Rebound – A Laser Chess Game

Dec. 3, 2019

Supervisor:
Swapnil Patel (swapnil@usc.edu)

Members (alphabetically):
Amber Guo (amberguo@usc.edu)
Brian Nlong Zhao (briannlz@usc.edu)
Cameron Williams (cqwillia@usc.edu)
Jiefan Yang (jiefanya@usc.edu)
Yiqian Yang (yiqianya@usc.edu)

Revision History

Version	Date Modified	Modified by	Comments
1.0	Oct. 20, 2019	Brian Nlong Zhao Jiefan Yang	High Level Requirements
1.1	Nov. 2, 2019	Brian Nlong Zhao	Generalized some terms
2.0	Nov. 3, 2019	Brian Nlong Zhao	Technical Specifications
3.0	Nov. 11, 2019	Amber Guo Brian Nlong Zhao Cameron Williams Jiefan Yang Yiqian Yang	Detailed Design
3.1	Nov 17, 2019	Brian Nlong Zhao	Changed server and board architecture in detailed design section
3.2	Dec. 3, 2019	Brian Nlong Zhao	Updated pseudo-finalized detailed documentation
4.0	Nov 17, 2019	Yiqian Yang	Testing Documents
5.0	Nov 24, 2019	Brian Nlong Zhao	Deployment

Table of Contents

1 Introduction

1.1 General

1.2 High Level Requirements

1.3 Technical Specifications

1.4 Detailed Design

1.5 Testing

1.6 Deployment

1.7 References

2 High Level Requirements

2.1 Overview

2.1.1 Introduction

2.1.2 Objectives and Goals

2.2 Architecture and Functionalities

2.2.1 Launch Menu

2.2.2 Users

2.2.3 Game Boards

2.2.4 Game Pieces and Movements

2.2.5 Lasers and Interactions with Pieces

2.2.6 Game Flow and Other Rules

2.2.7 Networking

3 Technical Specifications

3.1 Overview

3.2 Baisc Architecture

3.2.1 Basic Hardware Requirements

3.2.2 Baisc Software Requirements

3.3 Client Architecture

3.4 Server Architecture

3.5 Database Architecture

4 Detailed Design

4.1 Overview

4.2 Launching and Authentication

4.2.1 Desktop Launcher

4.2.2 Authentification

4.3 Game Core Architecture

4.3.1 General

4.3.2 LaserGame

4.3.3 GameInputProcessor

4.3.4 AbstractGameBoard

4.3.5 AbstractBoardTile

4.3.6 AbstractGamePiece

4.4 Play Flow and Networking

4.4.1 General

4.4.2 Player

4.4.3 GameSocket

4.4.4 GameRoom

4.5 Theme and Artistic Design

5 Testing

5.1 Launch Menu

5.1.1 Functionality

5.1.2 Login Form

5.2 Game Board

5.2.1 AbstractGameBoard

5.2.2 AbstractBoardTile

5.3 AbstractGamePiece

5.3.1 Individual Piece Moves

5.3.2 Laser Behaviors

5.4 GameServer

6 Deployment

1 Introduction

1.1 General

This document includes the high level requirements, technical specifications, and the detailed design documentation of the game project “Khet”. All of the functions and details are tentative and might be subject to change.

1.2 High Level Requirements

The high level requirements section of this document includes the basic objectives and goals of the game project, as well as the abstract structure and the basic rules of the game. High level requirements section does not include the detailed implementation, and the architecture, functionality and the rules are not finalized and might be changed during development.

1.3 Technical Specifications

The technical specifications section of this document includes the basic routines, methods, and plans about how the technical functionalities of the software should be built and how the high level requirements of this game project can be satisfied. All of the routines and plans are in high-level descriptions and no detailed implementation method is specified in this section.

1.4 Detailed Design

The detailed design section of this document includes the details in the form of the code organizations, inheritance and class structure, and the functions and algorithms that are already determined or implemented, however, everything is still subject to change at any time. A complete java doc documentation will be included in a saparated file.

1.5 Testing

The testing document section of this document includes some test case descriptions on some basic functionalities of the project. Since the development is still in progress, no detailed test case is implemented at this time, and all tests are described in high-level and are subject to change.

1.6 Deployment

The deployment document section of this document includes some plans on how we will promote the finished game project. The project is not intended to be published or promoted, however, we might distribute the game through game digital distribution service platforms in the future. This section will list the steps of distributing this game project through Steamworks Distribution Program provided by Steam.

1.7 References

Khet 2.0 by BlueLine Game Studio
http://bluelinegamestudios.com/khet-game

2 High Level Requirement

2.1 Overview

2.1.1 Introduction

The game project is called "Khet", a chessboard game involving two players playing against each other. In each turn players can move or turn his piece to block or reflect a laser emitted from a source. The goal is to direct the laser to opponent's king by moving and turning your pieces with different functionalities.

2.1.2 Objectives

Our goal is to implement a playable "Khet" game that runs on the desktop and can be played over a local network. It should allow users to:

- Register and login to the game
- Login and play as a guest
- Play with a computer locally
- Play with another user over local network

2.2 Architecture and Functionalities

2.2.1 Launch Menu

The game shows a launch menu once the user start the software. It should first let user login to the game. Options on the first launch menu should include register, login, guest login, and quit. Once logged in, the user can choose to play with a computer locally, play with another player over local network, go to the game setting page, or log out and go back to the first launch menu. The potential option in the setting page includes music and sound volume, resolution, change username, displayed name, and password, theme, etc.

2.2.2 Users

The game involves two users playing on the same game board. The player is either a registered user, a guest user, or a computer. Guest user could only play with a computer, while registered user can choose to play with another registered user online or with a computer.

2.2.3 Game Boards

The game board is like an ordinary chess board, but can have more variations. A board contains square tiles only, and one tile could hold only one piece at a time. The board can be a normal n by n square board, but it also be rectangle, triangle, or any other irregular shapes, as long as all the single tile is a square.

2.2.4 Game Pieces and Movements

The game pieces are classified into several categories. A source piece is the piece where your laser beam is emitted from. Usually it is placed on side or a corner of the board and cannot be moved. A single mirror can reflect the incoming laser at a right angle. A block piece cannot reflect laser. A target piece is like the king in the chess game. Whoever lost the target piece first lose the game. Mirrors and blocks may be divided into subclasses such as single-sided mirror or double-sided mirror. A piece could be turned ninety degrees on each turn, and every movable piece could be moved to one of the eight tiles next to it, but it cannot be moved to a tile which is already occupied or out of the board boundary.

2.2.5 Lasers and Interactions with Pieces

A mirror piece is oriented at a 45 degree difference from the orientation of other pieces and the board. It can reflect any incoming laser at a right angle if the laser hits the mirror side. For single-sided mirror, if either of its back sides is hit by a laser, the piece is then destroyed and should be removed from the board, while a double-sided mirror will never be destroyed. A block piece can block laser that hits it from its front, but will still be destroyed if a laser hits it from the side or back. More piece types such as a mirror that separates one beam into two might be added.

2.2.6 Game Flow and Other Rules

The two players in a game will be labeled in different colors. One player will move first and they take turns to make movements, with each player moving his or her own pieces. Each move involves either rotating a piece or moving a piece to an adjacent tile but not both. Occupied tile is not available. After each move, laser will fire automatically from the source piece of the player moving in that turn. The game will end when one player destroys the opponent’s target piece and that player wins. Or the game will end in a draw if the same board arrangement appears for a third time in the same game.

2.2.7 Networking

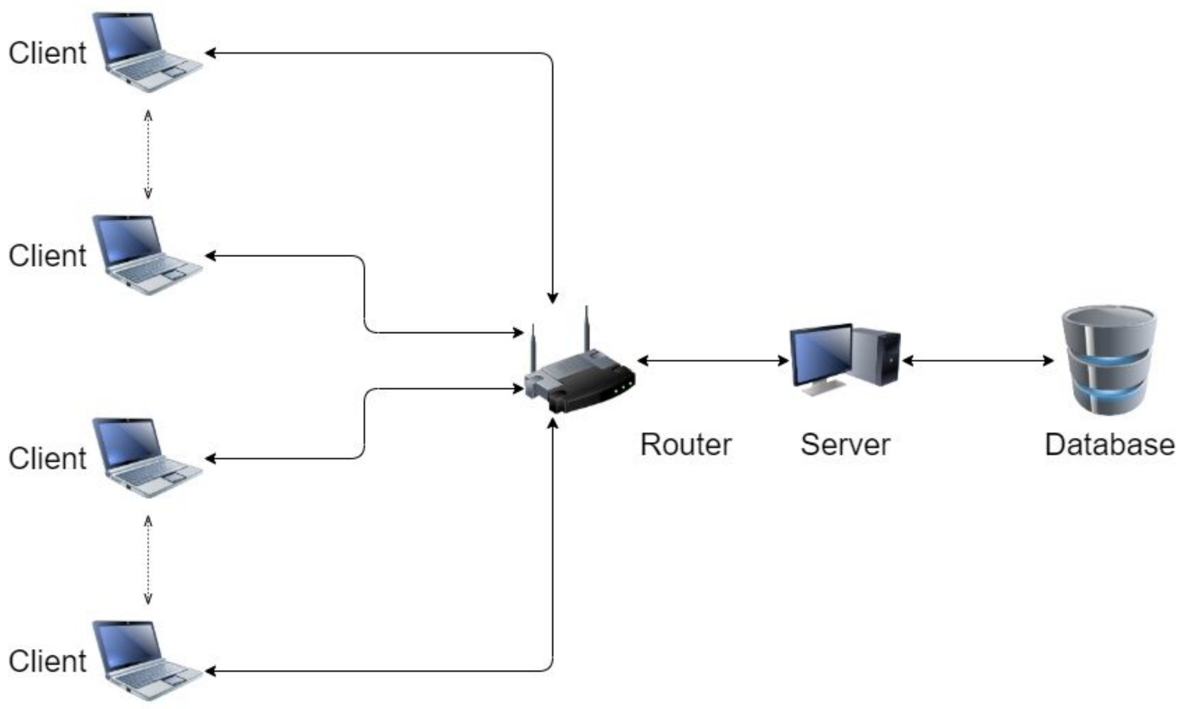
Users can choose to play the game online. Players who choose to play online will be put into a matchmaking queue, and eventually put in a game room with another player. The players will be able to see the live movements of the opponent.

3 Technical Specifications

3.1 Overview

The software is implemented using Java programming language. In addition, the project is mainly built on LibGDX game-development application framework. LibGDX framework helps handle the basics implementative functionalities of the game such as loading assets, processing user input, rendering, and creating graphical user interface. IntelliJ IDEA Ultimate is used for development.

3.2 Basic Architecture



3.2.1 Basic Hardware Requirements

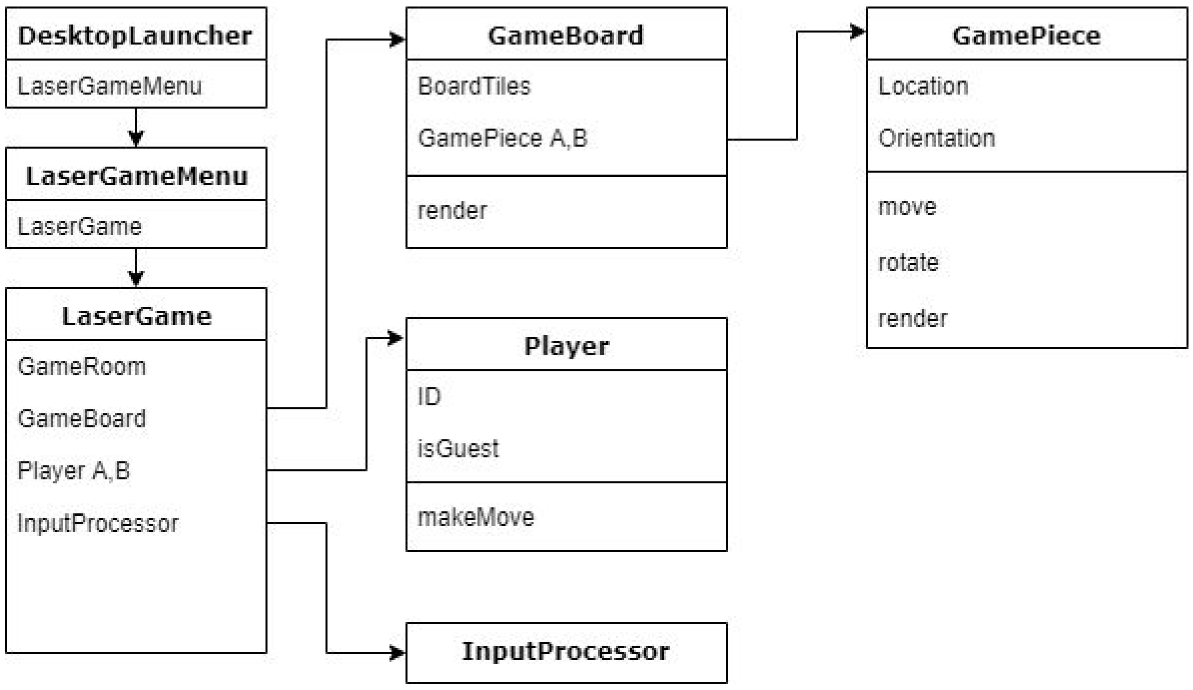
In order to fulfill the networking requirements and authentication functionality, a server side is needed to run all of the games at the same time and communicate with the clients in real time. A router is needed to connect all of the clients and the server together, and the server side needs to access the database, which stores all of the user information.

3.2.2 Basic Software Requirements

The game runs only on desktop. The project will be exported as a runnable jar file, therefore the client should has appropriate operating system to run the game. Windows/Linux/Mac OS X are supported by LibGDX desktop launcher. A PC serves as the server for the game, so the PC should have appropriate environment that allows the server side java code to be run. Since the game runs under local area network, a MySQL instance is run on the server PC as the database that stores the user information.

3.3 Client Architecture

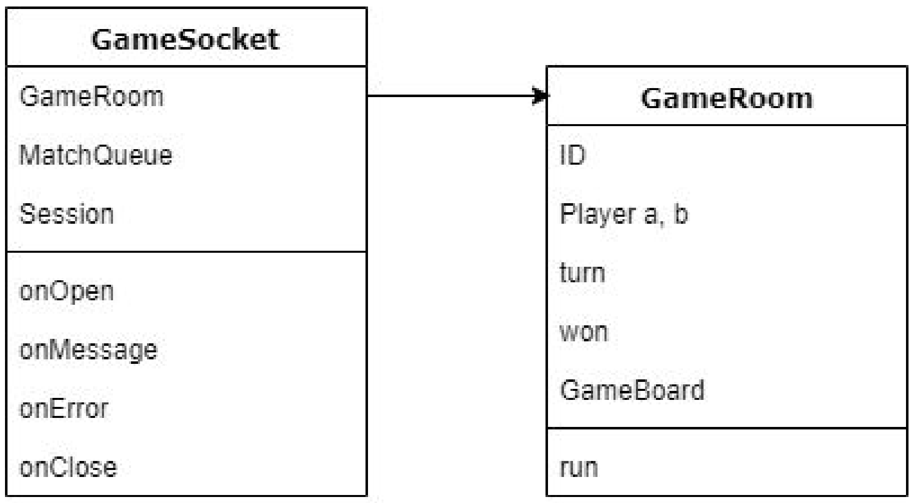
(Estimated time: 40hr)



User interface is launched by the DesktopLauncher. It should creates and show a menu of the game. After logging in, if the user choose to start game, a main class LaserGame should be created. The main class should hold the GameRoom class, which is used on the server side to communicate between clients. A GameBoard class represents the board that the two players are playing on, and it should holds GamePiece class. Players are allowed to makeMove on the pieces, and the piece can move and rotate. Besides, each Player holds an InputProcessor to keep track of keyboard or mouse input, and each game elements (tiles, board, pieces, laser, player info, etc.) should have the corresponding render function for display.

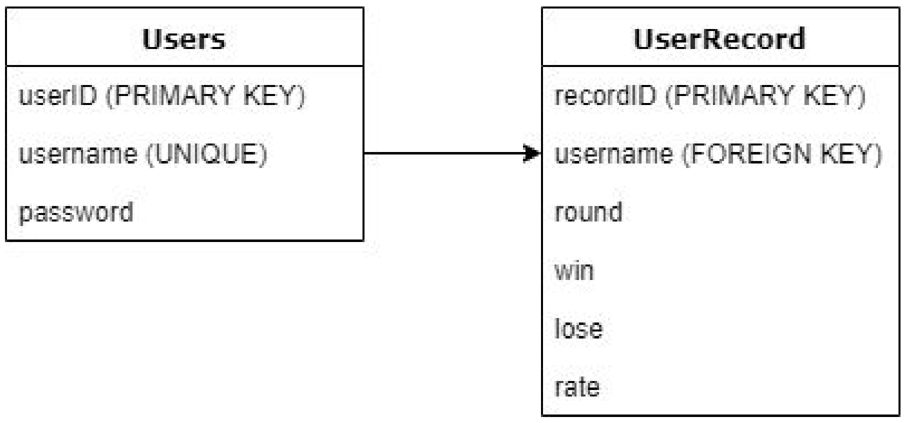
3.4 Server Architecture

(Estimated time: 20hr)



Server runs GameSocket class, which holds all of the GameRooms that are currently in game. The MatchQueue holds all the players that are waiting for matching with another player. GameRoom class holds the general game flow and login in the run function. It keep tracks players' turn and whether the player win the game. These code should be run on the PC that serves as the server for the games.

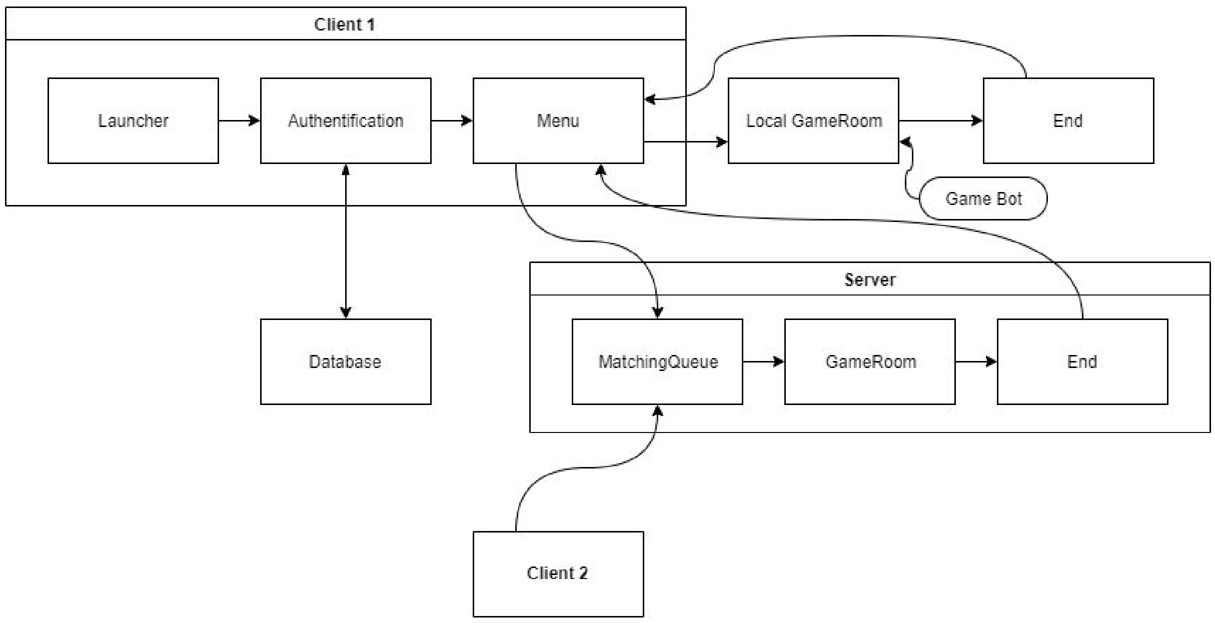
3.5 Database Architecture
(Estimated time: 5hr)



The database handles the login and register functionalities. It also keeps track of player's game record data. MySQL is used and the instance is run on the PC that serves as the server.

4 Detailed Design

4.1 Overview



Java language with LibGDX library will be used as the programming language of this game project, on both client side and server side.

4.2 Launching and Authentication

4.2.1 Desktop Launcher

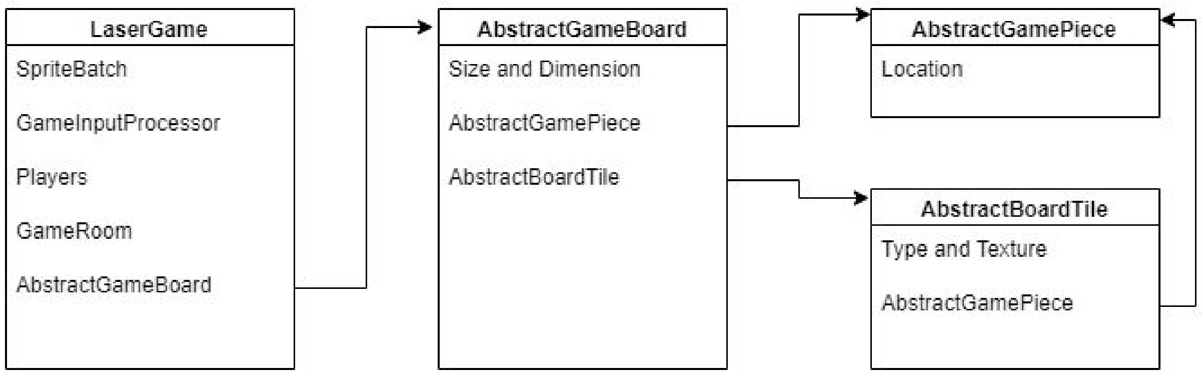
The software is launched by a launcher class named DesktopLauncher. This launcher is provided by the LibGDX library on creating the project. In the class the title, width, and height of the window will be specified, and then it creates the core class of the game, namely LaserGame.

4.2.2 Authentication

A local MySQL instance will be used to store the user informations. The authentication page will connect with the database using a JDBC document.

4.3 Game Core Architecture

4.3.1 General



4.3.2 LaserGame

This class implements the main game loop, and it is instantiated by the DesktopLauncher class.

Data Members

Modifier and Type	Name	Description
`private static com.badlogic.gdx.graphics.g2d.SpriteBatch`	**`batch`**	The `SpriteBatch` object responsible for rendering textures onto the game canvas.
`private` **`AbstractGameBoard`**	**`board`**	The game board that tracks current configuration and player movements.
`private static com.badlogic.gdx.utils.Array<com.badlogic.gdx.utils.Disposable>`	**`disposables`**	Tracks all objects (such as Textures) that implement `Disposable` and need to be freed.
`private com.badlogic.gdx.InputProcessor`	**`gameInputProcessor`**	Input processor that handles user mouse and keyboard actions.
`private` **`GameRoom`**	**`gameRoom`**	GameRoom that includes the two players of the same game.
`private` **`Player`**	**`playerA`**	Player A of the game.
`private` **`Player`**	**`playerB`**	Player B of the game.

Methods

Modifier and Type	Method	Description
`void`	**`create`**`()`	Construct the game with data members.

void	**dispose**()	Dispose unused elements.
static com.badlogic.gdx.graphics.Texture	**loadTexture**(java.lang.String path)	Centralized method for creating textures so they can be freed by dispose().
void	**render**()	Render and display the elements to screen.

4.3.3 GameInputProcessor

This class implements the library-provided interface InputProcessor and catches the user input on mouse and keyboard on an AbstractBoardTile. The following functions are included and some may not be used:

Modifier and Type	Method
boolean	**keyDown**(int keycode)
boolean	**keyTyped**(char character)
boolean	**keyUp**(int keycode)
boolean	**mouseMoved**(int x, int y)
boolean	**scrolled**(int amount)
boolean	**touchDown**(int x, int y, int pointer, int button)
boolean	**touchDragged**(int x, int y, int pointer)
boolean	**touchUp**(int x, int y, int pointer, int button)

4.3.4 AbstractGameBoard

This class is the super class of all kinds of game boards. One existing subclass of this class is StandardBoard, which is a standard 8*8 square board. DIfferent dimensions and shapes could be applied to the subclasses.

Data Members

Modifier and Type	Name	Description
protected **LaserPiece**	**aLaser**	The LaserPiece of player A.
protected **KingPiece**	**aPharaoh**	The KingPiece of player A.
protected **LaserPiece**	**bLaser**	The LaserPiece of player B.

private static **AbstractGameBoard**	**board**	Tracks the game board currently being played on for centralized modification by various game objects.
protected **KingPiece**	**bPharaoh**	The KingPiece of player B.
boolean	**flipBoard**	Variable that keeps track of whether the board is flipped
private boolean	**hasTurn**	Indicates which player has the turn.
private com.badlogic.gdx.graphics.Texture	**highlightTexture**	Texture of the highlighted part of laser.
private com.badlogic.gdx.graphics.Texture	**horizontalLaserTexture**	Texture of the horizontal laser.
boolean	**isOver**	Variable that keeps track on whether the game is over
private com.badlogic.gdx.audio.Music	**kingDestroyedSound**	The sound effect when a king is destroyed
private long	**laserDuration**	The duration time of laser on the board.
private com.badlogic.gdx.audio.Music	**laserSound**	The laser sound for firing a laser
private com.badlogic.gdx.utils.Array<com.badlogic.gdx.math.Rectangle>	**lasersToDraw**	The laser that is to be drawn.
private boolean	**local**	Determines if this game board describes a local game.
private boolean	**moveConfirmed**	Variable that keeps track of whether the move is confirmed
private **GameMessage**	**nextMove**	The next move to be sent to the server
private **AbstractGamePiece**	**pickedUpPiece**	Variable that stores the piece that is currently picked up
private com.badlogic.gdx.audio.Music	**pieceDestroyedSound**	The sound when a piece is destroyed

`private int`	**`pieceDim`**	Dimension of each piece
`protected com.badlogic.gdx.utils.Array<`**`AbstractGamePiece`**`>`	**`pieces`**	Array that includes all of the pieces on th board.
`private com.badlogic.gdx.audio.Music`	**`pieceSound`**	The click sound for picking and dropping a piece
`private int`	**`screenX`**	X-display of the screen.
`private int`	**`screenY`**	Y-display of the screen.
`private int`	**`tileDim`**	Dimension of each tile.
`protected com.badlogic.gdx.utils.Array<`**`AbstractBoardTile`**`>`	**`tiles`**	Array that includes all the tiles of the board.
`private com.badlogic.gdx.graphics.Texture`	**`verticalLaserTexture`**	Texture of the vertical laser.
`protected int`	**`x`**	X-dimension of the board.
`protected int`	**`y`**	Y-dimension of the board.

Methods

Modifier and Type	Method	Description
`private boolean`	**`canPickUpPiece`**`(`**`AbstractGamePiece`**` piece)`	Checks whether a piece could be picked up based on the turn and the owner of the piece
`private static boolean`	**`checkClickBounds`**`(int oldX, int oldY, int newX, int newY)`	Checks if a pair of click screen coordinates are legitimate (within bounds and corresponding to the same tile)
`abstract void`	**`createPieces`**`()`	Abstract method which places pieces on the board.
`abstract void`	**`createTiles`**`()`	Abstract method which populates the board with tiles based on the board type.
`private void`	**`drawLaser`**`(int startX, int startY, `**`Directions.Direction`**` d)`	Helper function for `fireLaser()`

void	**fireLaser**(int startX, int startY, **Directions.Direction** d)	Handles the logic and rendering of a laser being fired into a tile.
LaserPiece	**getActiveLaser**()	Getter of laser piece according to hasTurn.
LaserPiece	**getALaser**()	Getter of aLaser piece.
LaserPiece	**getBLaser**()	Getter of bLaser piece.
GameMessage	**getNextMove**()	Get the confirmed next move.
AbstractGamePiece	**getPieceFromCoordinate**(int x, int y)	Converts a coordinate to the corresponding piece on the current game board.
com.badlogic.gdx.utils.Array<**AbstractGamePiece**>	**getPieces**()	Getter of the pieces array.
AbstractBoardTile	**getTileFromCoordinate**(int x, int y)	Converts a coordinate to the corresponding tile on the current game board.
private static **AbstractBoardTile**	**getTileFromLocation**(int mouseX, int mouseY)	Converts a location on the virtual screen to the corresponding tile on the current game board.
com.badlogic.gdx.utils.Array<**AbstractBoardTile**>	**getTiles**()	Getter of the tiles array.
int	**getX**()	Getter of the x-dimension of the board.
int	**getY**()	Getter of the y-dimension of the board.
private boolean	**handleLaserClick**(**LaserPiece** laser)	Function dealing with clicking on the laser piece
private boolean	**handleLaserRotate**(**LaserPiece** laser)	Function dealing with rotation of the laser source.
void	**initialize**()	Initializes the client side fields of the game board (screen dimension, static board variable)
abstract java.lang.String	**isGameOver**()	Abstract method which returns whether game is over based on board type.

boolean	**isValidMove**(boolean color, int x, int y, java.lang.String moveType, int nX, int nY)	Check whether the movement of a player on a piece is valid.
static boolean	**keyPressed**(java.lang.String key)	Processes a key press on the current game board.
private void	**laserHelper**(int startX, int startY, **Directions.Direction** d)	A recursive fucntion for drawing the laser onto the board.
static boolean	**leftClick**(int oldX, int oldY, int newX, int newY)	Processes a left click on the current game board at the specified screen location.
private boolean	**makeMove**(**AbstractBoardTile** tile)	Attempts to make a piece move with the currently picked up piece onto the given tile.
private boolean	**pieceMove**(**AbstractGamePiece** piece, int x, int y)	Move a selected piece on the board.
private boolean	**pieceRotateLeft**(**AbstractGamePiece** piece)	Left-rotate a selected piece on the board.
private boolean	**pieceRotateRight**(**AbstractGamePiece** piece)	Right-rotate a selected piece on the board.
void	**render**(com.badlogic.gdx.graphics.g2d.SpriteBatch sb)	Renders the current game board.
static boolean	**rightClick**(int oldX, int oldY, int newX, int newY)	Processes a right click on the current game board at the specified screen location.
private void	**undoMove**()	Undo the move that is not confirmed by the player.
void	**update**(int x, int y, java.lang.String moveType, int nX, int nY)	Update the board configuration

4.3.5 AbstractBoardTile

This is an abstract superclass describing the behaviour of tiles on the game board. Determined subclasses include BlankTile, an available spot with no piece on it; LaserTile, a tile with laser shooting across it; StandardTile, The normal blank tile with no added effects. More subclass may be implemented.

Data Members

Modifier and Type	Name	Description
`private static int`	**`IMG_DIM`**	Dimension of each tile.
`private` **`AbstractGamePiece`**	**`piece`**	The piece that sits on the tile.
`static` **`AbstractBoardTile`**	**`ROTATE_LEFT`**	A new tile instance for left rotation.
`static` **`AbstractBoardTile`**	**`ROTATE_RIGHT`**	A new tile instance for right rotation.
`private static long`	**`serialVersionUID`**	Serial ID
`private com.badlogic.gdx.graphics.Texture`	**`texture`**	The texture of the board tile.

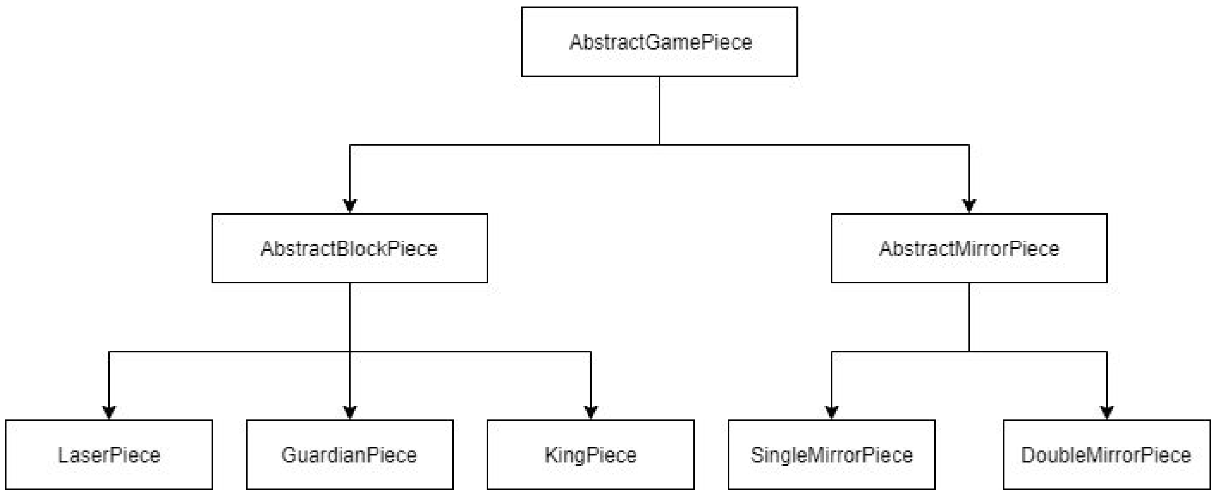
Methods

Modifier and Type	Method	Description
`private java.lang.String`	**`getPathFromTileType`**`(`**`AbstractBoardTile.TileType`**` type)`	Utility function which converts a tile type into a file path.
`AbstractGamePiece`	**`getPiece`**`()`	Getter of the game piece
`void`	**`loadRegion`**`()`	
`protected void`	**`loadRegion`**`(`**`AbstractBoardTile.TileType`**` type)`	Load the texture on the tile using the type of image.
`protected void`	**`loadRegion`**`(java.lang.String image)`	Load the texture on the tile using the path of image.
`protected void`	**`loadRegion`**`(java.lang.String basePath,` **`AbstractBoardTile.TileType`**` type)`	Load the texture on the tile using a base path and the type of tile.
`void`	**`onLeftClick`**`()`	Trigger for when a left mouse button is clicked and released within this tile.

void	**onPieceDestroyed**(**AbstractGamePiece** piece)	Trigger for when the game piece is removed from this tile.
void	**onPiecePlaced**(**AbstractGamePiece** piece)	Trigger for when a game piece is placed on this tile.
void	**onPieceRotated**(**AbstractGamePiece** piece)	Trigger for when the game piece placed on this tile is rotated.
void	**onRightClick**()	Trigger for when a right mouse button is clicked and released within this tile.
void	**render**(com.badlogic.gdx.graphics.g2d.SpriteBatch sb, int x, int y, int width, int height)	Render the board
void	**render**(com.badlogic.gdx.graphics.g2d.SpriteBatch sb, int x, int y, int width, int height, boolean flipped)	Render the tile, using the draw method of SpriteBatch class.
void	**setPiece**(**AbstractGamePiece** p)	Replace the current piece on this tile by a different piece.

4.3.6 AbstractGamePiece

This class is an abstract superclass describing the behaviour of a game piece. It has two subclasses: AbstractBlockPiece and AbstracMirrorPiece. AbstractBlockPiece is the superclass for all the pieces that has 90 degree orientations, while AbstractMirrorPiece is a superclass for all the mirror pieces that has 45 degree orientations. Below is a class inheritance diagram for the game piece classes. More subclasses may be added under AbstractBlockPiece and AbstractMirrorPiece.



Data Members

Modifier and Type	Name	Description
`protected boolean`	**`color`**	The color of the piece.
`private static int`	**`IMG_DIM`**	The dimension of the piece
`protected com.badlogic.gdx.graphics.Texture[]`	**`textures`**	The texture of the piece of different orientations.
`protected int`	**`x`**	The location of the piece on the board.
`protected int`	**`y`**	The location of the piece on the board.

Methods

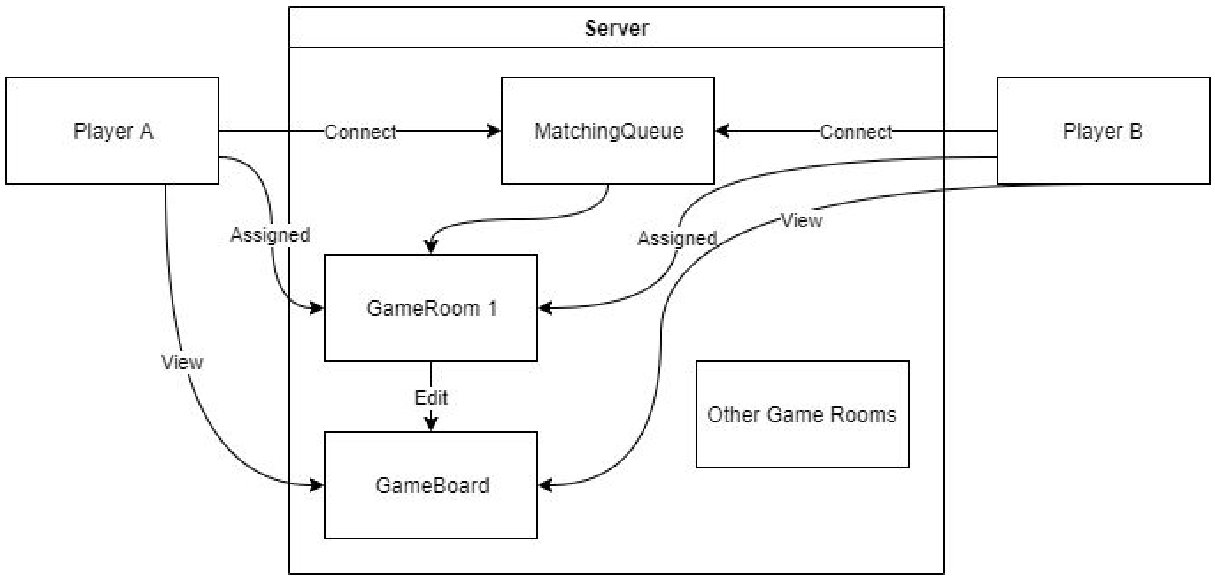
Modifier and Typer	Method	Description
`abstract com.badlogic.gdx.utils.Array<`**`Directions.Direction`**`>`	**`acceptLaser`**`(`**`Directions.Direction`**` laserDirection)`	Defines the behaviour of a laser when it encounters this game piece.
`protected abstract void`	**`flipOrientation`**`()`	Abstract function of flipping the orientation of the piece.
`boolean`	**`getColor`**`()`	
`com.badlogic.gdx.utils.Array<`**`AbstractBoardTile`**`>`	**`getLegalMoves`**`(`**`AbstractGameBoard`**` b)`	Get the legal tiles that this piece can be moved to.

`protected abstract com.badlogic.gdx.graphics.Texture`	**`getTexture`**`()`	Abstract function of getting the texture of the specific piece.
`int`	**`getX`**`()`	Getter of the x location of the piece.
`int`	**`getY`**`()`	Getter of the y location of the piece.
`abstract void`	**`loadRegion`**`()`	Abstract function of loading the region of the specific piece.
`AbstractGamePiece`	**`pickUpPiece`**`(`**`AbstractGameBoard`**` b)`	Defines the behaviour of a game piece when it is picked up
`void`	**`placePiece`**`(`**`AbstractGameBoard`**` b)`	Defines the behaviour of a game piece when it is placed on the board
`void`	**`render`**`(com.badlogic.gdx.graphics.g2d.SpriteBatch sb, int x, int y, int width, int height)`	Render the piece above the tile.
`void`	**`render`**`(com.badlogic.gdx.graphics.g2d.SpriteBatch sb, int x, int y, int width, int height, boolean flipped)`	Render the flipped piece
`void`	**`render`**`(com.badlogic.gdx.graphics.g2d.SpriteBatch sb, int x, int y, int width, int height, boolean flipped, boolean pickedUp)`	Render the picked up piece and flipped piece
`abstract void`	**`rotateLeft`**`()`	Transforms the orientation of this piece appropriately for one quarter turn counterclockwise.
`abstract void`	**`rotateRight`**`()`	Transforms the orientation of this piece appropriately for one quarter turn clockwise.
`void`	**`setX`**`(int x)`	Modifier of the x location of the piece.

void	**setY**(int y)	Modifier of the y location of the piece.

4.4 Play Flow and Networking

4.4.1 General



4.4.2 Player

Data Members

Modifier and Type	Name	Description
private **AbstractGameBoard**	**board**	A reference to the board that the player is playing
private **ClientThread**	**ct**	A reference to the client thread tah holds this player
private boolean	**lastGame**	
private int	**numLoss**	A record of ho many losses this player has
private int	**numPlayed**	A record of how many games this player has player
private int	**numWin**	A record of how many wins this player has
private java.io.ObjectOutputStream	**out**	The output stream that send game messages to the server
java.lang.String	**playerID**	The id of the player

private java.net.Socket	**socket**	The sockect of the client side

Methods

Modifier and Type	Method	Description
AbstractGameBoard	**getBoard**()	Getter of the board of this player.
boolean	**getLastGame**()	Get the last game result of this player
int	**getNumLoss**()	Getter of the numLoss.
int	**getNumPlayed**()	Getter of the numPlayer.
int	**getNumWin**()	Getter of the numWin.
java.lang.String	**getPlayerID**()	Getter of the id of this player.
boolean	**login**(java.lang.String playerID, java.lang.String pass)	Login the this user
void	**lost**()	This player lose a game
boolean	**register**(java.lang.String playerID, java.lang.String pass)	Register this player to the databse.
void	**sendMatchmakingRequest**()	Send a request for match making to the server.
void	**sendMessage**(**GameMessage** message)	Send a game message to the output stream.
void	**setBoard**(**AbstractGameBoard** board)	Setter of the board reference of this player.
void	**setPlayerID**(java.lang.String playerID)	Setter of the id of this player.
void	**updateRecord**(int numPlayed, int numWin, int numLoss)	Setter of the player status.
void	**won**()	This player win a game.

4.4.3 GameServer

This class is the server side of websocket. It hosts all GameRooms and implements matchmaking queue. This class should be in a separate server project instead of this local game project (client).

Data Members

Modifier and Type	Name	Description
`private static java.sql.Connection`	`conn`	Database connection
`private static java.util.Vector<GameRoom>`	`gameRooms`	stores mapping from playerID to GameRooms
`private static java.lang.String`	`host`	Hostname of the server
`private static java.util.Vector<ServerThread>`	`loggedInQueue`	Stores ServerThreads that are logged in
`private static java.util.Vector<ServerThread>`	`loginQueue`	Stores ServerThreads that are not logged in
`private static java.util.Vector<ServerThread>`	`matchingQueue`	Stores all player sockets in matchmaking
`static int`	`port`	Port number of the server
`private static java.lang.String`	`pwd`	Database password
`private static java.lang.String`	`url`	Database connection URL
`private static java.lang.String`	`user`	Database username

Methods

Modifier and Type	Method	Description
`void`	`addToMatchmaking(java.lang.String playerID)`	Add the logged in user to the matchmaking queue
`boolean`	`checkConnection()`	Checks if the server is successfully connected to the database.
`void`	`deleteRoom(GameRoom gr)`	Remove the game room from the gameRooms vector.

static java.sql.Connection	getConnection()	Get a connection to the database.
void	loginServerThread(ServerThread serverThread)	Remove the ServerThread from loginQueue and add that ServerThread to loggedInQueue.
void	logMessage(GameMessage message)	Print the game message to the server console.
static void	main(java.lang.String[] args)	
GameMessage	queryDatabase(GameMessage gm)	Query/update the database for login, register, stats_request
boolean	updateDatabase(GameMessage gm)	Update records for both players after one game is over.

4.4.4 GameRoom

The GameRoom class implements the basic game flow. This class holds the threads of two players in a same game and should be run on the server side under GameServer The flow is implemented in the run() function. Current version only supports local game.

Data Members

Modifier and Type	Name	Description
protected ServerThread	aThread	Thread of player a on the server
protected AbstractGameBoard	board	The game board that the two players are playing with
protected ServerThread	bThread	Thread of player b on the server
private GameServer	gs	A reference of the game server
boolean	isOver	Variable that keeps track of whether the game is over
static int	PORT	Port used for connection
protected boolean	turn	Variable that keeps track of player's turn

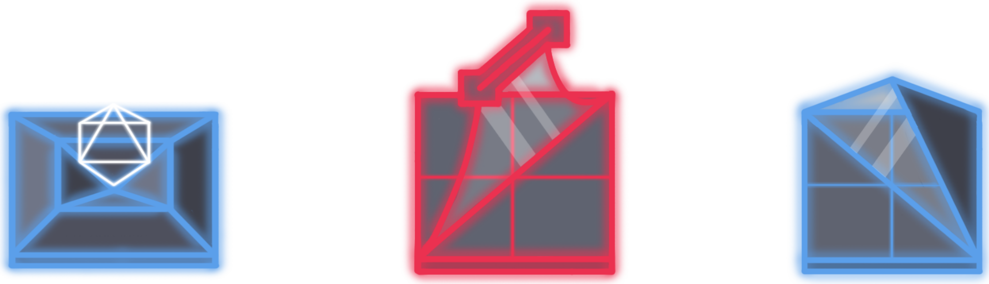
Methods

Modifier and Type	Method	Description

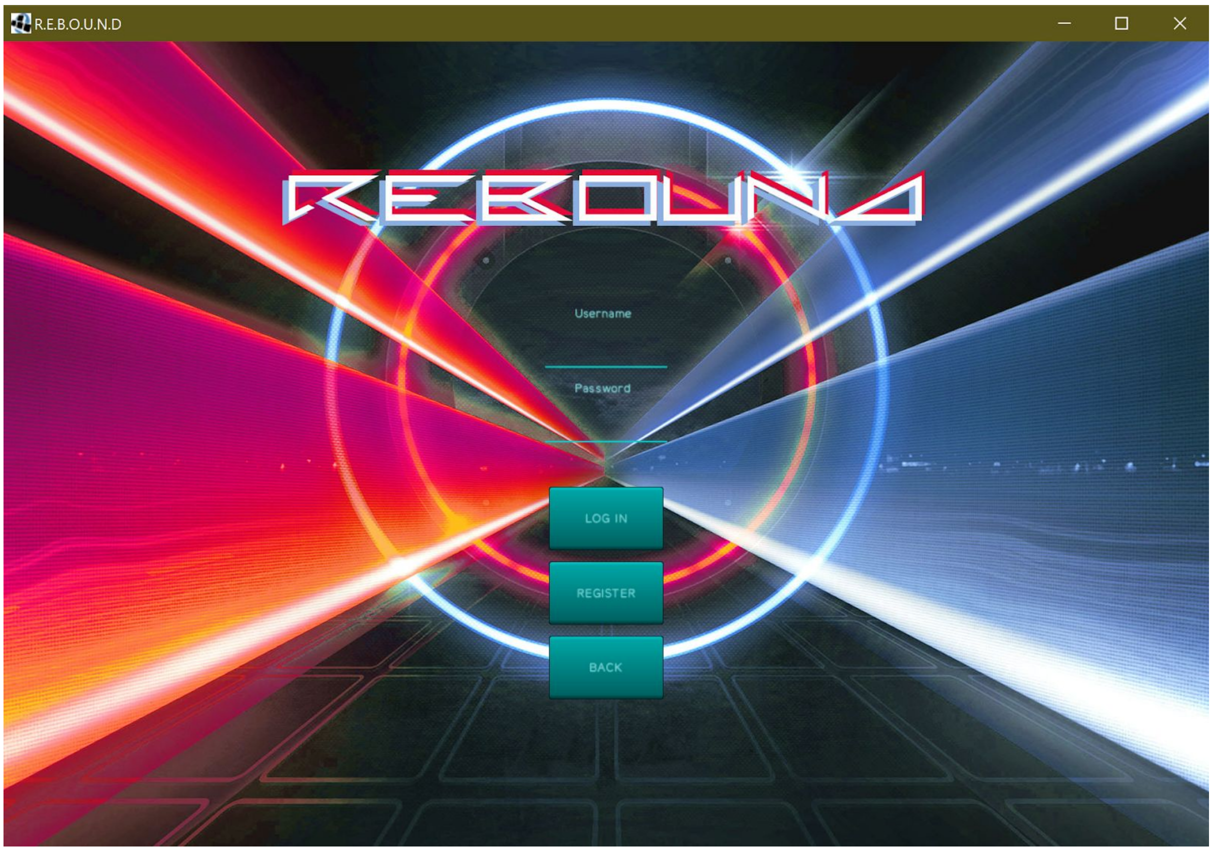
void	**broadcast**(**GameMessage** message)	Broadcast the message to both two clients.
void	**clear**()	Delete the game room.
void	**disconnect**(**ServerThread** st)	Disconnect the server threads after the game is over
void	**endGame**(java.lang.String res)	End the current game.
private java.lang.String	**getActivePlayerID**()	Get the player ID of the active player.
void	**handleMoveAttempt**(**GameMessage** move)	Validate move, send move success/failure message to server.
void	**readMessage**(**GameMessage** message)	Read the message from the server.
private void	**updateBoard**(int x, int y, java.lang.String moveType, int nX, int nY)	Update the board on the server

4.5 Theme, Music, and Artistic Design

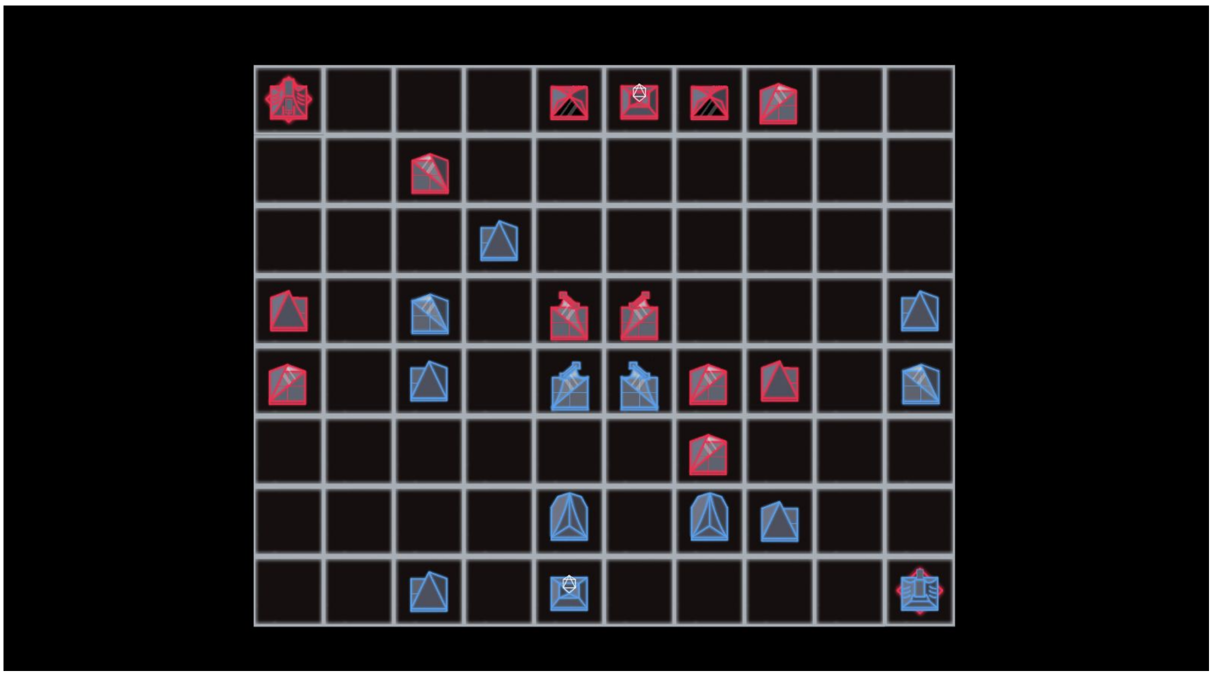
The theme of this game is laser and neon light. A sample game piece and the sample main menu layout is shown below. Other details are to be decided. Music and other details will be finalized after basic functional implementations.



Sample Game Pieces



Sample Start Menu



Sample Game Board

5 Testing

5.1 Launch Menu

5.1.1 Functionality

- The launch menu page shall allow the user to Register/Login as a member, Login as a guest, Update Settings, as well as Quit the game.
- It will ask users to Login through the Login Form or Register through the Register Form.
- If the user has an account, they can log in through the form.
- If the user does not have an account, they can register to become a member, or log in as a guest.
- By clicking on Register, the form will reload to the Registration Form.
- By clicking on Guest, the user can play as a guest, no Login/Register is needed. But if continue to play as a Guest, the user can only play local game(ie. play with the computer).

5.1.2 Login Form

- Invalid Case: If username and password do not match: display *Invalid Username or Password* above the login button.
- If username and password do match, the user should get an alert message.
- Pressing the login button with empty input fields causes the invalid case

5.1.3 Registration Form

- If username is taken display: display “*This username is unavailable”*
- If password and confirmPassword do not match display: display “*Passwords do not match”*

5.2 Game Board

5.2.1 AbstractGameBoard

- isValidMove() should check whether the player’s move is legal or not. Legal moves includes: rotate a piece or move a piece in one of the four directions. If the player makes an illegal move, this function should return false, and an error message of “Not a valid move” should be displayed to the player and the player must select a legal move again.
- canPickUpPiece() should determine whether the user can pick up a piece or not.

If the game is local and piece.getColor() == this->hasTurn (return value true indicates that the white player has turn, false indicates that the black player has turn), then the piece can be picked up by the player who owns it.
Otherwise the piece can only be picked up if this->flipboard returns false and this->hasTurn returns true.

- makeMove() should check if a piece has been picked up by checking this->pickedUpPiece and if the move is legal by piece->getLegalMoves(). If both conditions are satisfied, then execute the move. The player cannot pick up a piece that does not belong to him/her.
  If the player tries to make a move but not picking up a piece first, an error message of "Please pick up a piece first" should be displayed and the player must pick up a piece again.
  If the player picked up a piece, but tries to make an illegal move (piece->getLegalMoves() does not contain the tile which the player tries to move to), then piece will set to null (piece will be dropped).
- isGameOver() should return the correct winner of the game, or "No win" if there is no winner.
- render(com.badlogic.gdx.graphics.g2d.SpriteBatch sb) should render correct game board to the screen.

5.2.2 AbstractBoardTile

- onLeftClick(), onRightClick(), onPieceRotated(AbstractGamePiece piece), onPiecePlaced(AbstractGamePiece piece), onPieceDestroyed(AbstractGamePiece piece) should be correctly triggered.
- render(com.badlogic.gdx.graphics.g2d.SpriteBatch sb, int x, int y, int width, int height) should render correct game pieces to the screen.

5.3 AbstractGamePiece

5.3.1 Individual Piece Moves

- SingleMirrorPiece, DoubleMirrorPiece correctly reflects a laser to the correct direction, or reflects nothing if the laser does not come towards mirror's orientation.
- GuardianPiece, KingPiece, and LaserPiece implements correct behaviors based on its type. LaserPiece cannot be destroyed; GuardianPiece can only be destroyed if the incoming laser's direction does not come towards GuardianPiece's orientation, otherwise set

GuardianPiece's status to "DEAD" and remove it from the board; the KingPiece can be destroyed from every side, if it is destroyed, then set its status to "DEAD" and isGameOver() should return the correct winner.

5.3.2 Laser Behaviors

- acceptLaser(Directions.Direction laserDirection) should return correct outgoing laser's direction. If the laser is not reflected(it destroyed a piece or hit on non-reflective side of MirrorPieces or hit the orientation of the GuardianPiece), acceptLaser() should return null and the laser should be stopped.

5.4 Game Server

- Logged-in users should be added to loginQueue ordered by time they logged in. Two users will be paired up in the same GameRoom. Users are paired up using First-come, First-Serve principle.
- After two users being paired up in the same Game room, they should be deleted from the loginQueue, and their game play should not affect or be affected by other users in other GameRoom or in loginQueue.
- If failed to connect to the server, an error message of "Unable to connect to the server" should be displayed to the user.

6 Deployment

1. Rent a server (e.g. Google Cloud) and push the server-side documents to the server.
2. Setup the database (e.g. Google Cloud SQL, MySQL) and other configurations.
3. Setup a Steamworks account, download and setup the Steamworks SDK and other features. (e.g. pricing, promotion, etc.)
4. Setup the Steam Depot and other downloading and packet configurations.
5. Export the client side code into a runnable program.
6. Use SteamPipe Content System to build the program into a Steam compatible file and upload the game to the Steam Depot in the Steamworks account.
7. Configure other settings (version, system requirements, etc.) as needed and release the game.