Sylvain Bechet, Mark Yep

Warlords

Final Submission Technical Report

compsys 302   
java game project

Group 01

Table of Contents

[System Requirements 1](#_Toc479852962)

[System Outline 1](#_Toc479852963)

[Development Issues 2](#_Toc479852964)

[Features 2](#_Toc479852965)

[Tools Used 3](#_Toc479852966)

[Coding 4](#_Toc479852967)

[Software Development Methodology 4](#_Toc479852968)

[Future Improvements 5](#_Toc479852969)

[Appendix 6](#_Toc479852970)

[MVC Class Diagram 6](#_Toc479852971)

[Basic User Diagram 6](#_Toc479852972)

[Full Class Diagram 7](#_Toc479852973)

# System Requirements

All specifications given by the clients have been met, with some of them being adapted due to how vague they were. A keyboard-controlled menu is included along with a single player mode and a multiplayer mode. The single player mode ­— also referred to as campaign mode — involves a paddle controllable by a player facing three AI. The multiplayer mode can be configured so as to accommodate up to four players, with non-players being controlled by AI. When a game is launched, a countdown is displayed with paddle and ball movement disabled until a three second countdown runs down and the game starts. When the game starts, the ball starts moving in a random direction from the middle of the game screen with a random velocity. The game window has been made a square shape of 768x768 pixels in order to promote a more fair game between all players. When the ball collides with an object, it bounces in a predictable manner. Each bounce has a degree of randomness in order to prevent AI getting stuck in loops. The ball interacts correctly with objects, if a wall is hit then it is destroyed. If a warlord is hit then player is defeated, the warlord is destroyed and the paddle disappears.

In order to keep the game fast-paced, the game cannot go on for longer than 2 minutes. A timer on the game screen allows players to keep track of the remaining time. A pause screen is accessible by having the player press the "P" key and the game can be terminated by hitting the "Esc" key. The game is over when one of the following winning conditions have been met: only one warlord is alive, or the timer has run out. If the timer has run out, the winner is determined by finding the player with the highest number of walls while still alive. In order to help test this functionality, the "PgDn" key can be pressed to instantly force the timer to run out. Appropriate sound effects are played when the ball hits an object.

# System Outline

Our system follows the Model – View – Controller (MVC) software structure. The model classes contain the basic data and implementation for each of the game elements, while the view classes handle the graphical user interface and the various visual gameplay elements. The main controller classes are used to link the models and views together, and provide overall logic and structure. A basic top down MVC class diagram for our system is included in the appendix.

The WarlordsController class is the main controller class for our system. It contains the public static void main() function and is the class that is initially created when the system is run. Additionally, this class is responsible for creating the root window of the game GUI, creating JavaFX views from .fxml files, and linking with every view controller. This class is also responsible for creating instances of the game model and passing them into the game view.

We have model classes for the game, paddles, warlords, balls, and walls. Each of these classes implement their corresponding given interfaces and contain methods which override the given interface functions. These models mostly store data about each of their instances, such as their positions on the game screen, and if they have been destroyed yet. Additionally, they contain some public static variables, such as their sizes, which are utilised by other model and controller classes. These models also contain getters and setters for their private instance variables. The Game class also includes much of the logic for each in game tick. This logic includes the ball movement, ball collision detection, paddle movement, and some usable player abilities. The game logic could be moved to a separate game controller class in the future to more strictly follow the MVC structure.

Each view was designed using Gluon SceneBuilder to create .fxml files, which detailed layouts, shapes, sizes, and more. Each view has a corresponding view controller class, which handled graphical logic for that view. For example, the MainMenuViewController class handled the arrow key movement and dynamic keyboard menu selections for the main menu view, detailed in MainMenuView.fxml. The GameViewController handled the graphical logic for the gameplay itself, by linking the shapes on the screen to the individual objects of the Game instance. All the logic was completed separately from the view controller, which only handled displaying the results of that logic, in true MVC fashion. However, the GameViewController was responsible for counting in to the game, creating event handlers for key presses and releases, pausing and exiting the game, and timing the tick and countdown timer events.

# Development Issues

We had significant issues with developing the ball collision detection part of our system. Our final collision detection system calculates every pixel that the ball will pass over while travelling during its next tick. The ball will rebound if one of these pixels is a boundary, or an edge pixel on a paddle, warlord, or wall. We had issues with balls bouncing multiple times when hitting an object, and balls bouncing on the insides of objects. These issues were rectified by only accounting for collisions on edge pixels, and including a minimum tick delay where a ball cannot bounce if it has just bounced recently. Another issue was that this process was hugely memory intensive, as it had to calculate possible coordinates for every pixel on the 16x16 pixel ball. This was so that the collisions would appear realistic, preventing any overlaps between objects and balls. We managed to fix this issue by only calculating coordinates for every 5th pixel, dramatically reducing our memory usage.

Additionally we had significant issues with the graphics design for our game. Our game logic considered the bottom left corner of each object as its root (0, 0) coordinate, whereas JavaFX considered the top left corner of each object as its root. Additionally we implemented our paddles to follow a curved path, and incorporating the rotation of the paddle to maintain its orientation to the warlord was difficult. Further graphics issues included the use of animations for wall and warlord explosions, and ball animation. Animation issues were solved through the use of animation timers which displayed individual frames for set amounts of time to create an animated graphic image. Extensive research with JavaFX documentation on the Oracle website was sufficient to solve most of these graphics issues.

# Features

In order to encourage player enjoyment and interactivity, various other features have been added. The menu, game and endgame screen all have a unique music theme, all appropriate with the theme of the game. The menu screen is clear and easy to follow, with sound effects played when the player scrolls or selects an option.

To increase personalisation, 8 unique classes are available for players to use, with each class having a special ability for use in game. An example is the Britain class, whose ability is for the player’s warlord to become invulnerable for five seconds, once per game. Additionally, different sound effects are played depending on what the ball hits in the game or when abilities are used. Abilities are either passive, cool down based, or single use only. Non-passive abilities are activated by pressing a third key (in addition to left and right keys).

In order to increase the game difficulty, we incorporated configurable options such as a randomness factor, controls options, and ball speed options. The ball speed option modifies the ball's speed while the controls option offers a game variant with controls reversed. The randomness factor handles how random the bouncing angle of the ball will be, providing more of a challenge to players who can easily predict the ball's path. Player personalisation such as uploading a picture used for the warlord or entering their name has been considered but has ultimately not been included in the fears that inappropriate content would be included — especially if the game became available to play online.

The game offers graphics of higher quality with randomly changing game backgrounds and various sprites used for the ball, wall, warlords, and paddles. Additionally, animations have been used with the ball, wall destruction, and warlord destruction. The campaign mode (single player mode) offers a storyline spanning across 8 chapters, with a modern approach used for the story's plotline in order to attract younger audiences, while still keeping older audiences interested. The campaign mode can be continued at any time while the game is open, allowing the player to start the campaign, play a multiplayer game and continue the campaign from where it was left at. The campaign's difficulty level gradually increases, with the last level sure to provide a challenge for anyone.

In regards to functionality, our project featured multithreading and java timers. Each timer runs in its own thread, so enabled concurrency in the logic and graphics. Additionally we created our key press and release event handlers in a separate thread from the game logic, which allowed us to have smooth and consistent paddle movement.

# Tools Used

This project was coding in Java, on the Eclipse IDE. We developed on both Linux and Windows, but our targeted platform is Linux. The graphical user interfaces used the JavaFX library and were designed using fxml and Gluon SceneBuilder. Java and JavaFX are both powerful tools used in the software industry for creating cross platform desktop application and rich internet applications with structured and clean user interfaces.

JavaFX was chosen over Swing, as it is a newer and more advanced Java GUI library. More specifically, JavaFX has a much larger reliance on object properties, allowing us to create specific and reliable event handlers. Additionally, JavaFX supports CSS styling, and makes animations much easier.

For our purposes of designing and creating a game, Java and JavaFX are not the most efficient or most resourceful tools that we could have used. A dedicated game design software platform such as Unity would have eliminated much of the difficult ball collision and movement code, as it is specifically designed for creating games with physics and movement. This is contrasted to Java and JavaFX which are designed for general use application development. Coding in Unity would have not only made the whole project easier and quicker to create, but also look and feel much more professional. Unity is specifically designed for games, so adding sprites, animations, and music would be much easier to do.

Java and JavaFX do have some advantages over the Unity platform for this project. Although both work cross-platform across multiple devices, Java is free to use for all platforms, whereas Unity requires payment for development on platforms other than Windows and Mac.

We used the Git version control system, hosted on Atlassian Bitbucket. This allowed us to simultaneously work on the project whilst avoiding breaking each other’s code. Additionally, the version control system allows for reverting to previous commits, in case of project breaking errors. Both of us were more familiar with the GitHub user interface than the Bitbucket user interface, but they were similar enough that we had no trouble.

# Coding

Our system is completely coded in an object oriented design format. All code is contained within classes, with one main class that contains a main function. Abstraction is included in the use of interfaces to design our basic model classes. This reduces code complexity and provides a framework for us to work from. Encapsulation is addressed as each class can only see certain functions and variables from other classes. Additionally our project was separated into separate packages for models, views, controllers, and tests. This reduces code complexity and prevents bugs in one class from dramatically affecting other classes. Inheritance and polymorphism were not fully addressed in our solution. Our warlord, paddle, and wall classes should have all been derived from an object superclass, which would have allowed for much more code reuse and better code structure. In future iterations of the project, this issue would be addressed.

Most of our model classes are high cohesion, due to their main functionality being to store instance variables relating to their positions and current status in the game. The game class is not so cohesive, as it includes a lot of the game logic – which should be in a separate game controller class. This is our view and controller classes which are also mostly high cohesion due to their focus being solely on their purpose – displaying and running the game and menus. Overall we were aiming for a high cohesion software design, and our final solution is a solid effort at that aim.

We designed all of our classes to be as low coupled as possible, whilst still being able to fully interact in the game scenario. We did this by having all of the logic in one class, instead of spreading it out over multiple object logic controllers. This meant that small changes in an object or view class would usually only affect one or two other classes, rather than requiring a complete system redesign. As we got closer to the deadline, some of the coupling between classes increased as the additional player abilities began to affect multiple classes. However we think that we handled the issues fairly well and that our code is well structured and designed with ease of iteration in mind.

# Software Development Methodology

We followed a test-driven design approach for the first part of the project. Using given test cases and interfaces we created the implementation for much of the basic game logic and models. It was especially important to use the test driven design for this first part of the project, to ensure a solid and error free platform from which to implement graphics and further advanced features. The test-driven design allowed for us to easily measure our progress and work out if we had any initial development issues.

After the initial part of the project, we adopted an agile-type software development methodology for implementing the game features and design. As specified in our initial project timeline, in the first week we had the basic implementation of paddles, warlords, walls, and balls complemented. Over the next few weeks of the project we each worked on building and improving parts of the project, iterating over the previous developments. Examples of these iterations would be improving collisions, or adding a graphical interface to the project. This agile-type software development proved especially useful for developing additional features, as the development of each feature corresponded well to each iterative solution.

# Future Improvements

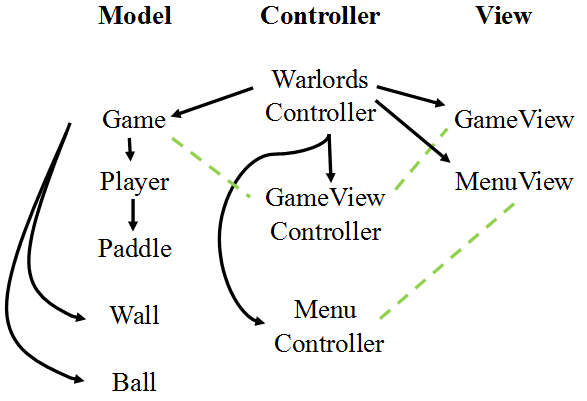
An improvement that we could make in the future would be to improve the AI so that they can use their class abilities. Currently AI simply follow the ball around the screen, positioning themselves to always be between the ball and their warlord. AI could be improved by allowing them to use their class ability, and also add logic so that they can decide when to use or not use their ability. Another improvement to the AI could be so that they can predict where a ball will be after a bounce. Currently AI do not account for ball bounces in their positioning, and they would become much more effective if they could do so.

Another improvement that we could make would be to the graphics of the game, and how the graphics interact with the physics. For example we currently use simple png images for sprites, and simple rotate transforms for paddle orientation. This could be improved by adding further animation to the ball (i.e squashes slightly on impact) or using better quality images with top down 3D views. Additionally we could add animation for paddle movement, ability use, and pause and exit screens. We would like to make the game look as close as possible to modern day arcade games, so that it would not look out of place whilst playing on an arcade machine. This might not be fully possible due to the limitations of JavaFX, but any future development would include significant graphics improvements.

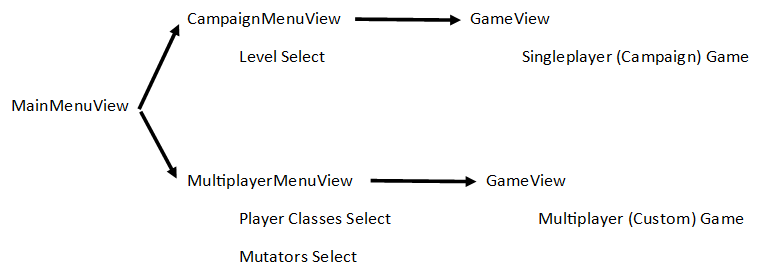
The campaign could also be improved in future development of this project. We would like to add some sort of video cut scenes or image overlays with the story as captions over the media. The campaign is only 8 levels long at the moment, and would only take a few hours to complete. We would like to make the campaign longer, with a deeper story and more characters. Additionally we would like to add meaningful choices into the campaign so that the player’s choices affect the outcome of the game.

# Appendix

## MVC Class Diagram



## Basic User Diagram



## Full Class Diagram

