WARLORDS REPORT

Brought to you by PixelCollider

Jerry Fan & Chris Ding
nano@pixelcollider.net

1 Introduction

This report details the process and implementation of the game made for the client. The client requested a simple game for his twelve year old son. We proposed a plan to build a game based on the retro game “Warlords” where up to four players / AI bounce a ball object with paddles to defend player forts positioned at the corner of the screen. The game would have simple mechanics that can be expanded upon with additional features to add interesting twists and new ways to play the game from the original.

We believe our current product has accomplished this end goal of providing a simple game that can be played single or multiplayer.

2 Requirements

The minimum requirements for the system has been met in our product. Upon starting the game, a title menu is shown for players to select game options or set up a match. The player has free selection of what goes where through a player selection interface, where either bots or player controlled forts are assigned to the corners. After player selection, a countdown is issued for players to get ready, after which players can move a shield (paddle) to deflect a ball that travels starting with a random direction. The ball bounces off all objects unless powered up, and will always be deflected by the boundaries of the window. A ball collision with objects will destroy it in most cases. A collision with the warlord will destroy the related shield unless ghosting is active. A status bar is available for keeping track of time. The escape button will exit the game. The P button will pause the game. The menu can be navigated through the use of the keyboard. Win conditions are checked at the end of each game or upon reaching the time limit, resulting in a player victory or draw depending on game mode and outcome of the match. The ball has collision sounds implemented on object collisions.

3 Features

Additional features were implemented beyond the minimum requirements to make the game more interesting and add additional mechanics.

3.1 Scoring

A scoring system was implemented to reward players for interacting with the ball. The active scoring player is the last player that deflected the ball with the shield. Players will get points when they destroy opponent walls or warlords. Destroying one’s own wall will not award any points. The score system also decides who wins should the game timeout with more than one warlord left alive. Scores are recorded at the top of the screen.

3.2 Powerups

Powerups were implemented that give the ball four different powers of invisibility, haste (speed boost), double damage (object penetration), or award bounty (score) to the scoring player. These will spawn in one of four spawn points randomly every 20 seconds. Effects will expire after a certain amount of time on the ball, or upon hitting a player owned object.

3.3 Unfair Bot

A second level of AI difficulty was added to give an additional challenge to players playing against AI. Whilst the normal AI used a deterministic ball trajectory based method of collision checking that could only see forward by one bounce, the hard AI uses simple heuristics when no collision is detected, simply moving the shield to the closest point on the rail to the ball.

3.4 Ghosts

Ghosting was added to the game to give dead players a chance to still earn points, or to mess up other player's attempts at doing so, although ghosts cannot win the game. Ghosts will retain control of their shield, but their warlord is set to be non-physical and cannot impact the ball. The warlord will appear transparent when it is a ghost.

3.5 Match Options

Match options selection was added to the menu so that players can have free selection of what game mode they want, and fill in the corners with either bots, players, or nothing.

3.6 Background Music

Background music was implemented to create a more competitive atmosphere. The volume can be controlled through a settings menu.

3.7 Unpause Countdown

An unpause countdown was implemented such that players can get ready before the unpause happens, much like the game start countdown.

4 Developed System

4.1 Window System

The main game window is provided by JavaFX on which the menu and game are displayed. The system operates based on scenes, which routes between the game and the menu components (see design considerations for explanation on components) as required. Scene changes are setup as observable objects, such that components can reactively use scene changes to self-load when they become the active scene, through the implementation of observer methods.

4.2 Master Timer

Our game uses one master timer service that controls the progression of the game. This timer is extended from JavaFX's AnimationTimer class and has a target interval between frames of 1/60 s, correlating with a 60 FPS target framerate. The AnimationTimer provides a timestamp based on the system clock, from which a global time can be inferred. The advantage of using the system timestamp is that regardless of what happens to the game process, the time will always progress correctly if the host OS implements the correct system clock. However, the system does not rely on the AnimationTimer specifically, a substitute time keeping method can be used in its place, as long as the timer service's methods remain functional. The timer also has an observable set up such that progression of the time is broadcasted on each frame to observers, making these observers reactive to the progression of time and able to implement time based methods. From this master timer, child timers can be created for use, such as the game timer which exclusively track game time (different to global time).

4.3 Loops

The game operates off two main loops, the game loop and render loop. Both loops are observers of the timer service and operate on the same clock. However, only the game loop is dependent on a steady time interval between timer frames. The render loop can operate regardless of the time interval.

The JavaFX canvas used as the main display for the game uses the render loop to clear itself and display game objects that have a visual presence in the game. The render service allows classes that implement the CanvasObject interface to hook into the render loop and display itself on each frame.

The game loop is used for computations such as collision checks, object movement, and AI actions. The game loop uses an interval argument that represents the amount of time to progress the game forward by. For example, the use of a 1 s interval would cause objects to move ahead using their current velocity by one second. Similar to the render service, a game loop service allows classes that implement the Looper interface to hook on to the game loop. Due to the timing requirements, each call to the game loop runs through a scheduler first. Loop intervals that are larger than the target interval (default 1/60 s) are broken down into smaller intervals equal to or lesser than the target interval. The loop is then iteratively run as many times as needed using the smaller interval. This prevents any sudden jumps in time causing a larger than normal interval, such as when the process is suspended. Several child loops are also called from the game loop, which powers services such as the AI. They use a modified interface but otherwise operate the same way.

5 Challenges

5.1 Collaboration

One of the major challenges in this project was collaboration in coding. At the beginning of the project, through our use of Git, merge conflicts occurred frequently since the codebase was quite minimal and we commonly worked on the same file. This presented an issue as merges conflicts took significant time to manually resolve, and was prone to error.

We resolved this issue eventually as the component based MVC architecture started taking shape and we moved on to working on individual components instead. This meant that merges could be done automatically as there were no conflicts between different file versions. We also used messaging to collaborate on which files we were working on when working on shared components to reduce the number of conflicts.

6 Tools Used

6.1 Java

Java was used as the primary language for developing our game. Java is a typed and JIT compiled language that runs code in a virtual machine. This makes Java code able to run on any platform that has the JVM installed. As no target platform was specified, making the game able to run on multiple platforms is a consideration that Java fulfills. Java is also a high-level language with many built in packages containing a wide range of classes such as Lists and Maps. This makes developing software easier and much faster compared with lower level languages such as C++ for example, where we would have to roll our own classes. However, the tradeoff for these pre-built classes is that Java applications will always be slower than applications built with a low-level language such as C++ due to the extra code that would be required to standardize these classes. In the context of our game, the game is simple enough such that the timing requirements are not on the level of a few milliseconds, and our time constraint for development is a more pressing matter, suggesting the use of Java was justified.

6.1.1 JDK 1.8

An added benefit of using JDK 1.8 is that we have access to functional expressions that use anonymous (lambda) functions. This means that instead of using for loops, simple list operations can be achieved using either map, filter, reduce or for Each expressions. This makes code more concise and reduces the chance of introducing errors during iteration cycles. The downside to using JDK 1.8 is that it requires all systems running our application to have JRE 8 installed.

6.1.2 JFX 8

By using JavaFX to build our GUI, we take advantage of the built-in separation between the view and controllers for the UI. FXML abstracts the view layer into an easier to visualize XML type structure, whilst linking FXML components through the use of an ID to the JavaFX controller. This is complementary to our use of MVC architecture. However, it does place restrictions on how we design our architecture, and how we instantiate FXML components. In some rare cases, using FXML to write the structure quickly became repetitive as the same could be done much faster with a loop in Java.

6.1.3 IntelliJ

The use of IntelliJ proved to be quite helpful in speeding up java development with the live linting and code suggestions. Renaming was a commonly used feature that helped make code more readable, along with the built-in Javadoc generator. It also had Gradle and JUnit support built in.

6.2 Gradle

Gradle was used as our build tool to make our code configuration not rely on any IDE, and usable on any platform. By installing the Gradle wrapper in our repository, any computer is able to run the Gradle scripts to build, run, package or test the application. The only downside to using Gradle is that the first build always takes a long time due to repository checking and dependency checking.

6.3 Git

The use of Git as a versioning and collaboration tool was highly suitable to the rapid development cycle that we used to push out code. Using Git, we were able to resolve many conflicts between different version of a file when working on code simultaneously, and have a general idea of the progress of our development through the Git history. The Git history also allowed us to roll back any unwanted changes that may have been done. This does add an extra step whenever a coding session finishes however, requiring a bit of time to write a commit, but the tradeoff is negligible.

6.3.1 Bitbucket

By taking advantage of the Bitbucket service, we were able to establish a build pipeline that automatically attempted to build our code and report the results of unit tests we include and the build itself. Additionally, issues were used to keep track of bugs or tasks as needed.

6.3.2 GitKraken

In addition to the Git shell, we used GitKraken as a GUI to manage our Git repository. This made merge conflicts easy to resolve compared to using the command line.

7 Design Considerations

7.1 Component Based MVC vs Traditional MVC

A large amount of time went into designing and implementing a game architecture that was extensible and highly decoupled, beyond applying just basic MVC architecture. We applied a more component based model to our application structure rather than the flat MVC model. This meant that our Java packages had vertical hierarchy as well as horizontal, which alleviates the issue of fat controllers containing most of the code processing. Instead, specialized services are used for both data storage and providing functionality where minimal controller access is needed. This meant that we spent longer designing the scaffolding for each component as a large number of files were created per component, but the end result is more concise code that is better organized.

7.2 Decoupling

By not connecting controllers horizontally, the component based MVC architecture reduces coupling between components by limiting interaction between controllers to just parent to child ideally. Where interaction between controllers was required beyond this, the reactive programming paradigm was applied using Java's observer and observable classes, along with event listener interfaces. This allowed observers or event listeners to register themselves as a receiver, being notified whenever a change or event occurs. A large number of interfaces were created for this purpose (Physical, CanvasObject etc).

An additional advantage of using reactive programming in our decoupling method is that concurrency is easier to implement, such that an event on one thread can be listened to in another.

7.3 Cohesion

By using the component model, cohesion was increased as each component was created for a specific purpose. Services were created with the express purpose of doing just one thing very well, whether that be providing some functionality or just data storage for a controller.

8 Development Methodology

8.1 Test Driven Design

Although we planned to use a test-driven design methodology, this quickly became impractical after the prototype as we attempted to roll out as many features in a short period of time as possible. Writing JUnit tests took the same, if not more time than implementing the methods themselves, this did not cover any JavaFX testing. As a result, our testing generally involved the use of visual inspection and debugging. To alleviate the effect of potential bugs, our software architecture was designed such that there would be as little dependency between the different game components as possible, making bugs mostly localized and easy to remove.

In the case of AI development, a debug mode was implemented to see what the AI is thinking at any given moment, which greatly helped AI testing.

If there was less of a time constraint and external libraries were allowed, more unit tests could be incorporated into our repository without sacrificing the development and bug testing of required game features, and allow us to test the GUI implementation.

8.2 AGILE

Our development process is largely in line with the feature driven agile design methodology. We started with a list of required features for the game, and then completed components as required based on the currently developed feature, before implementing the next one. Bitbucket was also helpful in providing the issue tracker to list feature requirements later on and keep track of bugs. This was found to be an effective way to focus programming efforts and target one problem at a time to meet the project requirements.

9 Future Improvements

The visual design of our game can be largely improved by providing a more user-friendly UI system and providing updated pixel art graphics for game components. We could also provide extended resolution options and support for high pixel density screens.

In terms of development methodology, switching to a more test-driven design would be more suitable at this point, as the application is approaching the point where using visual inspection would be more time consuming than unit testing with appropriate testing libraries.

Game physics is another area of improvement that could be worked on. The game could use some more varied physical interaction between objects than the ball bouncing at fixed angles, making the bounces somewhat predictable.