Project A Final Report

COMPSYS 302

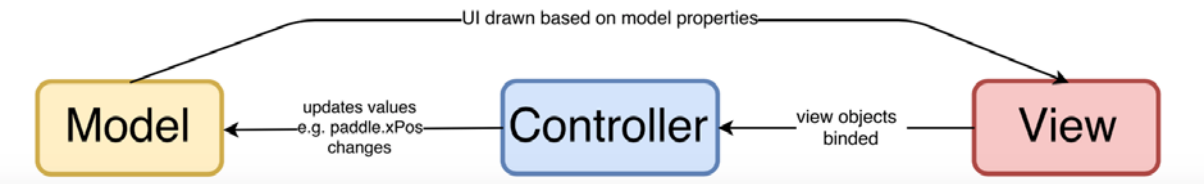
GROUP 19

Overview of System

The developed system we implemented is an offline game application written in Java, which runs on the Linux operating system. As per specification, the game is loosely adapted off the 1980 Atari game, “Warlords”, but with an aesthetic that is both nostalgic for the client and appealing to the client’s twelve year old son. The game had to meet certain minimum requirements, including (but not limited to) single player vs AI mode, local multiplayer mode, and a playable Warlords-style in game view.

The minimum game specifications listed in the original project brief have all been met with our implementation. We have also added additional quality of life improvements to both the user interface and gameplay which will help the game feel more exciting to our younger stakeholders (our twelve year old user).

Top-level View of System



Our game follows a software design pattern known as model-view-controller (MVC). This is a design pattern which separates an application into three parts which are interconnected but independent from each other. This ‘separation of concerns’ allows for easy parallel development and efficient code reusability.

To briefly explain the concept, the model stores relevant data involving the application behaviour. In a Java context, this may be a class which stores certain variables for important attributes about gameplay. The view is in charge of displaying the user interface that is displayed as output, and is usually dependent on the model’s aforementioned attributes to change how the view is drawn. The controller is the glue that connects the model and view together. The controller usually processes some user input which would prompt a change in model attributes, while also updating the view based on these updated model values.

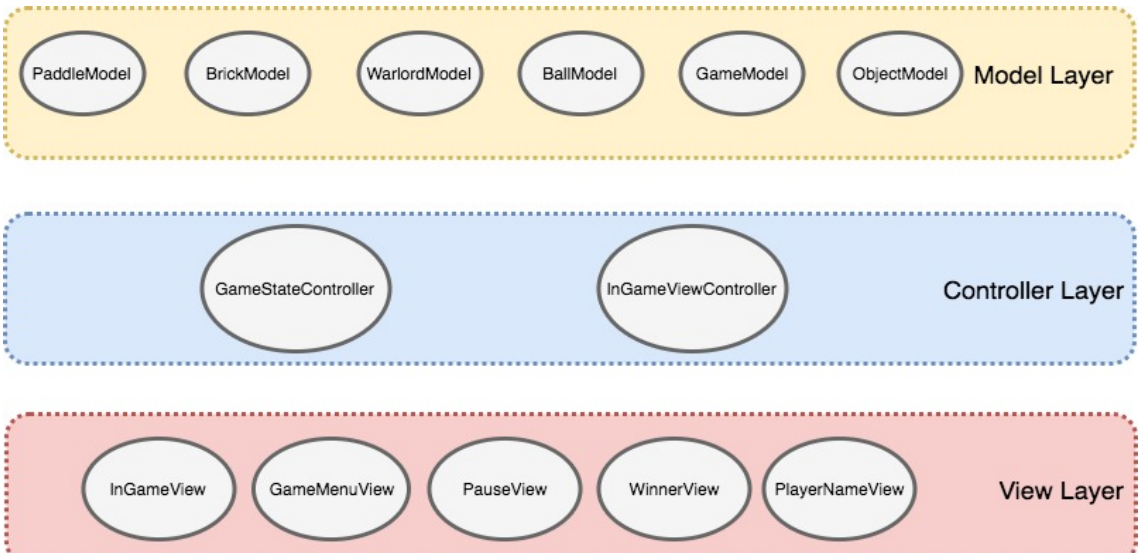


FIGURE 1: AN ABSTRACTED VIEW OF OUR SYSTEM

In the context of our game, the 'Model' components of our system largely consist of the elements that make up the game, for example, the paddle, brick and warlord. These classes hold variables (and their relevant getters/setters) that describe relevant information about the objects, such as position or speed. As many of these components have similar methods, we have also used the OO concept of inheritance in our design, by making all 'placeable' models 'extend' off of ObjectModel, which contains standard methods relating to position, of type DoubleProperty.

The GameStateController determines whether the game is paused, in progress or finished. It also manages the various modes of the game, such as single player or multiplayer modes. This information is used by the other (more important) controller, the InGameViewController, to consolidate the view and models. The InGameViewController does this by creating and managing the game loop in which the view is updated with, as well as providing the logic to update model attributes based on user keypresses.

Lastly, the view layers predominantly consist of pure JavaFX classes which are created to convey information to the user, bar the InGameView which draws and binds models to the game. It is also worth mentioning that GameMenuView provides the start() and main() methods, which in JavaFX signify the starting point of the application.

A detailed version of this diagram complete with how various classes link together is found in the Fig. 1 of the appendix.

Significant Issues During Development

An unforeseen issue during the development of our game was the complexity surrounding the graphical user interface design. As beginners to both GUI design and Java, it took a while to get accustomed to reading Javadoc descriptions to understand JavaFX methods. We were also very confused with Scene Builder and its link towards game design, which was presented to us as a way to build our view. Our team personally did not want the interface to look like a 'web application' with scrolling menus and clickable buttons, so a lot of time was spent learning how to create more 'game-like' views using Scene Builder and FXML. After a week of building the view using Scene Builder, we ultimately scrapped it as we were unhappy with the limitations Scene Builder had. As such, all our menus in our final implementation were created purely from JavaFX - this has led to a much smoother interface, complete with keyboard controls, dynamic 'highlighting' of selected options, and various other features that we found were very difficult to implement otherwise.

Another significant issue we came across in development, was optimising our game. Once all objects were drawn, there was a clear hit in performance. The game became very taxing on the system. In each tick of the game, collisions are checked. Each tick lasts about 16ms, giving a framerate of approximately 60fps. However, with how heavy and un-optimised the collision-checking algorithms were, these algorithms were taking much longer than 16ms to process. Not only did the framerate drop, but the game became very laggy; it would often freeze for many seconds at a time. It was unplayable at times. To address this, the collision detection algorithms were optimised by removing objects from their arrays when destroyed/killed, and algorithms are not run unless they are absolutely needed. Furthermore, collision detection and key input processing were separated into threads for concurrent execution.

Additional Functionality Improvements

To improve the functionality of our system, we have used JavaFX throughout the game to create its GUI. This has enabled us to create an aesthetically-pleasing GUI which is easy to navigate with just the keyboard. When the user selects menu items, feedback is given to the player in the form of visual (highlighting) and audio (SFX). Moreover, on the Quit Game menu, a confirmation prompt is displayed just in case the player accidentally selected quit.

To offer the player an essence of customization, there is an option to set custom usernames for each player. To display this information during gameplay, we have the HUD on the sides of the in-game view. This opens up more area to display important information, such as the timer.

The pace of the game increases as the game progresses through two mechanisms. Firstly, at every 10 second time interval, the speed of the ball increases. Likewise, there is a power-up which can spawn at random times in the game which will increase the speed of the ball. In addition to this, there is also a power-up which adds an extra ball into the playing field. These features overall make the game more fun.

Screenshots of some of these features can be seen in the appendix, in Fig. 3a through to 3e.

For a complete list of the additional design elements in our game, please review the Final Submission Table in the Appendix.

Discussion: Suitability of Tools

Java was the language of choice for our implementation. We felt that given the specifications of the project, this language was suitable for our design. Java supports OOP, which is necessary for a class-heavy application like a game. Design patterns, such as MVC, can be easily implemented in Java, due to the JavaFX API including important methods such as the ability to 'bind' objects on the view to a model attribute. Additionally, due to the portable nature of JVM, Java eliminates any discrepancies in the application across different operating systems - something more platform-dependent languages (such as C++) may have issues with.

The use of Git as a version control system was very helpful. As a team project, Git was an integral part of development which enabled our team to implement new features on our respective branches and easily view new changes to the code. As the current industry standard for software version control systems, we feel Git was a suitable tool to use.

We used the native Eclipse IDE and build tool to write and build our program during development. Eclipse has significant advantages with its ease of use - Git projects can be easily imported, and Eclipse's in-built build tool makes it easy to quickly test while developing our application. For our circumstances, Eclipse was sufficient, and even preferred as it removed the nuances of build tools to our team of Java beginners - however, it is worth noting that in the future, it may be more appropriate to use a build tool such as Gradle in larger projects. This is due to its independence from a specific IDE, its various build automation options, and multiple language support in a single build.

Discussion: Object-Oriented Design Principles

Encapsulation has been achieved in our code by having member variables declared as private within their respective classes, and these can only be accessed/changed by other classes through getter and setter methods. For example, the Ball class in the model layer will have the member variable xPos, holding the x-position of the ball. This can only be seen by other classes through the getXPos() method. Likewise for changing the x-position -- there is a setXPos(int) method within the Ball class.

Where possible, we have sub-classes which inherit from a parent class to reduce repetition of code. For example, we have a class from which all game objects (ball, paddle, warlord, brick) inherit from; these objects all have some common fields (position) and methods (getters and setters for position).

By employing the MVC pattern, this greatly helped in achieving high cohesion. By strictly following the MVC guidelines, wherein each class in each layer has a set role, this ensures that our code is highly cohesive. Model classes should only do and contain what is defined to be the 'model', and likewise for other layers.

As a limitation of the MVC pattern, our system is loosely-coupled to a certain extent. With this pattern, there is a level of dependency between the view and the model, and the controller and the model. By following the MVC pattern, the coupling introduced by this dependency is unavoidable. To reduce the amount of coupling as much we can, we made sure that our individual layers were as cohesive as possible – especially the model layer, as the other layers have dependencies with it. This way, the unavoidable dependency between the model and other layers can be taken advantage of -- for example, to bind the drawn in-game objects to its Model properties.

On the other hand, the model layer is not dependent on the view, nor the controller. This loose coupling meant that one member could be working on the model layer, whilst the other works on the controller or view. This made for easy parallel development, increasing our workflow efficiency.

To further reduce coupling, we have avoided using global variables and rather we pass variables in as an input parameter to other classes' constructors in order to access them. This can be seen in our code in that a single instance of the GameModel class (which instantiates all game objects) is instantiated at the beginning of each game, which is then passed through to the View and Controller classes.

With loose coupling and high cohesion, this makes our code easily maintainable and can be reused in the future. This has been proven as throughout development, we have added extra features and changed existing implementations, which did not require refactoring the entire system.

Discussion: Test-Driven Development

Leading up to the prototype submission, we followed the test-driven development methodology. Having been provided with unit tests, we then wrote production code to pass these tests whilst abiding by the MVC pattern. If the tests failed, the actual result was checked against the intended result from the test. Bugs were then identified and fixed accordingly.

Following such a methodology proved to be helpful as the prototype was largely the basic functionality of the game. If the barebones of the game was wrong, the whole game would not function correctly. Utilising test-driven development ensured that our game, at its barebones, was functional before we progressed any further with adding extra features.

Future Development

Future updates which could be added to our game include different game modes such as Attack Mode, whereby there is a running score which increases as you destroy more bricks. Game modes such as multiplayer co-op and a more in-depth story mode could also be added.

Improvements could be made to the graphics -- that is, using more modern and detailed graphical assets combined with animations. More thorough SFX could also be added, coupled with background music.

Many gameplay elements could also be added to the game. This includes ghosts, different paddle mechanics, and more types of power-ups.

Appendix

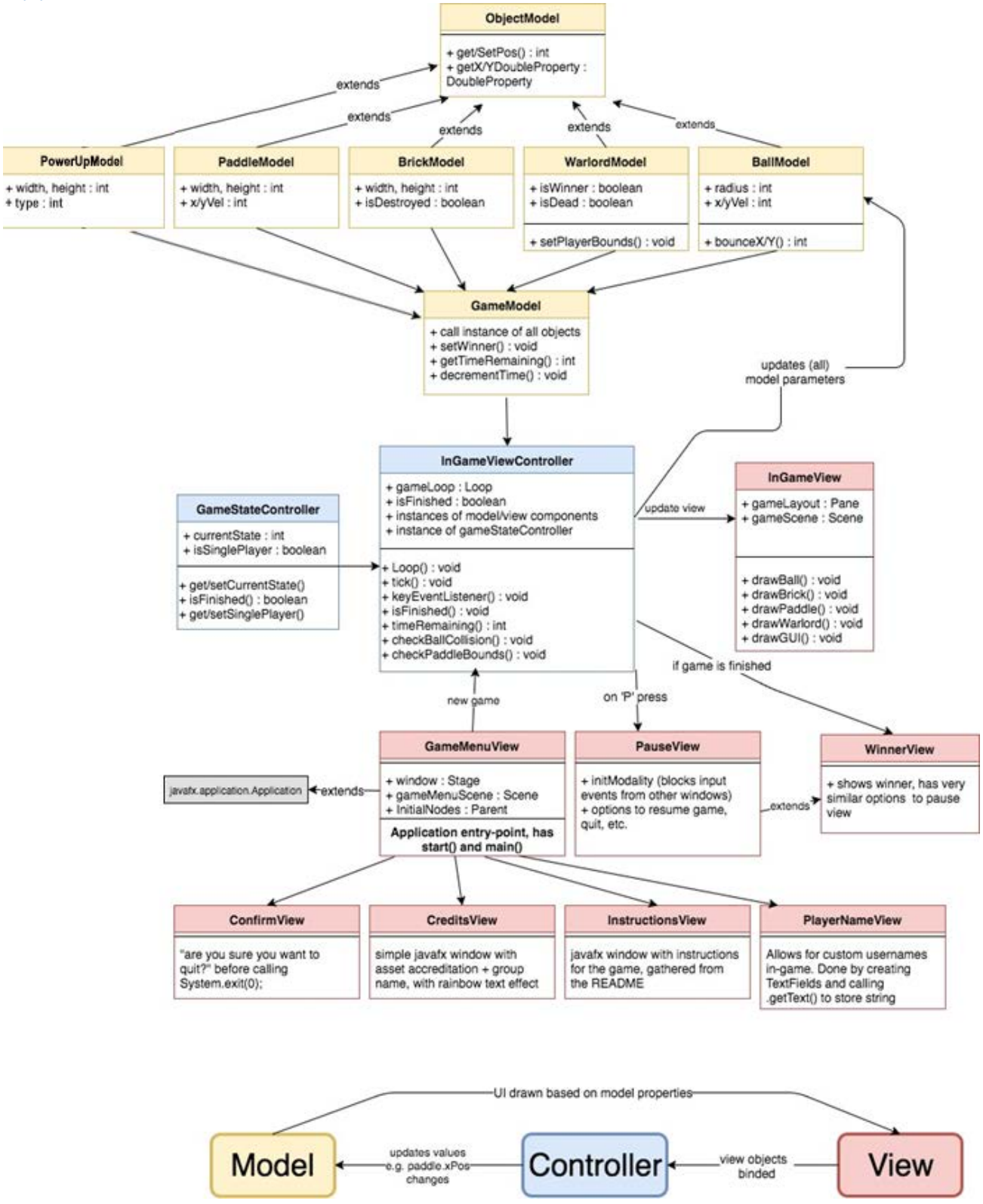


FIGURE 2: CLASS DIAGRAM

FIGURE 3A: MAIN MENU

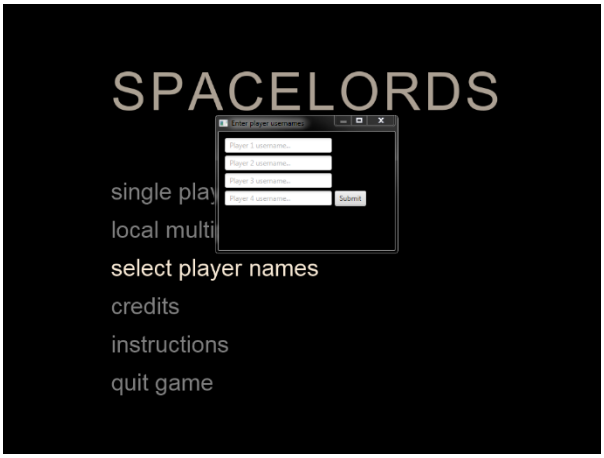


FIGURE 3B: CUSTOM USERNAMES

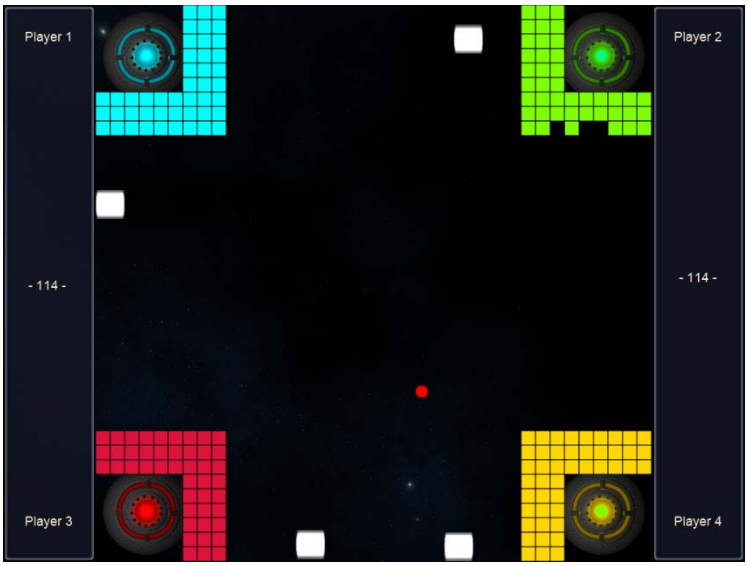


FIGURE 3C: GAMEPLAY

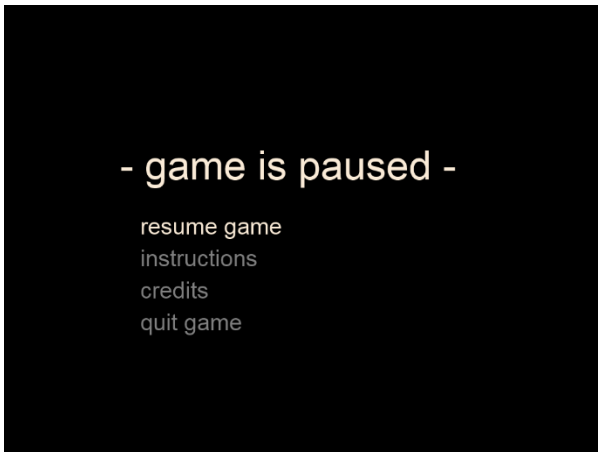


FIGURE 3D: PAUSE MENU

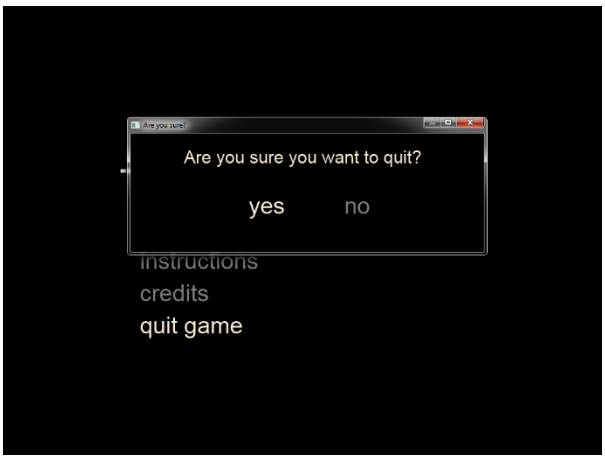


FIGURE 3E: QUIT CONFIRMATION

Final Submission Table

Group Number: 19

	Features (minimum specifications = 50%)	**Yes/ No?**	**Comments**	**Team member(s)**
1	Compiles and runs fine without errors/Code quality – comments, indenting, etc.	Yes		MW (50%), ST (50%)
2	Welcome screen: select a game mode using keyboard, three game modes: single player (vs AI), local multiplayer	Yes		MW(50%), ST(50%)
3	Start game: stationary paddles, countdown timer from 3, paddles should not be able to move	Yes		MW(30%), ST (70%)
4	At least one ball should spawn with random velocity	Yes		ST
5	Objects should not exceed 1024x768 boundaries	Yes		ST
6	Hit registered when ball collides with wall, event(s) follow (e.g. wall being destroyed)	Yes		ST
7	Ball should bounce off paddles and window edges predictably	Yes		ST
8	Hit registered when ball collides with base, destroying warlord and related paddle	Yes		ST
9	Game can be paused/resume with 'p', exited with 'Esc', back to main screen	Yes		MW
10	Win condition evaluated, exit screen at end of game with summary, PgDown to skip to exit screen	Yes		MW
11	Appropriate sound played for any collisions	Yes		MW

	Design Elements	**Team member(s)**
1	Game design follows MVC pattern	MW (50%), ST(50%)
2	Space style aesthetic, which is in line with the game story	MW (80%), ST (20%)
3	Menus can be navigated purely on keyboard, allowing for a cohesive experience without having to use the mouse	MW
4	Use of JavaFX throughout the game – notable effects: highlighting current selected menu option, use of transitions/animations to style text, creating the in game GUI	MW(80%), ST (20%)
5	Confirmation screen before 'quit game' is called, to make sure players don't accidentally quit the game	MW
6	Custom usernames can be selected, and shown in-game	MW
7	Use of sounds throughout game (including in menus) to create a more dynamic experience	MW
8	Single Player vs AI mode relates to story by providing text prompts on successive round completions	MW
9	Ball gets faster as game progresses	ST
10	Power-ups	ST
11	AI	MW (20%), ST(80%)