the application entirely. There are event listeners tied to the buttons that allow for the usage of the mouse of the keys listed on the buttons to activate the respective functions.

InstructionScreen Class:

After debating on whether to include a simple snapshot of the instructions or create them in-game, I decided to co-opt the uiTable that is inherent in all screens to embed icons and instructions for each part of the game and a button that takes the user back to the main menu via either a click or through pressing the escape key. Simple versions of the textures from the game were edited to make icons that show the user what each item looks like, what is does, and how they are controlled, if at all.

LevelScreen Class:

The LevelScreen class holds all of the initialization and logic for the game during runtime. There is an instance of each object that will be required to spawn as private class fields. After initializing all parts of the level and starting the game music, the massive update method is constantly called at each time delta to run checks for collisions, item collection, health changes from attacks or hitting the ghost.

The fireballs from each player will destroy a wall segment upon impact, hitting the other player reduces their health and stuns them, hitting the ghost stuns it temporarily so it can't move. If fireballs are shot at each other they will explode and negate each other.

While the game is not in an end-game state, there is wind noise attached to the ghost that increases in volume depending on how close it is to a player.

There are four resolutions to each game: player one wins, player two wins, a tie, or game over from both players dying. These can be checked quicker through the debug code placed in the initialization that populates only 6 coins instead of all 350. Commenting out the previous two lines and uncommenting the debug statement will prevent having to play a whole game three times to check that each coin-based win condition works.

MenuScreen Class:

The MenuScreen is the entry point into the game when loaded. It has a background, some basic buttons and music. The buttons work via key presses, as listed on the buttons, or through clicking due to event listeners attached to them.

Desktop Package-

DesktopLauncher Class:

This is a required class to be used to spawn an application for LibGDX on the desktop. It establishes the configuration of the application, takes a new instance of the MazeGame and runs the game from this class. This is the entry point when running from within the IDE and not utilizing the stand-alone fat JAR in the jar folder.

8 Project Reflection

This project was an excellent exposure to the importance of polymorphism and inheritance in a large-scale project. The ability to create base classes and simply bolt on new classes by extending them and expand their functionality when needed made this project's development much easier than attempting to read my code in various places unnecessarily. As much as I wanted to simply dive in and start coding the game, I had to think of what I wanted to create because I was set on not making just another tank game, if possible. This resulted in a lot of brainstorming and thinking about what would be challenging but plausible to make in the time allotted.

So, while I ended up deviating from the standard game design, the tank game, and chose to use a library that was not covered in the lectures. I found that really learning the ins and outs of a new library is not as daunting as it seems sometimes. Once I began to understand how the game process flowed and everything was related to one another, the code came together quickly. The most tedious part of the process was the constant debugging and testing. During which, I would think of new ideas or concepts I wanted to implement and I would set out to find a way to handle it without breaking everything else already in place.

When all was said and done, I felt a great sense of accomplishment from what I had created and all the work I put into tweaking it and making it somewhat enjoyable to play given my limited experience and skill with this library and game development as a whole.

9 Project Conclusion/Results

My game probably went through 50-60 revisions since it became a running prototype last month. It was an arduous process of trying to break the game and checking that each part

worked after making a large revision. As it stands upon final submission, the game worked well as a stand-alone from the attached JAR in my repo and I hope you enjoy testing it and playing because a lot of work went into making something that I could feel proud of submitting.