



RAILROADINK

Bojie JIA and Harriet PHILLIPS

u6730978, u6698981



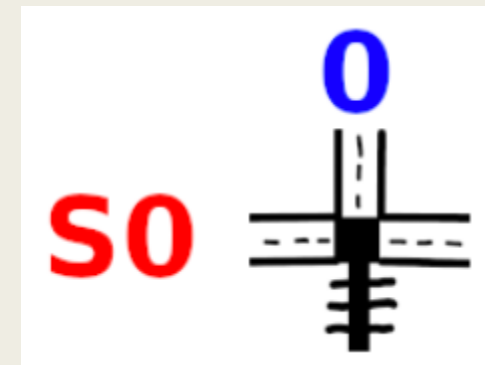
- 1. *Enum Tile*
- 2. *isValidPlacementSequence*
- 3. *generateMove*
- 4. *getBasicScore*
- 5. *getAdvancedScore*
- 6. *Design Approach*
- 7. *Interesting Features*
- 8. *Game Operation*
- 9. *Start Screen*
- 10. *Rules*
- 11. *Single Player*
- 12. *Multi-player*
- 13. *Pseudo Code*



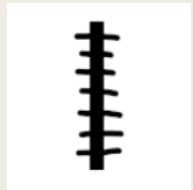
CONTEXT

Enum Tile (authored by Bojie Jia)

- Redefine the tiles with for numbers.
- The number represent the type of road on each side in a clockwise direction.
- “0” means no way.
- “1” means highway.
- “2” means railway.
- Ex: $S00(1, 1, 2, 1)$ means the situation of the **special tile S0** with the **orientation 0**.
the northern, eastern and western way are highways.
the southern way is railway.



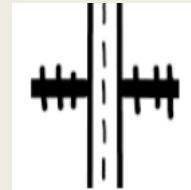
Enum Tile more examples



$A10(2,0,2,0)$



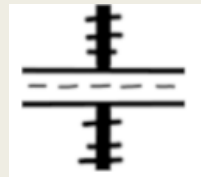
$A36(1,1,1,0)$



$B22(1,2,1,2)$



$A33(1,1,0,1)$



$B27(2,1,2,1)$



$S45(2,1,1,2)$

Enum Tile

partial code

```
public enum Tile {  
  
    S00( north: 1, east: 1, south: 2, west: 1),  
    S01( north: 1, east: 1, south: 1, west: 2),  
    S02( north: 2, east: 1, south: 1, west: 1),  
    S03( north: 1, east: 2, south: 1, west: 1),  
    S04( north: 1, east: 1, south: 2, west: 1),  
    S05( north: 1, east: 1, south: 1, west: 2),  
    S06( north: 2, east: 1, south: 1, west: 1),  
    S07( north: 1, east: 2, south: 1, west: 1),  
  
    S10( north: 1, east: 2, south: 2, west: 2),  
    S11( north: 2, east: 1, south: 2, west: 2),  
    S12( north: 2, east: 2, south: 1, west: 2),  
    S13( north: 2, east: 2, south: 2, west: 1),  
    S14( north: 1, east: 2, south: 2, west: 2),  
    S15( north: 2, east: 1, south: 2, west: 2),  
    S16( north: 2, east: 2, south: 1, west: 2),  
    S17( north: 2, east: 2, south: 2, west: 1),  
  
    S20( north: 1, east: 1, south: 1, west: 1),  
    S21( north: 1, east: 1, south: 1, west: 1),  
    S22( north: 1, east: 1, south: 1, west: 1),  
    S23( north: 1, east: 1, south: 1, west: 1),  
    S24( north: 1, east: 1, south: 1, west: 1),  
    S25( north: 1, east: 1, south: 1, west: 1),  
    S26( north: 1, east: 1, south: 1, west: 1),  
    S27( north: 1, east: 1, south: 1, west: 1),  
  
    S30( north: 2, east: 2, south: 2, west: 2),  
    S31( north: 2, east: 2, south: 2, west: 2),  
    S32( north: 2, east: 2, south: 2, west: 2),  
    S33( north: 2, east: 2, south: 2, west: 2),  
    S34( north: 2, east: 2, south: 2, west: 2),  
    S35( north: 2, east: 2, south: 2, west: 2),  
    S36( north: 2, east: 2, south: 2, west: 2),  
    S37( north: 2, east: 2, south: 2, west: 2),  
}
```

isValidPlacementSequence

(authored by Bojie Jia)

Three methods used in this task

1. notCover:

Check whether all the tiles are not coincident.

2. isExit:

Check whether all the connections to exits are legal.

3. isNeighbor:

Check whether the connections between adjacent tiles are legal.

isValidPlacementSequence

notCover

- *Purpose: check whether all the tiles are not coincident.*

if they are, return true.

- *Method: 1. traversing all the tiles which the length is 5.*

2. check if their 3rd and 4th characters are consistent, if they are, return false.

3. if all the tiles are not coincident, return true.

```
public static boolean notCover(String boardString) {  
    for (int i = 0; i < boardString.length(); i += 5) {  
        for (int j = 0; j < boardString.length(); j += 5) {  
            String s1 = boardString.substring(i, i + 5);  
            String s2 = boardString.substring(j, j + 5);  
            // trasversing all the tiles  
            if (i != j && s1.charAt(2) == s2.charAt(2) && s1.charAt(3) == s2.charAt(3)) {  
                return false;  
            }  
            //check if their 3rd and 4th characters are consistent  
        }  
    }  
    return true;  
}
```

isValidPlacementSequence

isExit

- *Purpose: check whether all the connections to exits are legal, if they are, return true.*
- *Method: 1. Set a 'flag' haveExit and initialize it to false.*
2. Seek the tiles which located at 'A1, A3, A5, B0, B6, D0, D6, F0, F6, G1, G3, G5'. Check if the tile type match the exit.

Ex: For A1, check if the northern number of tile which 3rd and 4th characters are 'A1' is '1' (According to Enum Tile).

-		-		-		-		-	
-	exit	-	h	-	r	-	h	-	exit
-		-		-		-		-	
A	---	A0	A1	A2	A3	A4	A5	A6	---

```
for (int i = 0; i < boardString.length(); i += 5) {  
    String s = boardString.substring(i, i + 5);  
    String t = s.substring(0, 2) + s.charAt(4);  
    Tile tile = Tile.valueOf(t);  
  
    if (s.charAt(2) == 'A' && (s.charAt(3) == '1' || s.charAt(3) == '5')) {  
        if (tile.north == 1) {  
            haveExit = true;  
        } else if (tile.north == 2) {  
            return false;  
        }  
    }  
}
```


isValidPlacementSequence

isExit

- *Method: 3. In the following 'if' statements, if the connections to exits are legal, assign true to haveExit.
If not, return false.
4. return haveExit.*

IMPORTANT: *If do not set haveExit as a flag, the methods would return true even if there is only one legal connections to exits and the others are illegal.*

isValidPlacementSequence *isNeighbor*

- *Purpose: check whether the connections between adjacent tiles are legal.
if they are, return true.*
- *Method: 1. traversing all the tiles in the boardString,
2. Seek adjacent tiles.
3. check if the tile types in the four directions (northern, eastern, southern, western) match. If they are, return true.
4. check if any tile have neighbour.*

```
for (int i = 0; i < boardString.length(); i += 5) {  
    String s = boardString.substring(i, i + 5);  
  
    if (!connect[i] && !isExit(s)) {  
        return false;  
    }  
}  
} //check if any tile have neighbor  
return true;
```

isValidPlacementSequence

isNeighbor one example Codes

```
for (int i = 0; i < boardString.length(); i += 5) {
    for (int j = 0; j < boardString.length(); j += 5) {
        String s1 = boardString.substring(i, i + 5);
        String s2 = boardString.substring(j, j + 5);
        String t1 = s1.substring(0, 2) + s1.charAt(4);
        String t2 = s2.substring(0, 2) + s2.charAt(4);
        Tile tile1 = Tile.valueOf(t1);
        Tile tile2 = Tile.valueOf(t2);
        char row1 = s1.charAt(2);
        char row2 = s2.charAt(2);
        char column1 = s1.charAt(3);
        char column2 = s2.charAt(3);
        if (row1 == row2 && (column1 - column2) == 1) {
            if (tile1.west != 0 && tile1.west == tile2.east) {
                connect[i] = true;
            }
            if (tile1.west != 0 && tile2.east != 0 && tile1.west != tile2.east) {
                return false;
            }
        }
    }
}
```

isValidPlacementSequence

isNeighbor

- **IMPORTANT:** same as *isExit*, it needs a boolean array `connect[]` as 'flag', if not, the methods would return true even if there is only one legal connections to exits and the others are illegal.

isValidPlacementSequence

Return true when all the three methods are true.

```
public static boolean isValidPlacementSequence(String boardString) {  
    if (notCover(boardString) && isExit(boardString) && isNeighbor(boardString))  
        return true;  
    }  
    return false;  
}
```

generateMove

- 1. consider the number of special tile (Actually, there is no corresponding test data about special tile in T10 GenerateMoveTest)

```
for(int s=0;s<specialTile.length&&specialNumber<3&&haveAddedSpecial;s++){
    if(!specialTile[s]){
        if (specialTile[s]) {
            continue;
        }//whether the sth bit in specialTile has been used
        char t='0';
        t += s;
        String specialType="S"+t;

        for (char c = 'A'; c < 'H'; c++) {
            for (char j = '0'; j < '7'; j++) {
                for (char l = '0'; l < '8'; l++) {
                    String tailString=specialType+c+""+j+l;

                    if (isValidPlacementSequence( boardString: boardString + tailString) && !newTilesString[i]) {
                        boardString = boardString + tailString;
                        placementSequence = placementSequence + tailString;
                        newTilesString[i] = true;
                        specialTile[s]=true;
                        haveAddedSpecial = true;
                    }
                }
            }
        }
    }
}
```

generateMove

- 2. Construct models of tileString based on the diceRoll

```
// Authored by Bojle and Harriet  
public static String generateMove(String boardString, String diceRoll) {  
    String tileString1 = diceRoll.toCharArray()[0] + "" + diceRoll.toCharArray()[1] + "A0" + "0";  
    String tileString2 = diceRoll.toCharArray()[2] + "" + diceRoll.toCharArray()[3] + "A0" + "0";  
    String tileString3 = diceRoll.toCharArray()[4] + "" + diceRoll.toCharArray()[5] + "A0" + "0";  
    String tileString4 = diceRoll.toCharArray()[6] + "" + diceRoll.toCharArray()[7] + "A0" + "0";  
  
    String placementSequence = "";  
  
    String[] tileStringArray = {tileString1, tileString2, tileString3, tileString4};  
}
```

generateMove

- 3. Find the available place for new tileString and reconstruct them.

```
for (int i = 0; i < 5; i++) {
    for (int k = 0; k < 4; k++) {
        if (tiles[k]) {
            continue;
        } //whether the kth bit in diceroll has been used
        char[] tile = tileStringArray[k].toCharArray();
        for (char c = 'A'; c < 'H'; c++) {
            tile[2] = c;
            for (char j = '0'; j < '7'; j++) {
                tile[3] = j;
                for (char l = '0'; l < '8'; l++) {
                    tile[4] = l;
                    String tileString = "" + tile[0] + tile[1] + tile[2] + tile[3] + tile[4];
                    if (isValidPlacementSequence( boardString: boardString + tileString) && !newTilesString[i]) {
                        boardString = boardString + tileString;
                        placementSequence = placementSequence + tileString;
                        tiles[k]=true;
                        newTilesString[i] = true;
                    }
                }
            }
        }
    }
}
```


getBasicSocre

- `exitNumber`(*search the number of exits in the process of searching from the current tile*)
- `centreGridNum`(*Find the number of tiles which located at the centre grid*)
- `missEdges`(*Find the number of unconnected edges in the boardString*)

getBasicSocre exitNumber

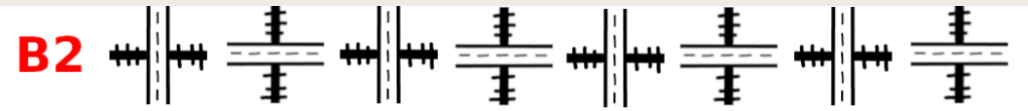
- Use *depth-first-search* to search the number of exits in the process of searching from the current tile
 - * @param pieces the current tile
 - * @param orderOfPieces the order number of tile in the boardString
 - * @param direction the direction of the next search
 - * @param firstexit the firstexit in the route of searching
 - * @param boardString a board string representing some placement sequence
 - * @return the number of exits in the process of searching from the current tile

```
public static int exitNumber(String pieces, int orderOfPieces, int direction, String firstexit, String boardString) {  
    String location = pieces.charAt(2) + "" + pieces.charAt(3);  
    if (!location.equals(firstexit)) {  
        int isAnExit = -1;  
        for (int i = 0; i < 12; i++) {  
            if (location.equals(exits[i])) {  
                isAnExit = i;  
            }  
        }  
        if (isAnExit >= 0 && direction == isAnExit / 3) {  
            if (!touchExit[isAnExit]) {  
                touchExit[isAnExit] = true;  
                return 1;  
            }  
            return 0;  
        }  
    }  
    //judge the number of exits on the edge  
    int total = 0;  
    for (int i = 0; i < boardString.length(); i += 5) {  
        String s = boardString.substring(i, i + 5);  
        if (areConnectedNeighbours(s, pieces) && isValidDirection(pieces, s, direction)) {  
            String next = s;  
            touchPile[orderOfPieces] = true; // assign true to the tiles which have been searched  
            if (s.substring(0, 2).equals("B2")) {  
                //touchPile[i/5] = false;  
                total += exitNumber(s, orderOfPieces: i / 5, direction, firstexit, boardString);  
            } else if (!touchPile[i / 5]) {  
                total += exitNumber(next, orderOfPieces: i / 5, direction: 0, firstexit, boardString);  
                total += exitNumber(next, orderOfPieces: i / 5, direction: 1, firstexit, boardString);  
                total += exitNumber(next, orderOfPieces: i / 5, direction: 2, firstexit, boardString);  
                total += exitNumber(next, orderOfPieces: i / 5, direction: 3, firstexit, boardString);  
            }  
            touchPile[orderOfPieces] = false;  
        }  
    }  
    return total;  
}
```

getBasicSocre

one important point

- About B2, due to its special construction, when *depth-first-search* through it , its direction wont change.



```
touchPile[orderOfPieces] = true; // assign true to the tiles which have been searched
if (s.substring(0, 2).equals("B2")) {
    total += exitNumber(next, orderOfPieces: i / 5, direction, firstexit, boardString);
} else if (!touchPile[i / 5]) {
    total += exitNumber(next, orderOfPieces: i / 5, direction: 0, firstexit, boardString);
    total += exitNumber(next, orderOfPieces: i / 5, direction: 1, firstexit, boardString);
    total += exitNumber(next, orderOfPieces: i / 5, direction: 2, firstexit, boardString);
    total += exitNumber(next, orderOfPieces: i / 5, direction: 3, firstexit, boardString);
}
touchPile[orderOfPieces] = false;
}
```

getBasicScore

calculate the final score

- Calculate the points which matches the number of exits, then
- `score = score + centreGridNum(boardString) - missEdges(boardString);`

getAdvancedScore

- Similar to `getBasicScore`, *use one Method "findMaxLength" (based on depth-first-search) to find the max length Railway and Highway.*
- `return maxHighWay + maxRailWay + getBasicScore(boardString);`

getAdvancedScore

findMaxLength

- Find the maximum length of the road from the current piece
 - * **@param piece** a given current piece
 - * **@param orderOfNumber** the order of piece in the boardString
 - * **@param direction** the direction of the next search. '0' means north, '1' means east, '2' means south, '3' means west
 - * **@param type** the type of tile, '1' means highway, '2' means railway.
 - * **@param deep** the depth of iterator function, the deep will be deep+1 after one iteration.
 - * **@param boardString** a board string representing a completed game
 - * **@return** the maximum length of the road from the current piece

```
// Authored by Bofie
public static int findMaxLength (String piece, int orderOfNumber, int direction, int type, int deep, String boardString) {
    boardString){
        int length = deep;
        for (int i = 0; i < boardString.length(); i += 5) {
            String s = boardString.substring(i, i + 5);
            if (!touchPile[i / 5] && areConnectedNeighbours(piece, s) && isValidDirection(piece, s, direction)) {
                touchPile[orderOfNumber] = true;
                String next = s;
                String nextType = next.charAt(0) + "" + next.charAt(1) + next.charAt(4);
                Tile nextTile = Tile.valueOf(nextType);
                if (nextTile.north == type) {
                    length = Math.max(length, findMaxLength(next, orderOfNumber: i / 5, direction: 0, type, deep: deep + 1, boardString));
                }
                if (nextTile.east == type) {
                    length = Math.max(length, findMaxLength(next, orderOfNumber: i / 5, direction: 1, type, deep: deep + 1, boardString));
                }
                if (nextTile.south == type) {
                    length = Math.max(length, findMaxLength(next, orderOfNumber: i / 5, direction: 2, type, deep: deep + 1, boardString));
                }
                if (nextTile.west == type) {
                    length = Math.max(length, findMaxLength(next, orderOfNumber: i / 5, direction: 3, type, deep: deep + 1, boardString));
                }
                touchPile[orderOfNumber] = false;
            }
        }
        return length;
    }
}
```

getAdvancedScore

Iteration process

```
if (tile.north == 1) {
    maxHighWay = Math.max(maxHighWay, findMaxLength(piece, orderOfNumber: i / 5, direction: 0, tile.north, deep: 1, boardString));
}
if (tile.east == 1) {
    maxHighWay = Math.max(maxHighWay, findMaxLength(piece, orderOfNumber: i / 5, direction: 1, tile.east, deep: 1, boardString));
}
if (tile.south == 1) {
    maxHighWay = Math.max(maxHighWay, findMaxLength(piece, orderOfNumber: i / 5, direction: 2, tile.south, deep: 1, boardString));
}
if (tile.west == 1) {
    maxHighWay = Math.max(maxHighWay, findMaxLength(piece, orderOfNumber: i / 5, direction: 3, tile.west, deep: 1, boardString));
}
//calling the iterator function in for direction to calculate the maxHighway

if (tile.north == 2) {
    maxRailWay = Math.max(maxRailWay, findMaxLength(piece, orderOfNumber: i / 5, direction: 0, tile.north, deep: 1, boardString));
}
if (tile.east == 2) {
    maxRailWay = Math.max(maxRailWay, findMaxLength(piece, orderOfNumber: i / 5, direction: 1, tile.east, deep: 1, boardString));
}
if (tile.south == 2) {
    maxRailWay = Math.max(maxRailWay, findMaxLength(piece, orderOfNumber: i / 5, direction: 2, tile.south, deep: 1, boardString));
}
if (tile.west == 2) {
    maxRailWay = Math.max(maxRailWay, findMaxLength(piece, orderOfNumber: i / 5, direction: 3, tile.west, deep: 1, boardString));
}
//calling the iterator function in for direction to calculate the maxRailway
```

Design Approach

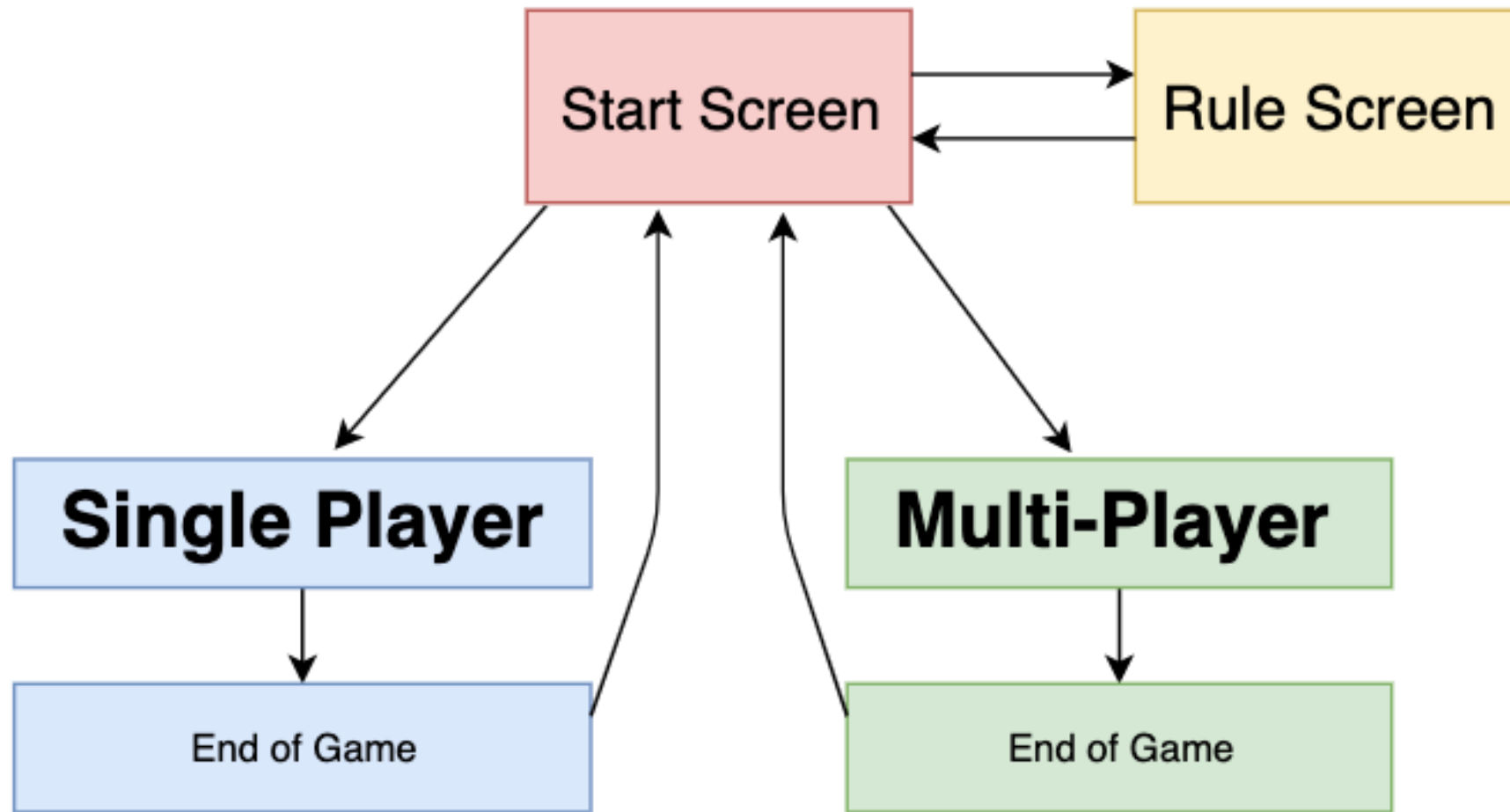
- Intuitive, “pick up and play” design
 - Easy to use drag and drop system
 - Labelled buttons that show exactly what they do
- Rule page so player doesn't have to source outside information to learn how scoring, etc, works

Interesting Features

- Drag and drop system
- Start Screen
- Rules Page
- Ending

Game Operation

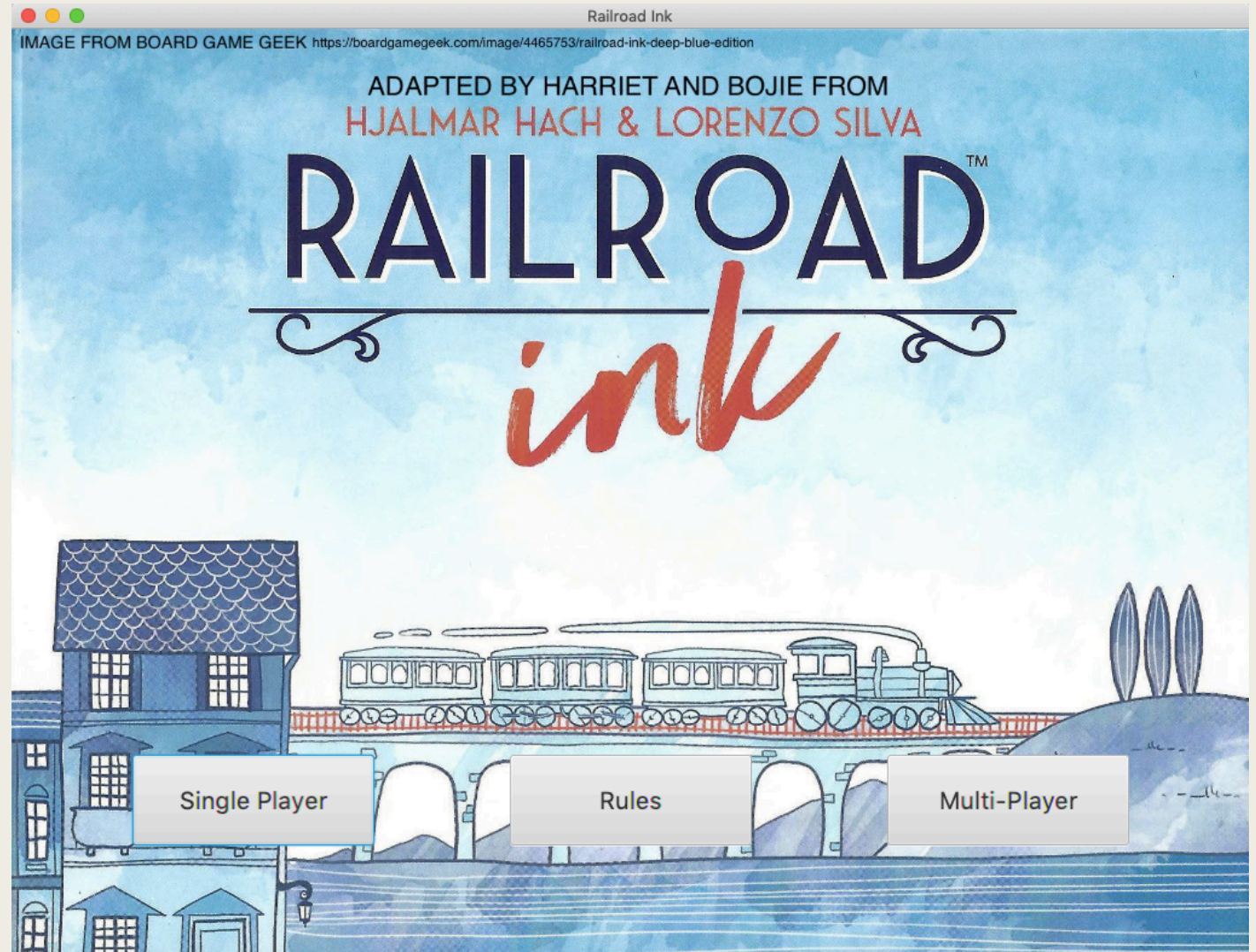
- 4 Components;
 - *Start Screen*
 - *Rules Screen*
 - *Single Player Game*
 - *Multi-player Game*



On Startup - Start Screen

Start screen consist of:

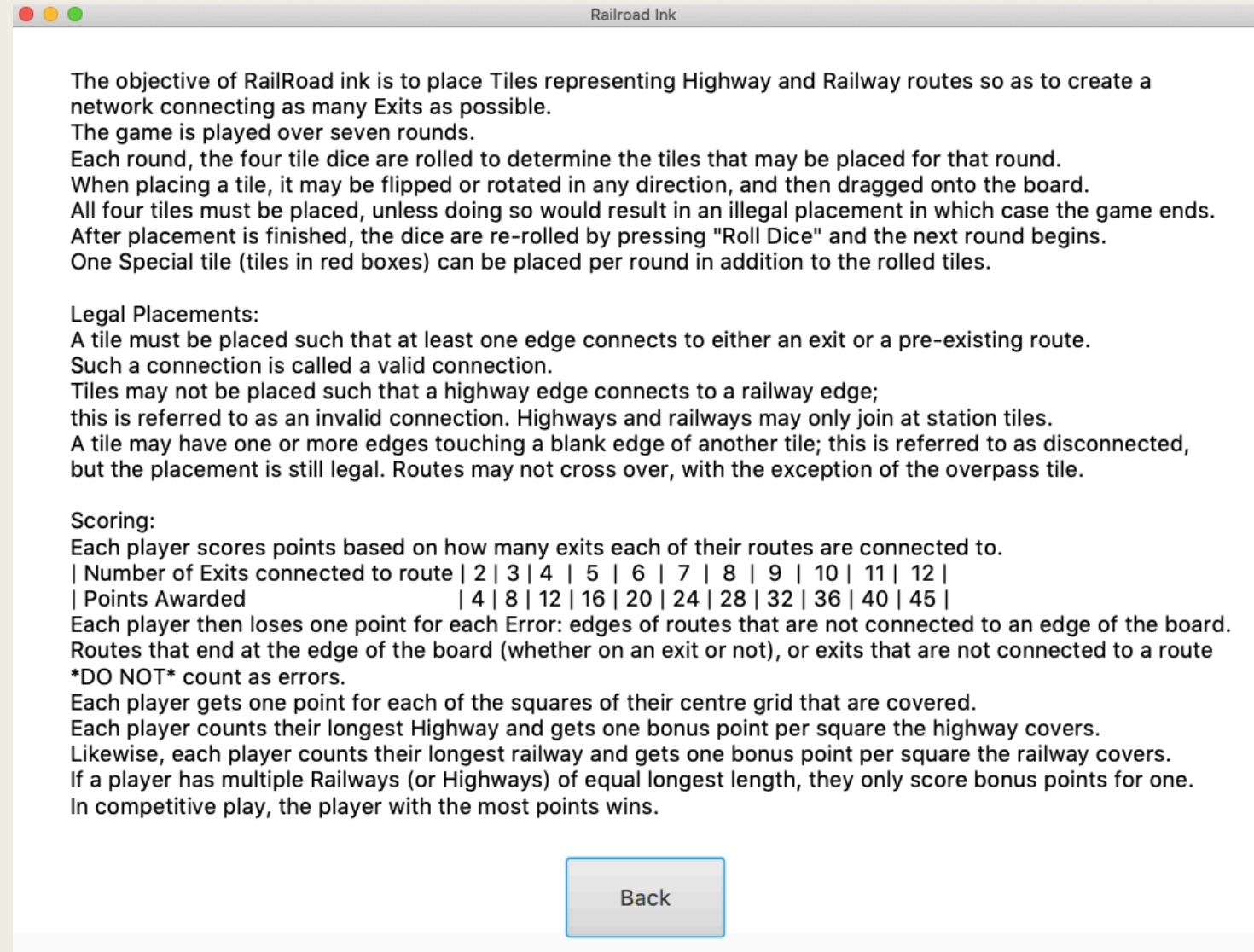
- Background ImageView
- 3 Buttons
 - Single Player
 - Launches a new instance of a Single Player stage and closes the start screen stage
 - Rules
 - Launches a new instance of a Rules stage and closes the start screen stage
 - Multiplayer
 - Launches a new instance of a Multi-Player stage and closes the start screen



Rules Screen

Consist of:

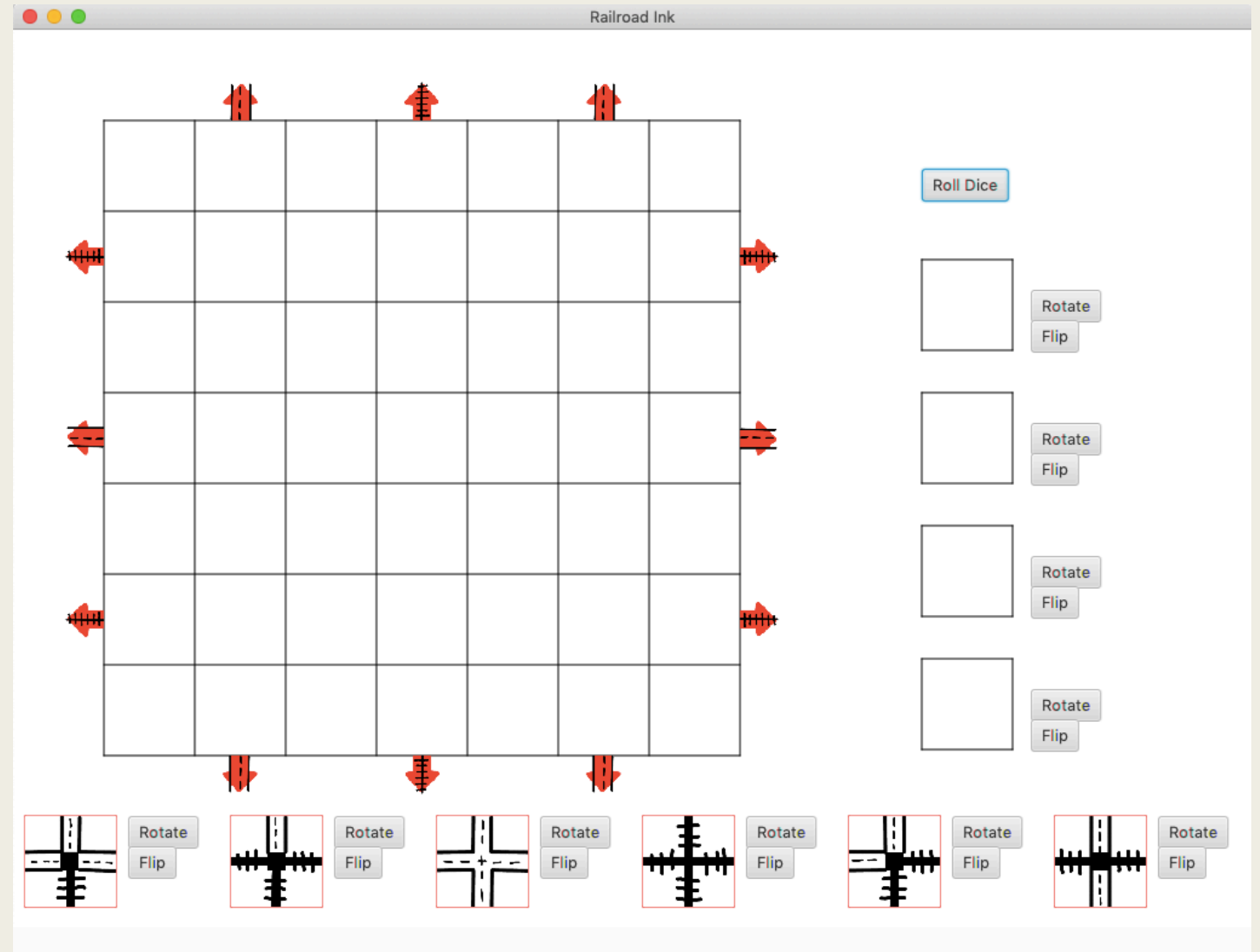
- Text object containing all the rules for the game
- Back button which closes this stage and creates a new instance of the start screen stage.



Single Player

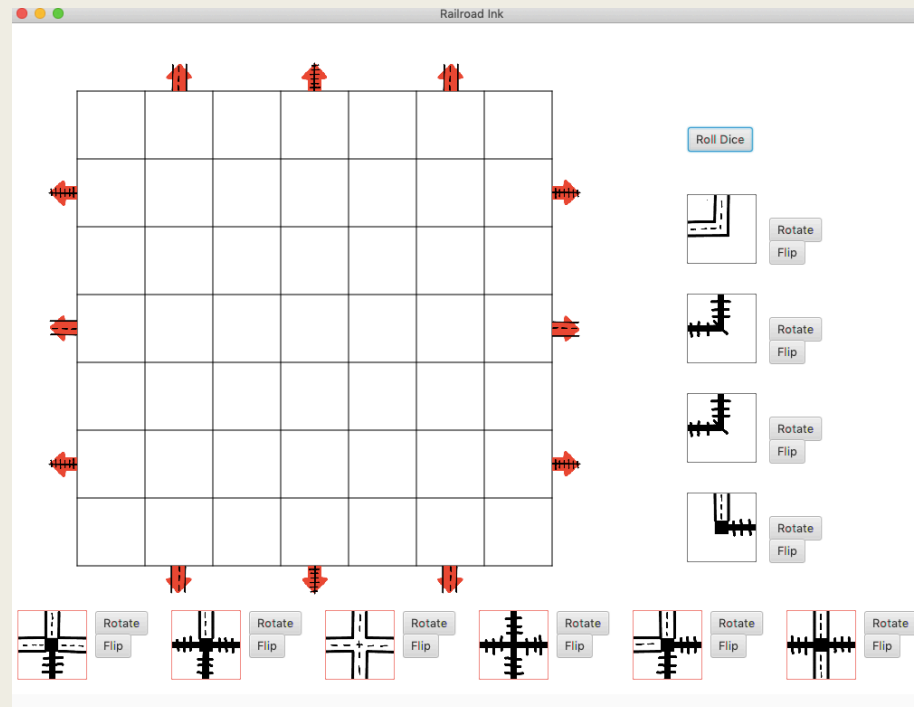
Consist of:

- Board and tile starting placement boxes drawn using Line objects
- The Image View highway and railway exits
- Draggable Image View Special Tiles
- The “Roll Dice” button which generates tiles for each round
- “Rotate” and “Flip” buttons for each tile



How a Single Player Game Works

- The game starts with special tiles being unplayable
 - If the player attempts to play one it will snap back to it's original position
- Once the “Roll Dice” button has been pressed, the round number is incremented, if > 7 calls endGame else the method drawNewTiles() is called.

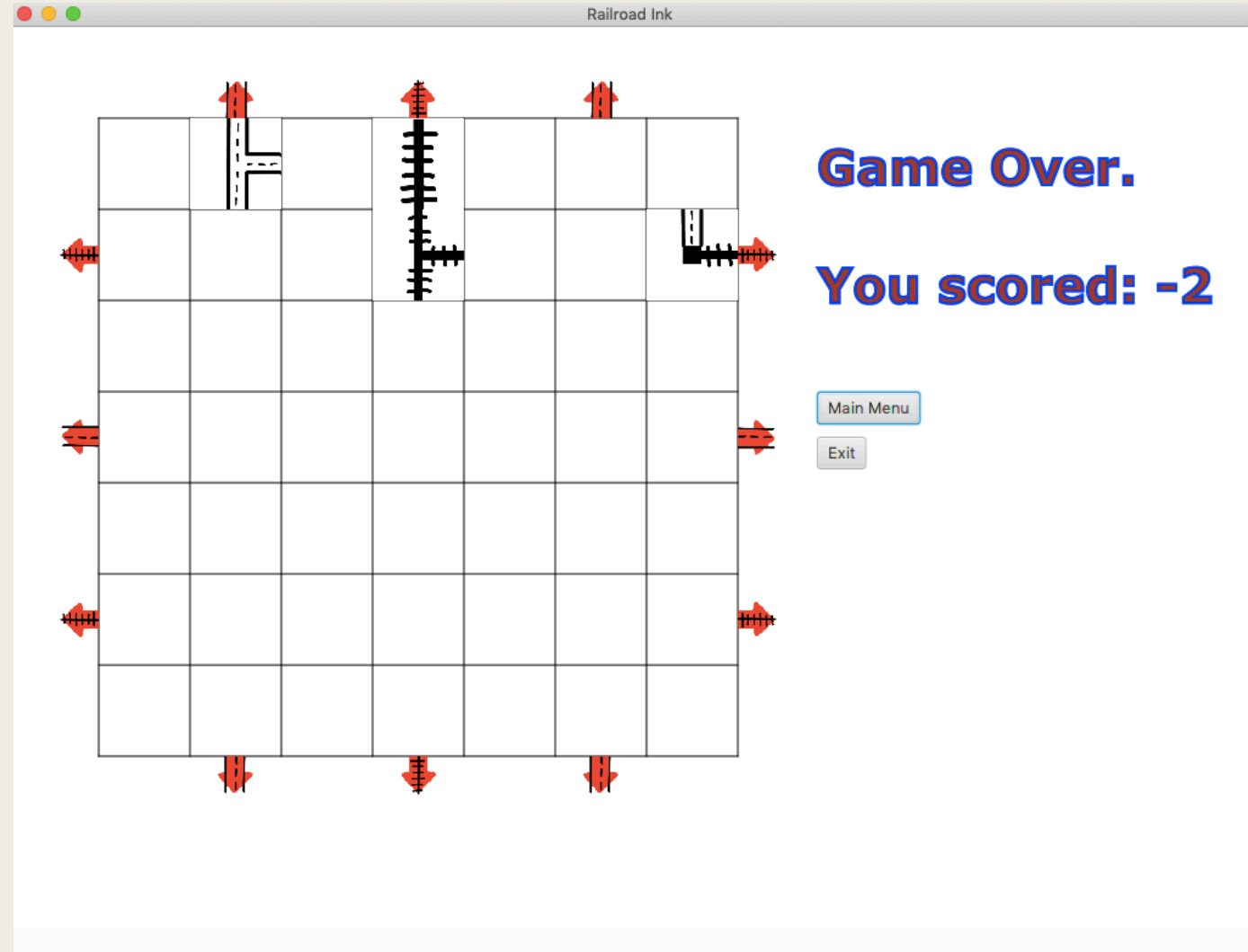


Single Player Game cont'd

- `drawNewTiles()` generates a dice roll, checks that all the tiles can be played (if not, calls `endGame()`), and then creates a draggable `TileImageView` object for each tile generated.
- Tiles then become playable. Dragging and dropping a tile calls `drawTile(double x, double y, String tileName, int rotation, int orientation)`, where `x` and `y` are the location the mouse was at when released, `rotation` is the tile's rotation and `orientation` is the tile's orientation.
- If a tile placement is invalid, `drawTile` returns false and the tile is then returned to its origin
- If the tile placement is valid, the draggable `ImageView` is deleted and a new `ImageView` is created with the same specifications of the draggable tile. The `x` and `y` locations are found by the method `snapToGrid(double x, double y)` which finds where the tile was dropped on the board and snaps its location to it.
- `drawTile` then checks if each of the remaining tiles can be played, and if not, calls `endGame()`.

Single Player - endGame()

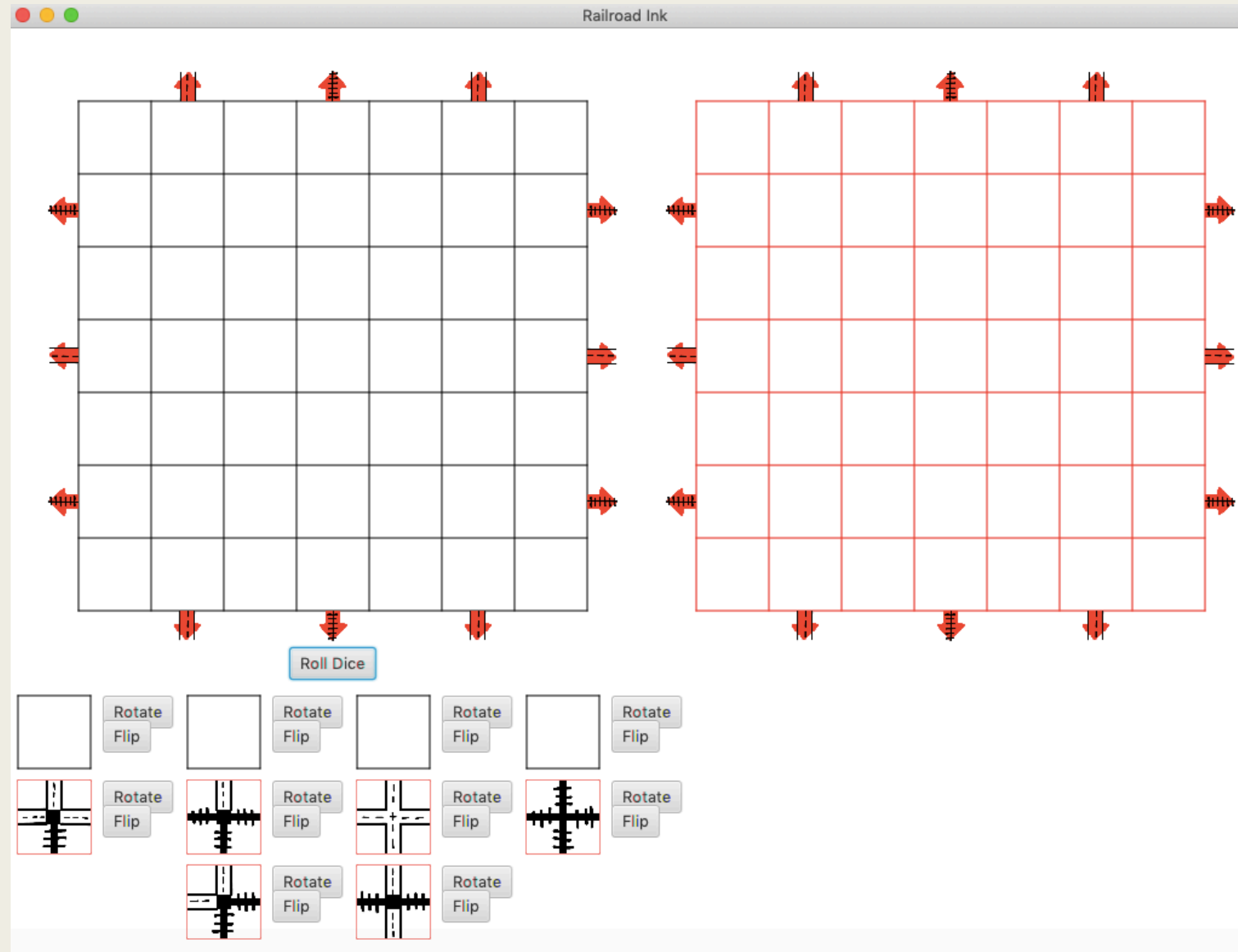
- Removes everything from the root excluding the board and played tiles
- Creates a Text object displaying the player's score
- Creates 2 buttons;
 - *Main Menu* - Closes the stage and creates an instance of the start screen stage
 - *Exit* - Closes the stage



Multi-Player

Consist of:

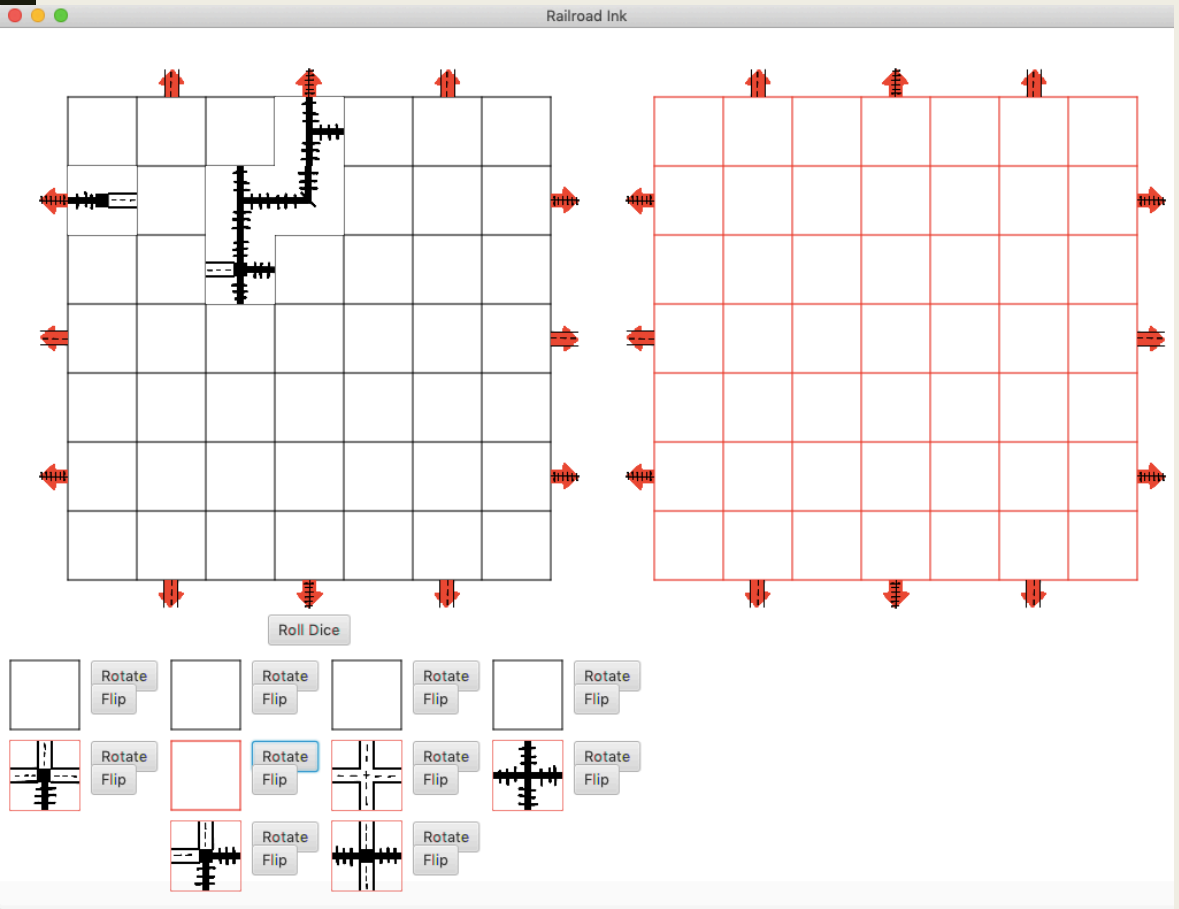
- Player and computer Boards and tile starting placement boxes drawn using Line objects
- The Image View highway and railway exits
- Draggable Image View Special Tiles
- The “Roll Dice” button which generates tiles for each round
- “Rotate” and “Flip” buttons for each tile



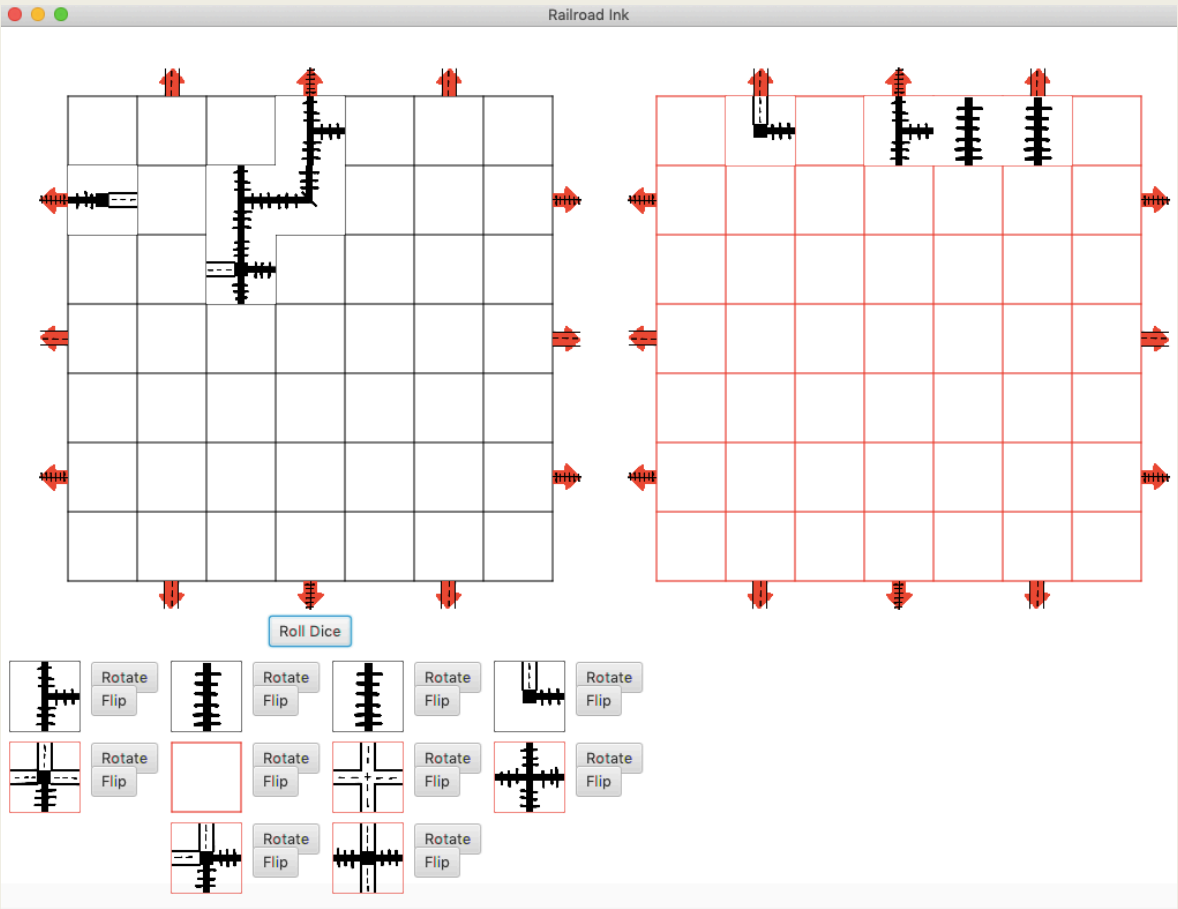
How a Multi-Player Game Works

- Very similar to a Single Player game, with a few adjustments;
 - When a new Round begins, if there was a previous round, the Computer plays the tiles from that round.
 - If the computer cannot make valid placements for each tile, it plays the ones it can but does not continue to play in further rounds.

End of first round – player has played all of their tiles

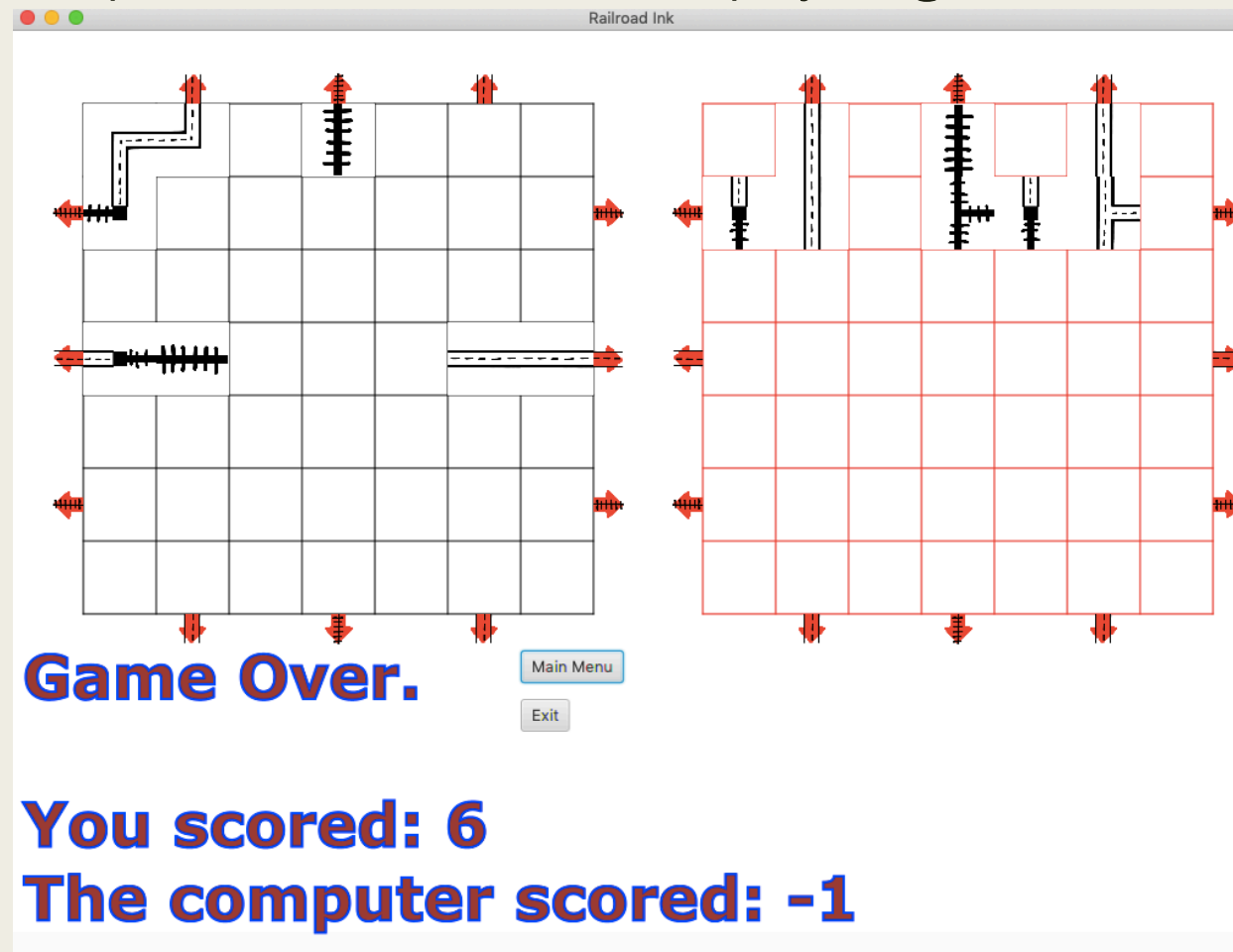


Start of second round – Computer plays Round 1 tiles



End of Multiplayer Game

- The end of a multiplayer game is the same as a single player game except that it shows the computer's score as well as the player's game.



Pseudo Code

```
Start() {
    Add board to root
    Add controls to root
    drawBoard();
    makeControls();
    drawSpecial();
}

makeControls() {
    Button rollDice {
        If(clicked) {
            //if multiplayer
            if (Round > 0 and computer is in play) {
                place tiles from previous round
                if (computer hasn't player all tiles) {
                    computer out of play
                }
            }

            drawNewTiles();

            Round ++

            If(all rounds played) {
                End game
            }
        }
    }
}
```

```
for(all tiles) {
    Button rotate {
        If(clicked) {
            Change tile rotation;
        }
    }
    Button flip {
        If(clicked) {
            Flip tile;
        }
    }
}

drawNewTiles() {
    Roll = generateDiceRoll();
    For (each tile in Roll) {
        New draggable tile(image tileImage, origin x, origin y);
    }
}

Mouse Event Click {
    get origin x;
    get origin y;
}

Mouse Event Drag {
    Move tile with mouse
}
```

Pseudo Code Cont'd

```
Mouse Event Drop {  
    Boolean draw = drawTile(tile, x, y);  
  
    If(!draw) {  
        Return to origin  
    }  
}
```

```
Boolean draw tile(x, y, tile) {  
    If (placement is valid) {  
        Draw non draggable tile on board  
        Return true;  
    } else {  
return false;  
    }  
}
```

```
endgame() {  
    remove everything but played tiles and board(s)  
    show score(s)  
    Menu Button {  
        Exit;  
        Launch start screen;  
    }  
}
```

```
Menu Button {  
    Exit;  
}  
  
}
```

Questions?