COMPSYS302

Nilesh Magan – Jilada Eccleston

Introduction

The client has provide us with a brief indicating a need for a game to be created for his son. The game has been designed to provide a simple local multiplayer gaming experience for the client's twelve-year-old son. The game would need to be visually appealing and buttons would need to be coordinated so it is comfortable to play for a moderate amount of time. The game would also need to be varying, supplying different game modes and challenges for the twelve-year-old.

The style of the game will be similar to Warlords, a game originally manufactured and sold by Atari in 1980. Our game will consist of a minimum of one user controlled player and three AI controlled players to a maximum of four user controlled players. The users, and AI, will each control a paddle that used to deflect a ball away from their base; a corner protected by a wall. If the ball touches any surface it will deflect back into the game. When the ball touches a brick, the brick will be removed from the game. When the ball touches the player or the player has no walls left, the player 'dies' and is removed from general gameplay. The objective of the game is be the last remaining player

System Requirements and Overview

Welcome Screen and Game Mode Selection

The flow of the game involves a total of five different screens. The user is greeted with "Welcome" screen which gives the user three options, "Play", "Controls" or "About". When the user selects the "Play" button, the user is prompted to select a difficulty level. Once selected the user is taken to the "Select" screen which prompts the user to select the number of players. The user then presses play and is lead to the main game screen. From here, the user is able to press Esc to leave the game and return to the "Welcome" screen. Otherwise when the game timer runs out or a winner arises, the game transitions to the "End" scene. This "End" scene returns the player back to the "Welcome" screen so the user can choose to play again.

From the "Welcome" screen, if the user selects "Controls" the user is taken to a screen which explains the way the game is played and the controls used. If the user selects "About", they are taken to a screen that display developer and copyright information about the game.

Game Initialisation and Countdown Timer

Once the user selects the game mode, the game is initialised with the 4 player characters. Boundaries of the game window were specified by the client, this was set to be 1024x768. The walls are then built using brick images that are randomly selected when building. The paddles are also initialised at a horizontal position. A ball is spawned in the middle of the screen and will begin to move in a random direction once the countdown timer reaches zero. Starting at three, the countdown timer begins to countdown in second intervals determined by the *nanotime()* function. Before the countdown timer reaches zero, the paddles and ball remain unmoved and any key pressed will be ignored.

Collision Detection

A large part of the program is dedicated to collision detection. In order to keep the collision detection generalised, our program extends all detectable objects from *modelSuperClass*, a class which will allow for objects to be stored in an accessible location and easily checked when a collision occurs. *Paddle*, *Ball*, *Brick* and *Player* are object classes that extend from this superclass.

Collision detection also requires the knowledge of the object's shape and its properties. Another class called *gameObject* was created in order to store information on both the model and the view of the

object. During instantiation, the shape and the model of the object are added to a new *gameObject* class and then added to an *ArrayList* of *gameObjects.*

Within each tick of the animation timer, collision checks occur. For each object in the *ArrayList*, the *checkCollision* function will be called. The function uses basic maths to determine the points of the shape that is being checked and then from there determines if the line between two points on the shape intersects with the path of the ball. If the path intersect multiple lines then the closest line to the ball's current position will be taken. Once the collision point has been found, a *CollisionStruct* class will be instantiated and the new points of the ball are stored within it. It is then returned and used in the *moveBall()* function where it will be used to determine the new position of the ball.

Keyboard Inputs

All keyboard inputs are computed in the *IOHandle* class. In this class, key event handlers are listening to key pressed. For multiplayer functionality, the keys pressed are added into an ArrayList of strings and then the ArrayList is evaluated in a single loop during the tick. Then a key has been pressed it flags a boolean which will then be checked when handling the paddle movement. Currently, the keys are wired as such:

- Paddle one movement - "A" to move left, "S" to move right
- Paddle two movement - "F" to move left, "G" to move right
- Paddle three movement - "J" to move left, "K" to move right
- Paddle four movement - left arrow to move left, right arrow to move right
- Exit game - "Esc" to exit the game
- Pause game - "P" to pause game
- End Timer - "Page Down" to time out the game

Game Sounds

Game sounds are played at appropriate times when the ball collides with a paddle, wall or player. These sounds are started and stopped within a new thread of the program, this allows for the sound to be played at the same time as actions that would occur following the collisions. In game music is also played in the background in all the menus and in the main game as well.

Top-Level View

The user interacts with the system using key pressed and mouse click actions. The first screen displayed is the main menu screen which allows the user to decide what they want to do with the program; such as view options, get help or play the game in one of its game modes. These actions can be selected using both keyboard and mouse inputs. A diagram is attached in the appendices that gives a visualisation of the game flow.

Development Issues

In the initial stages of development, we found it difficult to configure the Git repository. Part of using git is dealing with the conflicts that may occur when merges between the codes occurs. Before we realised how to deal with the conflicts manually, we were searching for tools that would help us visualise the issues, however none of these methods really worked. The issues were then resolved by manually investigating the conflicts flagged by Git and manually removing them.

Following on from the issues with Git, we also had a couple of issues involving JavaFX 8 and running on Eclipse IDE. To simplify the process, we both decided to switch to developing on the IntelliJ IDE. This proved an easy move as the libraries required and the integration of the jUnit testing system were packaged with the IDE.

The design of the collision detection proved to be a challenging one. Our method involved drawing lines around the perimeter of the object in the hope of it being more accurate than other methods; such as intersects.

When first implementing variables in JavaFX, we needed to make sure they were compatible, to do this some of the variables needed to be declared as final. Before we noticed this solution we had a lot of trouble getting our GUI to function.

Functionality Improvement Features

In our main menu, there is an option to navigate the user to the controls menu. This menu displays the controls used in the game and allows the user to interact with the objects before beginning the game.

We decided that since the game can be fast-paced, we would allow the user to still maintain the use of their paddle once they have 'died'. The player of the 'dead' character may still deflect the ball using the paddle and ultimately help or destroy another character. This enhances the multiplayer experience between players.

Design Discussion

Suitability of Application Tools

The library used to develop all the code was JavaFX. Although Java Swing has methods and an integrated GUI building interface that makes it easier to produce GUIs, Java Swing is also an outdated library in which Oracle is no longer enhancing. To maintain reliability and currency in our design we have chosen the more modern and powerful library; JavaFX.

To begin with, both developers were using the Eclipse IDE to develop the game. However, due to issues with getting JavaFX 8 to work on the IDEs, IntelliJ IDE was adopted. As IntelliJ had pre-installed JavaFX libraries, it was not necessary to go through the installation process. JUnit was used to test the software without any visual representation.

To maintain correct version control of the code, a Git repository was created and used extensively to monitor, test and control the code. In the end it proved to be practical tool allowing us to quickly revert back to previous version for checking as having branches containing ideas that may or may not have worked.

Object-Oriented Design - Cohesion and Coupling

Our software has been broken up into three packages; *control_classes*, *model_classes* and *view_classes*. These packages contain classes that are associated to one of three class types in the MVC development schedule as indicated in the package name.

Included in the *view_classes* package are the *Boundaries* and *RenderView* classes. The *RenderView* class contains information on the setup of each scene. This class instantiates all the shape objects for

the paddle, player and ball. It also determines where these shapes lie within the boundaries of the window and continues to relocate the shape as the model is changed.

The *model_classes* contains the model classes for all the objects included in the game. The *Ball*, *Brick*, *Paddle* and *Player* classes extend from the *modelSuperClass*. They hold information regarding instances of the objects in the screen. The *Wall* class stores an *ArrayList* of instances of the *Brick* class. *Player* classes are designed to keep information from each player separated. When instantiated, a *Wall* instance and *Paddle* instance associated to the particular *Player* instance is added to the *Player*. All locational information of the model objects are stored in a *Point* class. In order for the game to be able to run collision checks on the different objects, the *gameObject* class is used to generalise all the object types. This particular class is used by the control classes and stored a reference to the model and the associated view (shape) of the model. *CollisionStruct* is a class that is instantiated when a collision has occurred instead of performing the same calculated again when moving the ball. It contain positional information to be used in the control of the objects when the collision was calculated.

As part of the MVC modelling technique, control classes are required to converse between the model and view classes. The *control_classes* package contains a set of classes which implement logic and determine the state of the game. *GameSetUp* is a class is that instantiated all the model objects, that is the paddle, player, ball, bricks and walls. The *IOHandle* class sets up the implementation for when keys are pressed. These are dependent on a number of attributes such as the number of players, which introduces AI players and less key press actions. When the key pressed handled, booleans are flagged which determines which keys have been pressed in on tick. These flags are then used by methods within the *ObjectControl* class are used to move the paddle and ball. The *ObjectControl* class also contains methods that will take appropriate action when the player dies, including removing all remaining bricks and the paddle. Collisions in the game are detected by using the *Collision* class which *gameObject* model of the objects in the game.

The *AI* class is considered part of the *control_classes* package as instead of handling the user inputs for the paddles it will call the *AI* class to set the flags in the *IOHandle* class. The *AI* class contains two methods, one of an easy *AI* which does not track the ball's position and an advanced *AI* which does. Depending on the game mode selected, the main game loop will call a different method. Sound played within the game are called through the *GameSound* class, this class uses multithreading to play the sounds concurrently to the other methods occurring in the main game.

To set and render the scenes we use the *SceneChanger* class. This class renders the scene of all the different scenes used in the game. It also contains button click input handlers to change the scenes. This particular class contains the method for the main game animation. This class was the main controller of the game flow. All of the classes in the different packages came together in the *MainGame* class, which instantiates the entire game. It calls the *SceneChanger* class to create the scenes and add it to the game.

We have applied methods in programming that increase cohesion and decrease coupling between classes, particularly those in different packages. Certain classes within the control classes speak to classes in the model and view more than others. Although some of the rendering occurs in the control classes, we reason this as controlling the way the image looks. Therefore our view class package is rather small. In our model classes, there is a class that stores information on both the model and the

view of an object, *gameObject*. This class is simply a means to connecting a shape representation to the model properties of an object. The two do not communicate.

Software Development Methodology

To develop our game we used a mixture of the more popular software development methodologies. Of which include the “Waterfall” and “Feature driven development” methodologies. We started off by carefully analysing the client's requirements and from that we created a priority list of what should be implemented first. We created diagrams of what our UI should look like as well as diagrams that describe our classes. It was only then that we actually started implementation. We would code different parts of the project and would have scrum meetings to discuss what we have been doing since the last visit. We used Trello to keep track of our progress and goals. It was this half of the project that was based on the waterfall methodology. We would make some code, test it and then move on. It was only later on that we diverged to feature driven development. Here we just implemented what was required by the customer to make sure that they were satisfied. This became an iterative process where we made code, tested it and then rewrite code again to make the specific features functional.

Improvements and Future Design

Our game currently supports up to four players making the remaining players AI. In the future we look to add more game modes that will make the experience more enjoyable. Game modes will come in the form of a team-based mode and a score-based mode. In team-based modes, players will be assigned to a team and will have to work together to beat their opponents. Score-based modes will work by allowing players to win by getting the highest score in the game. Scores will be increased by getting powerups and by destroying other players bricks. Scores would also decrease if you were hit by an enemy power up. In terms of the game experience, we want to add maps so that players can be refreshed while still enjoying the same game mechanics.

We also want to improve the number of power ups and game abilities. Players should also be able to hold onto their powerups and then deploy them when they are ready. This mechanic would add much more dynamics to the game as players will be more immersed in the experience as they now have more of an incentive to pick up powerups, so they can tactically deploy it at certain players.

An aspect of the game that we can improve on for the future technically is our collision detection algorithm. Our current algorithm is effective however sometimes collides with objects and doesn’t reflect the ball. The mechanics of the algorithm was well designed as it currently detects the path that the ball is going to take using mathematics. This is useful as in the future, we can look to improve our AI by using this similar approach making it very difficult to beat.

Copyright Information

Sound sourced:

- Background Menu Music - Sourced from: https://opengameart.org/content/battle-theme-a
  Licensed under the Creative Commons 0 License
- Brick breaking sound effect - Sourced from: Drum Pads 24 Android App - Paul Lipnyagov
  Licensing: Eligible for use as long as there is acknowledgement
- Game music - Sourced from: https://opengameart.org/content/tragic-ambient-main-menu
  Licensed under the Creative Commons 0 License
- Paddle collision sound effect- Sourced from: Drum Pads 24 Android App - Paul Lipnyagov
  Licensing: Eligible for use as long as there is acknowledgement
- Player death sound effect - Sourced from:
  http://www.freesound.org/people/rene___/sounds/56778/
  Licensed under the Creative Commons 0 License

Images sourced:

- Game background - Sourced from:
  http://forum.halomaps.org/index.cfm?page=topic&topicID=30343&start=6826

Appendices – Class Diagram