Compsys 302 - Java Game - Final Report

By Savi Mohan & Ira Syamira

The minimum requirements met by the system

Our system meets all the minimum requirements for the project.

The system developed has a functional welcome screen menu that allows the user to navigate to the single player mode, multiplayer mode, instructions ,credits screen as well as an AI demo. Along with the single player mode, the multiplayer mode can have between 2-4 human players. In single, two and three player modes, the remaining non-human players are controlled by the AI.

The game starts off with four stationary paddles. A countdown timer will start counting down from three before the actual two minutes starts and only then will the players be able to start moving their paddles.
When the starting countdown finishes, at least one ball is spawned in the middle of the game, moving with a set velocity in a random direction.

The objects are created within the 1024x768 boundaries of the game window and are not able to move beyond them.
When a collision is detected between a wall and a ball, the respective wall object is deleted. This will remove the wall from the screen.

The ball object has been designed to bounce off in a predictable mirrored path when a collision is detected with the screen boundaries and any other objects within the system(including paddles).

When a collision between a ball and a player occurs, the player is killed, then their player icon becomes a 'skull and crossbones' to signify their death, but the paddle remains so that the player can still keep participating in the game.

Each level takes no more than two minutes to finish. A countdown timer is displayed at the top of the window.
During the gameplay, the key 'p' will allow users to pause and resume the game. If the 'Esc' key is pressed while the game is paused the game will exit back to the main menu.

The win condition is checked when any of the following events occurs; 1. When three players have been killed in multiplayer mode 2. When human player is killed in single player mode 3. When two minutes have passed in a certain level. If the level's time limit has been reached then the winner is determined by which alive player has the most walls remaining. The 'PgDn' key will set the timer to 0 seconds left for the respective level to help with testing.

A collision sound is played at any ball collision with another object.

Top-level view of how the system works (full MVC in appendix)

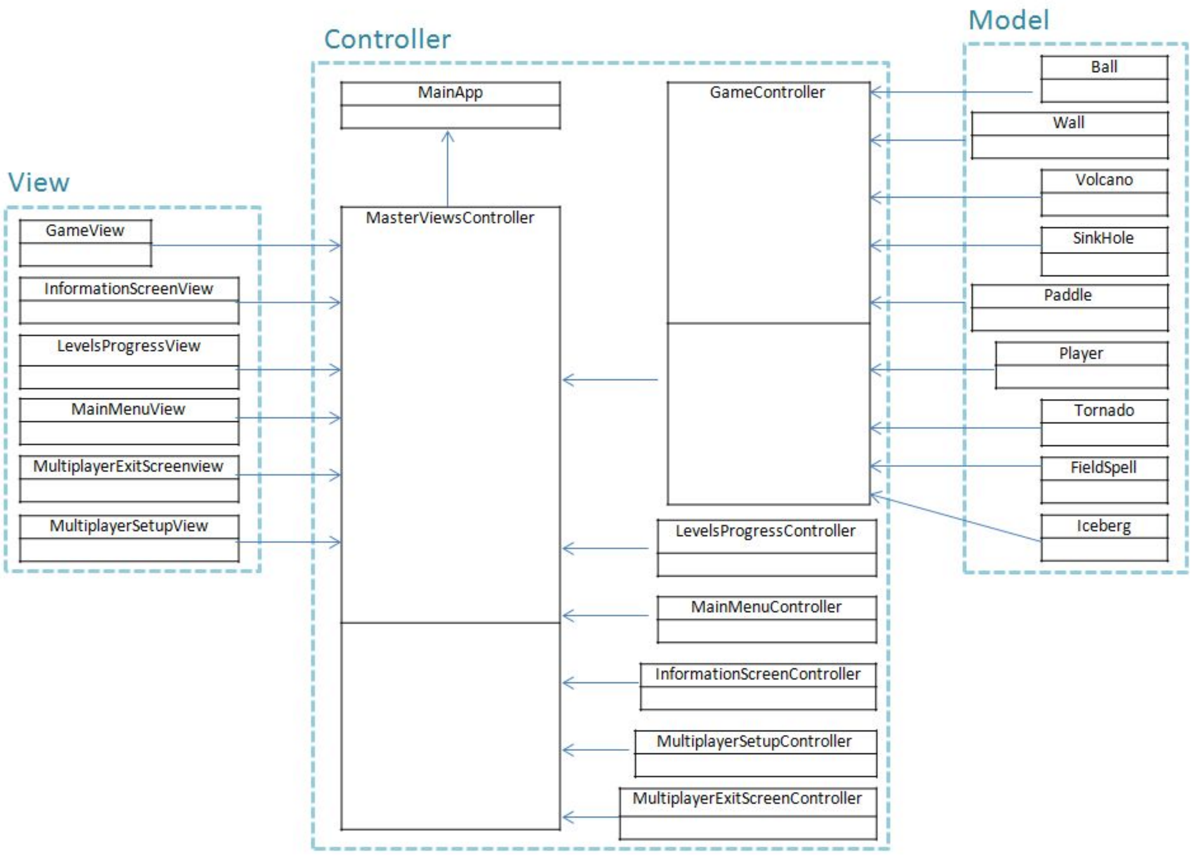


Diagram 1: The simplified class diagram of the system

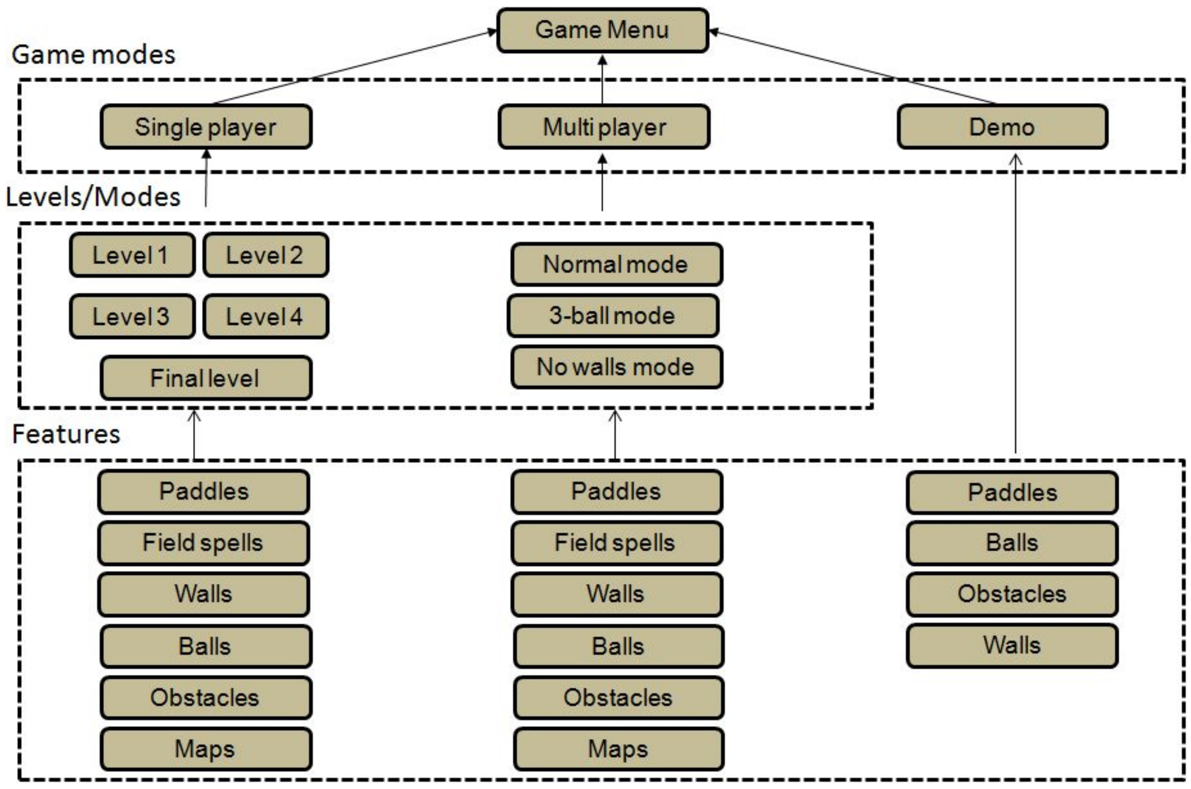


Diagram 2: The top-level view of the the system

Diagram 1 above shows how the classes from separate packages interact with classes from the same package and classes from different packages. The full class diagram which includes all the methods is included in the appendices. Diagram 2 is the top-level view of the system which shows the clients how the system works and the options/features they encounter throughout the system.

Significant issues encountered during development

A significant issue that arose was memory leaks in our code. We noticed that when a game level ended and went to the next level there was a spike in RAM usage that kept increasing until the game crashed completely. We figured out that this was being caused by the previous scene instance not being properly destroyed, ie. instead of the gameScene variable being overwritten a new gameScene variable was being created. To fix this issue we created the gameScene variable in the header of our ViewController class so that every time a new game scene was created it was assigned to that gameScene variable, ensuring that the previous game scene instance was destroyed.

Another significant issue then arose where we noticed that the game timer kept getting faster and faster every time the game level was restarted. We realized that the game controller's tick() function was being called at shorter and shorter time intervals resulting in the game timer counting down faster that it should've. To fix this issue we ensured that every time the game level was restarted, the KeyFrame instance (which controls how often the tick function is called) was cleared from the animation Timeline before assigning a new KeyFrame instance to the Timeline.

Features that improve system functionality

On top of the minimum requirements, additional features have been implemented in order to enhance the functionality of the system. The additional features we implemented help differentiate our game from previous versions of the Warlords game as well as making our game more appealing to users by making it more challenging and engaging. Some of the additional features included are obstacles and game spells.
We have four different field obstacles in the game (volcano,iceberg, sinkhole and tornado). The tornadoes randomly deviate balls that collide with them from their original path. The tornado is a circular object that has been set to rotate about a fixed point and radius. The icebergs are rectangular objects moving in the y-direction deflect balls from their paths. This has been implemented by creating a rectangle object and having it move at a fixed velocity within a specified boundary. The sinkholes teleport balls that collide with them to another sinkhole on the map. The volcano erupts fireballs in a fixed direction and at fixed time intervals. These fireballs will destroy a player or a wall when a collision happens. The fireballs are treated the same as any other balls except that they disappears once they collide with any object and only bounce off the screen boundaries. This is to ensure that the game screen doesn't get clogged with too many balls.

The game also has four field spells(ball speedup,disable obstacles,disable walls and paddles lock) that can temporarily alter the game conditions. The field spells' effects stay active for five seconds once activated. The field spell objects are added to the map at four different points. When a field spell has

its effect activated, it then disappears from the game screen for a specific cooldown time, after which the field spell reappears on the map.Whenever a collision occurs between a ball and a specific field spell object, that field spell's effect is activated.
For instance, when a ball collides with a 'paddle lock' field spell, the paddle movement function in the code is stopped from being carried out momentarily, preventing all paddles from being moved. This applies similarly to the obstacle freeze field spell, which prevents the obstacles from moving. The 'ball speedup' field spell doubles the velocity of any ball that crosses it. While the 'disappearing wall' field spell makes the walls transparent and prevents balls from colliding with them(ie. balls go straight through walls), leaving the player bases 'exposed'.

Another feature we implemented was rotating paddles that moved in a circular path. This was an improvement on the minimum requirements which only needed the paddles to be able to move in 1 dimension. The rotating paddles also gave the game a more modern feel.

The Multiplayer mode has the option to have up to 4 human players, all of whom can play together on a single keyboard. This allows for a more satisfactory user experience as more people can play the game at once.

The Multiplayer mode also has the option to choose between multiple game modes (normal , no walls mode and 3 ball mode). This serves to increase the variety of experiences possible with the game.

In the multiplayer mode, players can also enter their names in textfields to have them be displayed on the game screen next to their player bases, which improves the feeling of immersion in the game.

On top of the whole system being navigable by keyboard, the system can also be traversed by mouse (ie. use mouse to click buttons). This makes the UI more easy to use and intuitive.

In order to showcase all the features of the game (field spells and obstacles), an AI demo mode is also available where all the players are computer controlled.

The Single Player Mode has a storyline and multiple levels. The storyline has an elemental theme and follows a character on their conquest of a fantasy themed world. At the end of each level a screen is displayed showing more of the story. The landscapes of the levels (in terms of obstacles and visuals) are unique. This combination of story and multiple varied levels helps to improve player immersion and engagement with the game.

Discussion of the suitability of the tools for the application (e.g. Java, Git)

Since Java is an OO language, it was an extremely useful tool in creating our system. OOP has made managing, adding and removing data much easier. The system as a whole becomes a bunch of loosely coupled modules that communicate with each other via methods. It is a great alternative to the traditional way of programming in which data and a separate algorithm operates on the same global set of data. Also being able to treat objects in the game as objects in the code made it much easier to conceptualise how our code needed to be structured. The cross platform support for Java meant that it

was easier for us to develop the system in Windows and then still be able to run it on Linux with minimal compatibility issues.
The minimum coupling and high cohesion of OOP allows the development of systems on a larger scale. The objects and their attributes become the building blocks of the whole game system.

We used the JavaFX library as our GUI library. The JavaFX library components allowed us to create a more modern looking game in terms of visuals and gameplay experience.

Memory management for our system was made easier thanks to the Java garbage collector. Objects that are no longer reachable and referenced are automatically deleted, saving us the hassle of having to delete heap objects when we no longer need them.

The use of Git was a very important aspect of this project. Version control allows us to retain previous versions of the code, making it more convenient to access specific versions after making amendments rather than having multiple backups on personal computers with unorganised naming conventions. It also allows us to figure out the source of errors by tracing changes since the last stable build.

Apart from version control, git made it easier for both of us to work on the code simultaneously as each of us could simply pull the code from the online repo, then commit and push the changes that we made on our local machine.

Discussion on OO design and how cohesion and coupling issues were addressed

High cohesion and low coupling are essential in making sure that a system is reusable and easy to modify. Having high Cohesion is the OO design principle that is closely associated with making sure a class has a well-focused purpose. While having low Coupling is the OO principle meant to ensure that different components have minimal interaction. OO design requires a class to only function within its own boundaries and limits its knowledge of other classes.

High cohesion and low coupling in our game code was ensured through the use of the MVC design pattern. MVC forces us to make sure that all the data pertaining to game objects are kept in model classes, while view classes are the only classes allowed to plot that data, and controller classes are the only ones allowed to interface with the user as well as control what data from the model classes gets plotted by the view classes. This Separation of Concerns (SoC) ensures low coupling (as there is minimal interaction between components) and high cohesion (as each class only does one or a few functions).

The use of OO significantly reduces code duplication in this system. Since there are multiple players, paddles and other repeated elements, new instances can be created without having to rewrite the whole chunk of code. The reduction of duplicate code eases code maintenance. If any properties of the class needs to altered, all the objects of the class type will have the new altered property instead of having to change everything when chunks of code have duplicates.

Discussion of the software development methodology

Before the prototype submission, our software development process was test-driven since we wrote our code to pass a fixed set of test cases. However for the rest of the project, our developmental process was based more on Agile principles. Agile was suitable for us as it is geared around working on short timeframes, and we only had a few weeks to complete the game.

After the prototype stage, we focused on continuously delivering working software. Everytime we added a new feature we ensured that we still had a working version of the game. We were always open to changing the final requirements for the game. We always tried to communicate and work together face-to-face for maximum efficiency. We also regularly discussed our progress on the game, what we completed so far, what needed to be done, what we were doing well and what we weren't doing well.

In hindsight we think that we should have continued carrying out test-driven development after the prototyping stage, so as to minimize the amount of manual testing that we had to end up doing.

Suggested improvements for future development

In the future, we hope to produce systems with a more customisable gaming experience for users. The ability to customise features in games are important as users are always leaning towards seeing themselves in the game rather than engaging in a game to which they have no relation to.Whether it is character customisation, game graphics customisation or control keys customisation, having the ability to enjoy a game with personal preferences incorporated into it may be one of the main factors in the popularity of the game.

We would also like to implement concurrency/threading in the future so as to improve game performance especially in game levels with a lot of objects on the screen. We would also like to implement more game modes as well a high score screen, so as to increase competition between players.

APPENDICES

View

GameView
GameView()

InformationScreenView
InformationScreenView()

LevelsProgressView
LevelsProgressView()

MainMenuView
MainMenuView()

MultiplayerExitScreenView
MultiplayerExitScreenView()

MultiplayerSetupView
MultiplayerSetupView()

Controller

MainApp
mediaPlayer : MediaPlayer
main(String[])
music()
start(Stage)

MasterViewsController
MasterViewsController()
goToGameScene()
goToInformationScreen()
goToLevelsProgressScene()
goToMainMenu()
goToMultiplayerExitScreen()
goToMultiplayerSetup()

MainMenuController
buttonIndex : int
buttonList : ArrayList<Button>
masterViewsController : MasterViewsController
menuScene : Scene
root : GrouP
MainMenuController(MasterViewsController)
initialise(int, int)

MultiplayerExitSetupController
buttonIndex : int
buttonList : ArrayList<Button>
gameLevel : int
gameMode : int
humanPlayerCount : int
masterViewsController : MasterViewsController
multiplayerSetupScene : Scene
root : Group
setupStage : int
setupText : Text
MultiplayerSetupController(MasterViewsController)
initialise(int, int, int)

MultiplayerExitScreenController
gameMessage : String
masterViewsController : MasterViewsController
multiplayerExitScreenScene : Scene
root : Group
MultiplayerExitScreenController(MasterViewsController, String)
initialise(int, int)

InformationScreenController
displayInfo : int
informationScene : Scene
masterViewsController : MasterViewsController
root : Group
InformationScreenController(MasterViewsController, int)
initialise(int, int)

LevelsProgressController
level : int
levelsProgressScene : Scene
masterViewsController : MasterViewsController
playerNames : ArrayList<String>
root : Group
LevelsProgressController(MasterViewsController, int)
initialise(int, int)

Controller

GameController
ballList : ArrayList<Ball> durationMilliseconds : double enterButtonPressed : boolean escapeButtonPressed : boolean fieldSpellsList : ArrayList<FieldSpell> fieldSpellText : Text frameRate : int gameIsTied : boolean gameMode : int gameScene : Scene humanPlayerCount : int icebergList : ArrayList<Iceberg> isGameFinished : boolean isSinglePlayer : boolean level : int masterViewsController : MasterViewsController obstaclesFrozen : boolean paddleList : ArrayList<Paddle> paddlesLocked : boolean pauseMenuEnabled : boolean player1Walls : ArrayList<Wall> player2Walls : ArrayList<Wall> player3Walls : ArrayList<Wall> player4Walls : ArrayList<Wall> playerList : ArrayList<Player> playerNames : ArrayList<String> random : Random root : Group singlePlayer : Player sinkHoleList : ArrayList<SinkHole> spawnBalls : boolean stringTimer : String timeRemaining : double timerText : Text tornadoList : ArrayList<Tornado> volcanoList : ArrayList<Volcano> wallsInvisible : boolean winner : Player

GameController(MasterViewsController, int, int, int, boolean, ArrayList<String>)
addBall(Ball)
addPaddle(Paddle)
addPlayer(Player)
addPlayer1Wall(Wall)
addPlayer2Wall(Wall)
addPlayer3Wall(Wall)
addPlayer4Wall(Wall)
checkBallCollision(Ball)
findClosestBall(Rectangle)
getTimeRemaining()
initialise(int, int)
isFinished()
setTimeRemaining(double)
tick()

Model

Wall
bounds : Rectangle wallDestroyed : boolean xPos : int yPos : int

Wall()
Wall(int, int)
destroyWall()
getBounds()
isDestroyed()
setXPos(int)
setYPos(int)

Volcano
bounds : Circle coolDownTime : double

Volcano(int, int)
getBounds()
getCoolDownTime()
setCoolDownTime(double)

FieldSpell
bounds : Circle coolDownTime : double type : int

FieldSpell(int, int, int)
getBounds()
getCoolDownTime()
getType()
initialise()
setCoolDownTime(double)
setType(int)

Paddle
bounds : Rectangle moveLeft : boolean moveRight : boolean rotate : int xPos : int yPos : int

Paddle()
Paddle(int, int)
getBounds()
getMoveLeft()
getMoveRight()
getRotate()
getXPos()
getYPos()
setMoveLeft(boolean)
setMoveRight(boolean)
setRotate(int)
setXPos(int)
setYPos(int)

Player
bounds : Circle dead : boolean isHuman : boolean playerName : String winner : boolean xPos : int yPos : int

Player()
Player(int, int)
getBounds()
getIsHuman()
getPlayerName()
getXPos()
getYPos()
hasWon()
isDead()
kill()
makeWinner()
setIsHuman(boolean)
setPlayerName(String)
setXPos(int)
setYPos(int)

Iceberg
bounds : Rectangle yVelocity : int

Iceberg(int, int, int)
getBounds()
getYVelocity()
setYVelocity(int)

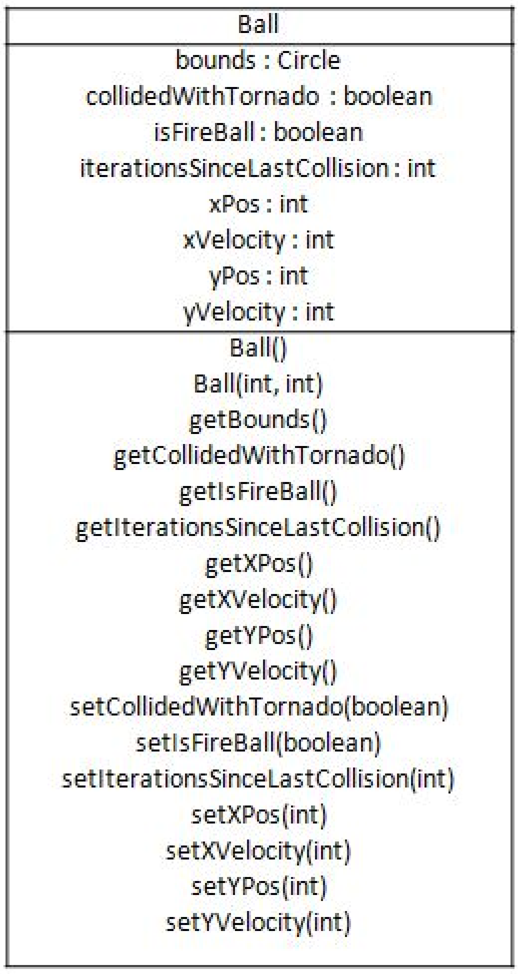
SinkHole
bounds : Rectangle

SinkHole(int, int)
getBounds()

Tornado
bounds : Circle

Tornado(int, int)
getBounds()

Model


Ball
bounds : Circle
collidedWithTornado : boolean
isFireBall : boolean
iterationsSinceLastCollision : int
xPos : int
xVelocity : int
yPos : int
yVelocity : int
Ball()
Ball(int, int)
getBounds()
getCollidedWithTornado()
getIsFireBall()
getIterationsSinceLastCollision()
getXPos()
getXVelocity()
getYPos()
getYVelocity()
setCollidedWithTornado(boolean)
setIsFireBall(boolean)
setIterationsSinceLastCollision(int)
setXPos(int)
setXVelocity(int)
setYPos(int)
setYVelocity(int)