ECE 373 FINAL REPORT
GUITAR HERO
GROUP 16

JULIE MASON
DAVID POLK
EVAN RAINS

TABLE OF CONTENTS

PROBLEM DESCRIPTION

Our project is a musical rhythm game based on the concept of Guitar Hero. The gameplay involves pressing keys on the keyboard to the rhythm of a song. Colored blocks fall from the top of the screen, and each color corresponds to a certain key. The user must press the right key when a block reaches the bottom of the screen. At startup, a title screen will appear, giving the user various options in the form of buttons. Such options include choosing difficulty, choosing a song, and viewing gameplay instructions. When the user selects the instruction button, a popup window will appear, displaying the instructions and object of the game. There is also an option to view high scores, which displays the top 3 highest scores.

When the user decides to start the game, they will select the “Play” button. When the game begins, the song will start playing and the blocks will begin falling. The current score will be displayed in the upper portion of the screen. If the user misses a block, points will be deducted from their score. If they miss a certain number of blocks, the game will end before the song completes, and their score will be displayed along with a message stating that they failed the song. If the user is able to complete the entire song, their score will be displayed along with a congratulatory message.

If a new high score has been achieved, a popup window will appear asking the user if they would like to save their high score and enter a username. There is also the option to exit the game at any time by pressing the escape button. However, the user’s score will not be recorded if they exit prematurely. Exiting will lead the user back to the title screen.

HOW TO SETUP AND RUN THE SYSTEM

In order to run the application, first load the *RhythmGame* java project into Eclipse. Then, expand the 'src' folder in the package explorer. Expand the '*RhythmGame.Screen*' package. Right-Click the '*GameDriver.java*' file, and select '*Run as*' -> '*Java Application*'.

Once the application is open you will be presented with a Main Menu with the following options. You can access these options by clicking on the corresponding button block…

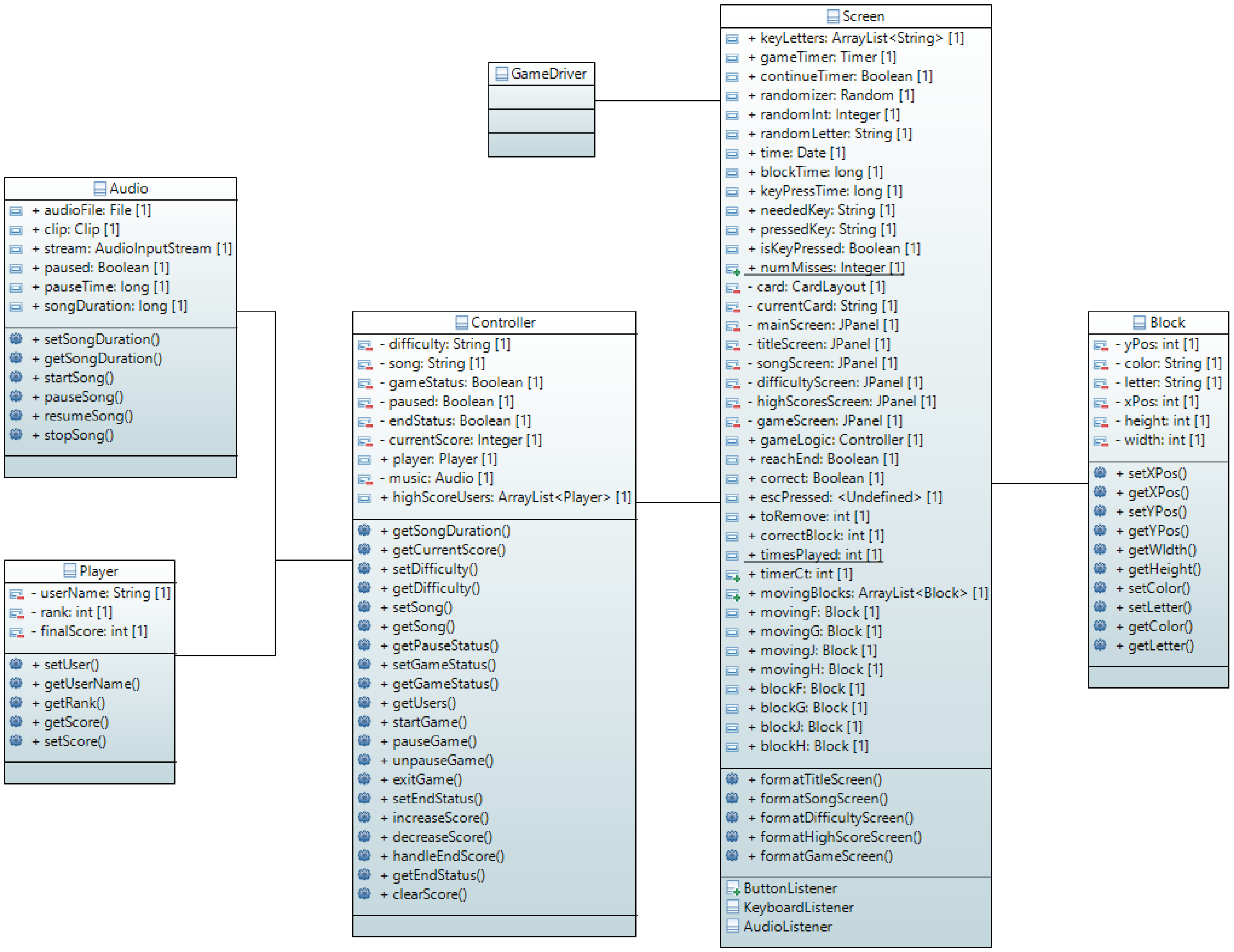
- *Play Now! - immediately starts a new game session*
- *Choose Song - opens screen for selecting the desired song*
- *Choose Difficulty - opens screen for selecting the desired difficulty*
- *View High Scores - opens screen displaying the current high scores*
- *View Instructions - opens a popup window explaining game controls*

To play the game, click '*Play Now!*' and the game will immediately start…

- As soon as a letter appears on the screen, you must quickly press the corresponding letter on your keyboard.
    - If you input the correct letter in time, your score will increase
    - If you input the incorrect letter, your score will decrease
    - If you fail to input the correct letter in time, your score will decrease
- Keep playing until the song ends, at which point the game will be completed.
    - If you lose too many points, the game will end immediately.
        - A 'Failed Game' screen will appear. You must then click 'OK' to return to the main menu
    - You may also pause the game at any time by pressing the 'Esc' key on your keyboard.
        - An 'Exit' screen will pop-up. Click 'Yes' to return to the main menu, or click 'No' to resume the game.

HAVE FUN!

DOMAIN CLASS MODEL



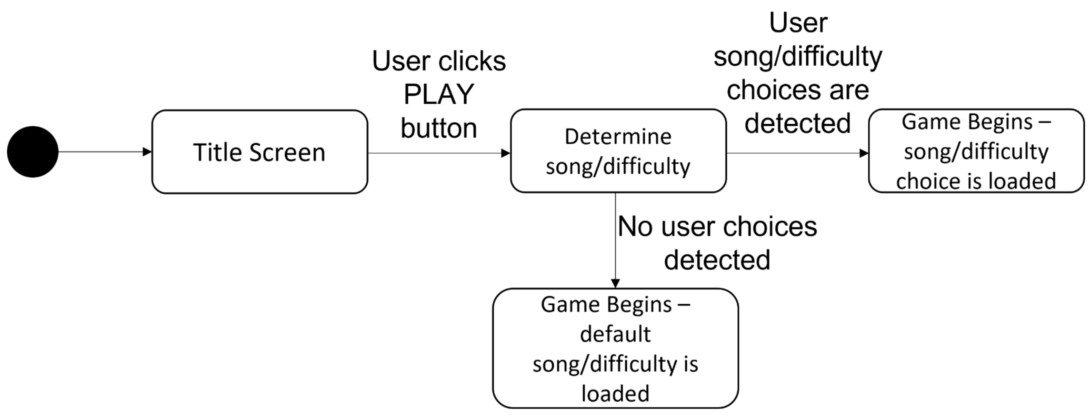
*****

Almost all classes are related by simple association, except for blocks, which are derived from an abstract base class. Majority of game logic is contained within Screen and Controller classes.

*****

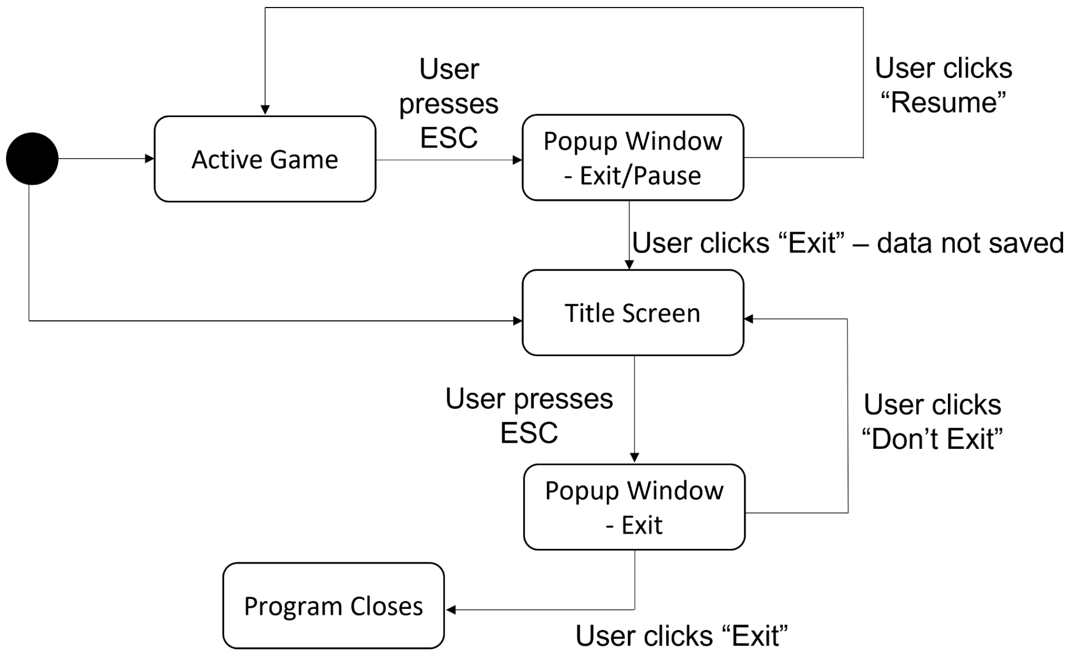
STATE MODELS

START NEW GAME


Title Screen
User clicks PLAY button
Determine song/difficulty
User song/difficulty choices are detected
Game Begins – song/difficulty choice is loaded
No user choices detected
Game Begins – default song/difficulty is loaded

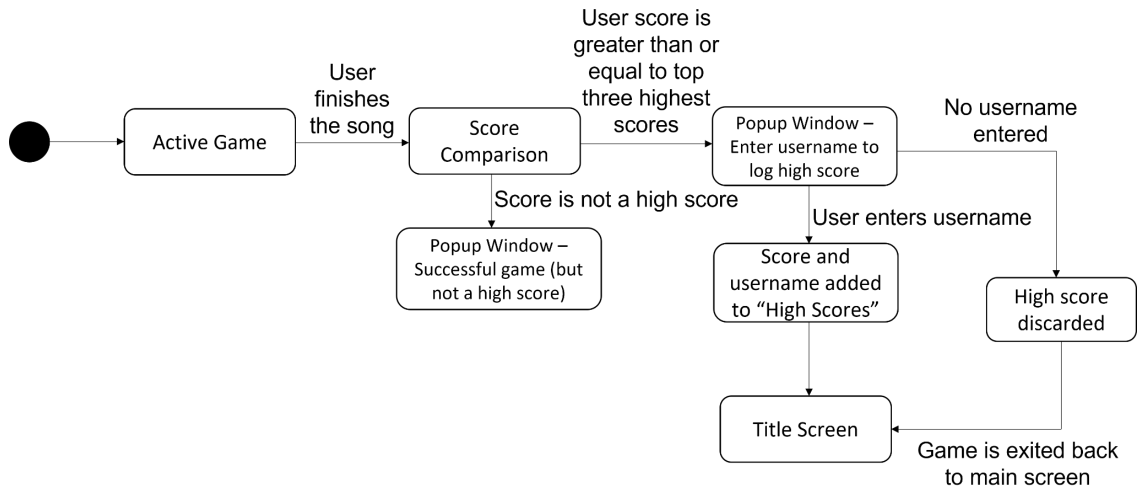
formatTitleScreen() method enters into this state. formatDifficultyScreen() and formatSongScreen() methods are then used to select desired values. State can be exited using formatGameScreen() and startGame() methods to begin the game instance.

PAUSE/EXIT GAME



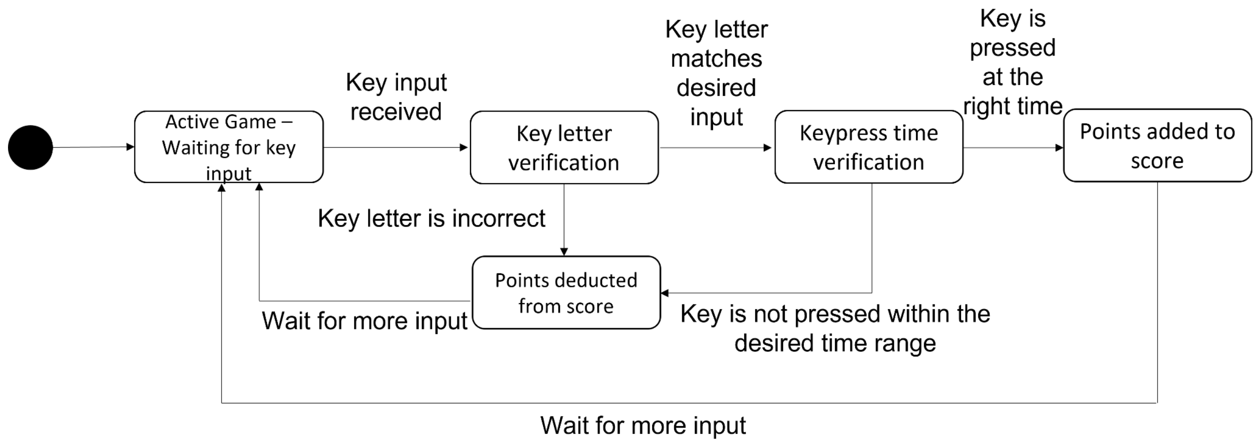
formatGameScreen() along with startGame() methods enter into this state, which then may accept pauseGame() to display the pause window, which waits for further input. State can be exited using unpauseGame() method to resume the game instance. State is also left using exitGame() method to return to the main title screen.

NEW HIGH SCORE


Active Game
User finishes the song
Score Comparison
User score is greater than or equal to top three highest scores
Popup Window – Enter username to log high score
No username entered
Score is not a high score
User enters username
Popup Window – Successful game (but not a high score)
Score and username added to "High Scores"
High score discarded
Title Screen
Game is exited back to main screen

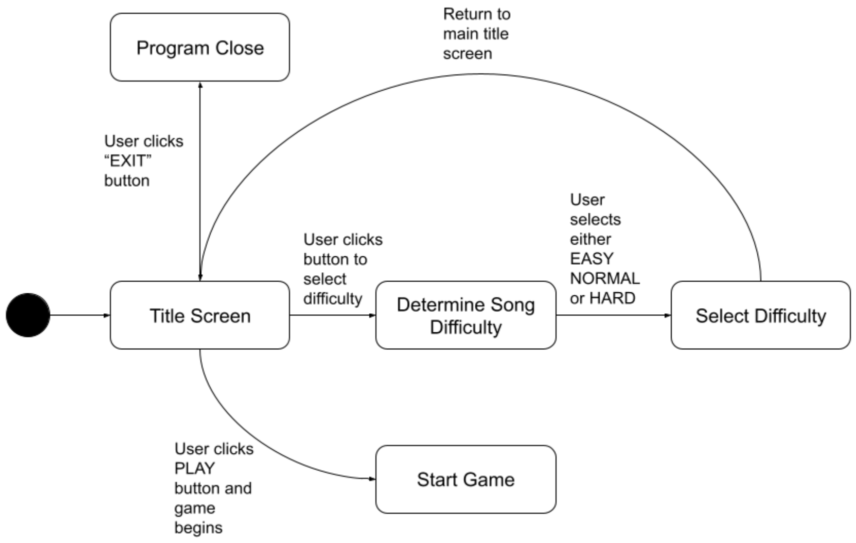
formatGameScreen() along with startGame() methods enter into this state. When the current game is complete, the state then accepts handleEndScore() and setEndStatus() methods to determine if the high score should be updated, gather username information, and update accordingly. formatHighScoreScreen() method returns state to title screen.

KEY INPUT


Active Game – Waiting for key input
Key input received
Key letter verification
Key letter matches desired input
Keypress time verification
Key is pressed at the right time
Points added to score
Key letter is incorrect
Points deducted from score
Wait for more input
Key is not pressed within the desired time range
Wait for more input

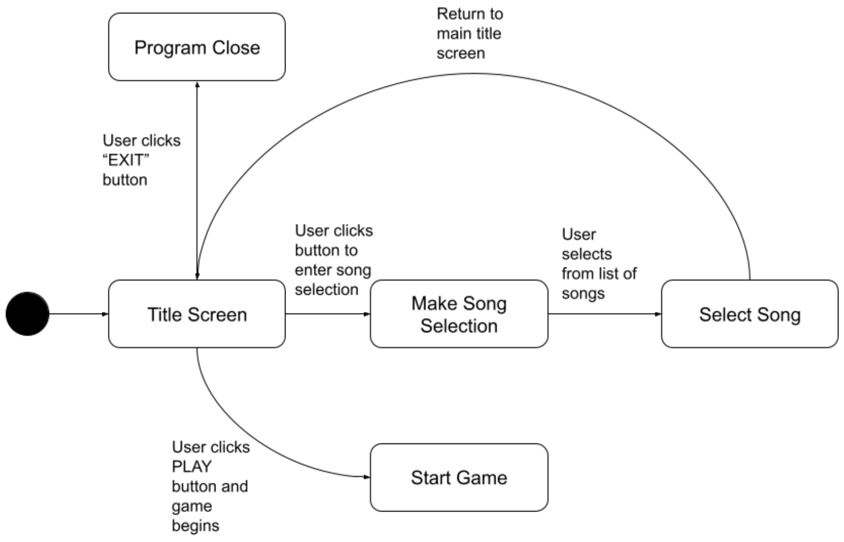
formatGameScreen() along with startGame() methods enter into this state. While the game is running, KeyboardListener accepts and verifies input to determine whether score should increase or decrease using increaseScore() and decreaseScore(). Signals are set to block instances to display correct visual feedback.

SELECT DIFFICULTY


Program Close
Return to main title screen
User clicks "EXIT" button
User selects either EASY NORMAL or HARD
User clicks button to select difficulty
Title Screen
Determine Song Difficulty
Select Difficulty
User clicks PLAY button and game begins
Start Game

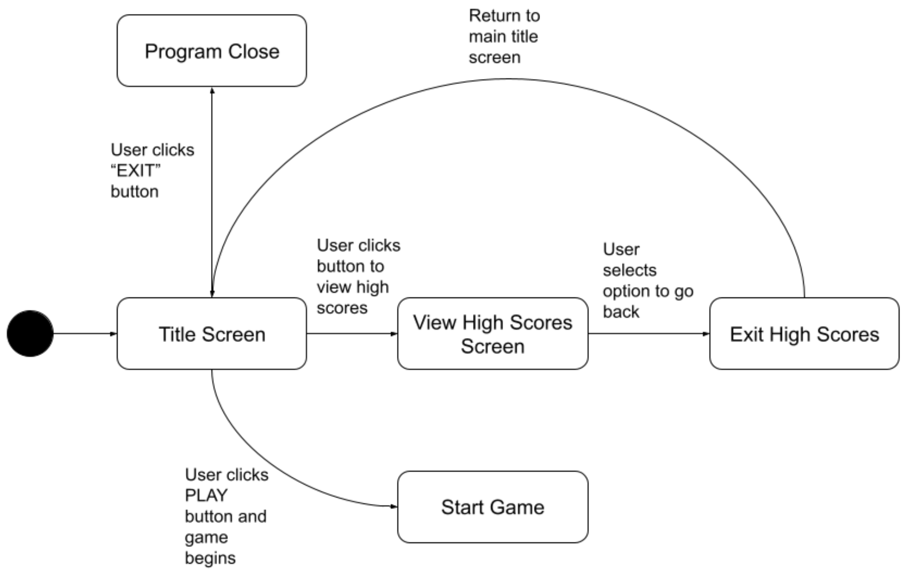
formatTitleScreen() method enters into this state. formatDifficultyScreen() method is then used to select desired values. After accepting a difficulty value with setDifficulty(), State can be exited using the exit button, which calls formatTitleScreen() to return to the main menu.

SELECT SONG


Program Close
Return to main title screen
User clicks "EXIT" button
User clicks button to enter song selection
User selects from list of songs
Title Screen
Make Song Selection
Select Song
User clicks PLAY button and game begins
Start Game

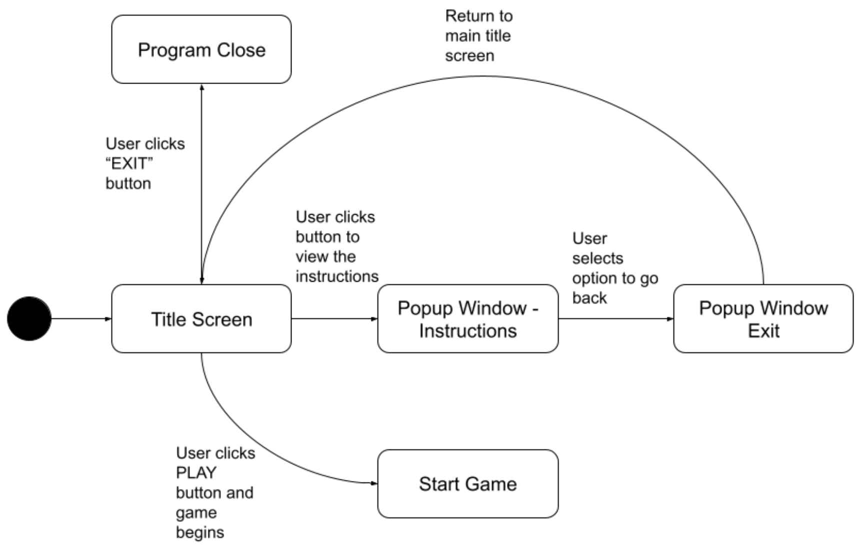
formatTitleScreen() method enters into this state. formatSongScreen() method is then used to select desired values. After accepting a difficulty value with setSong(), State can be exited using the exit button, which calls formatTitleScreen() to return to the main menu.

VIEW HIGH SCORES


Program Close
Return to main title screen
User clicks "EXIT" button
User clicks button to view high scores
User selects option to go back
Title Screen
View High Scores Screen
Exit High Scores
User clicks PLAY button and game begins
Start Game

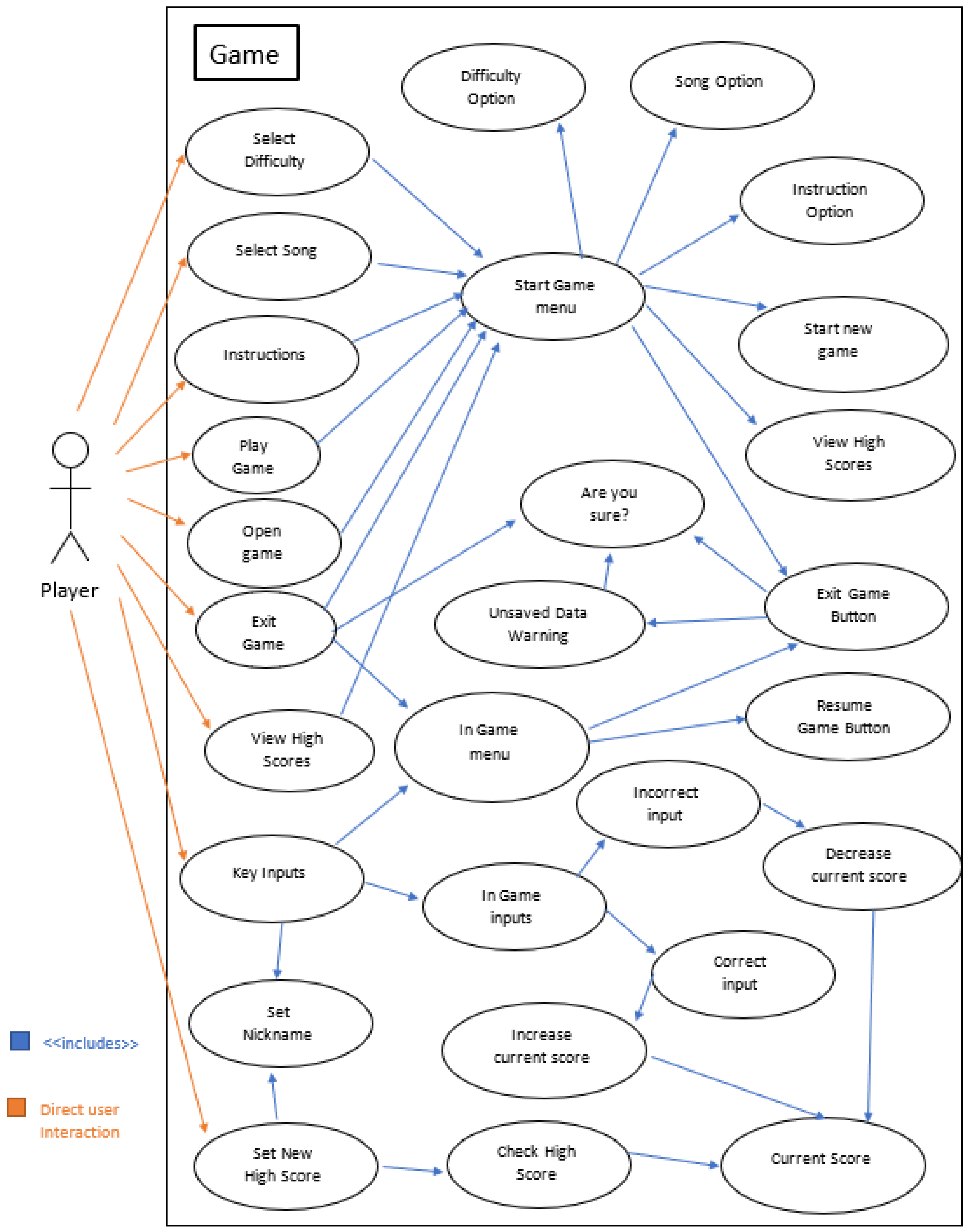
formatTitleScreen() method enters into this state. formatHighScoreScreen() method is then used to display score values along with usernames. After viewing this information, the State can be exited using the exit button, which calls formatTitleScreen() to return to the main menu.

VIEW INSTRUCTIONS

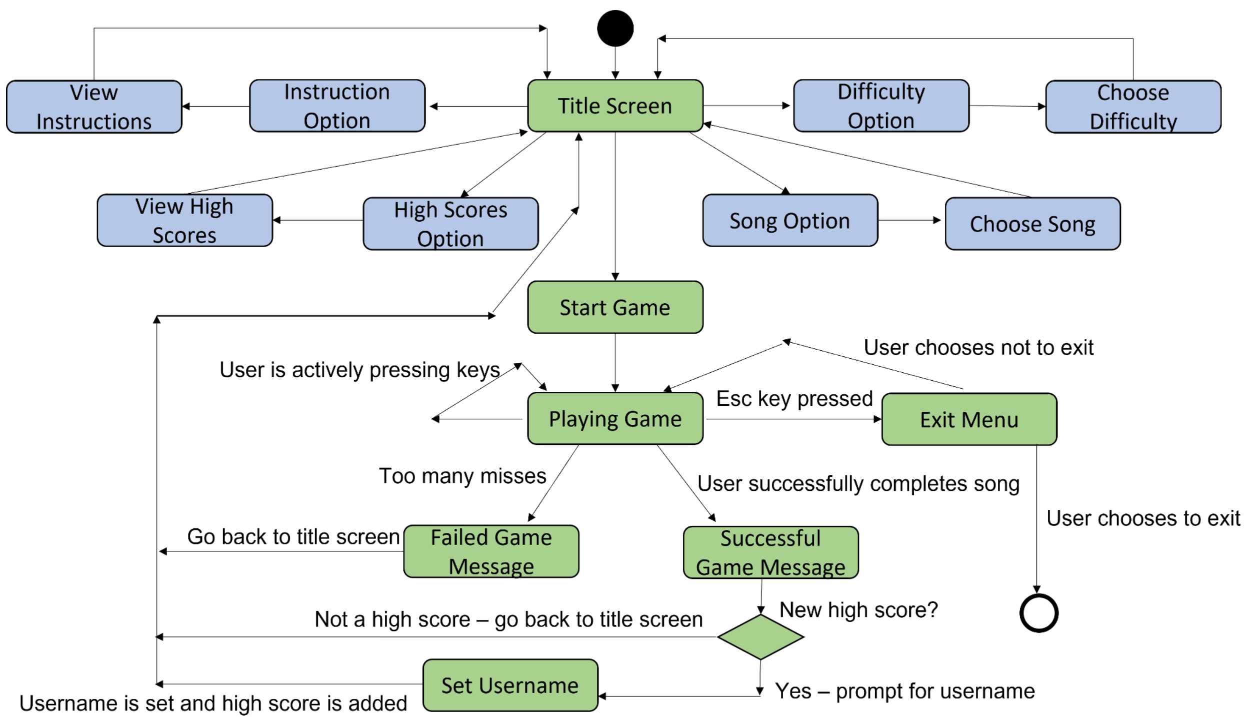

Program Close
Return to main title screen
User clicks "EXIT" button
User clicks button to view the instructions
User selects option to go back
Title Screen
Popup Window - Instructions
Popup Window Exit
User clicks PLAY button and game begins
Start Game

formatTitleScreen() method enters into this state. ButtonListener determines if instructions need to be displayed, in which case a popup method is called. After accepting an exit signal, the popup is terminated and the formatTitleScreen() method returns to the main menu.

INTERACTION MODEL



ACTIVITY DIAGRAM



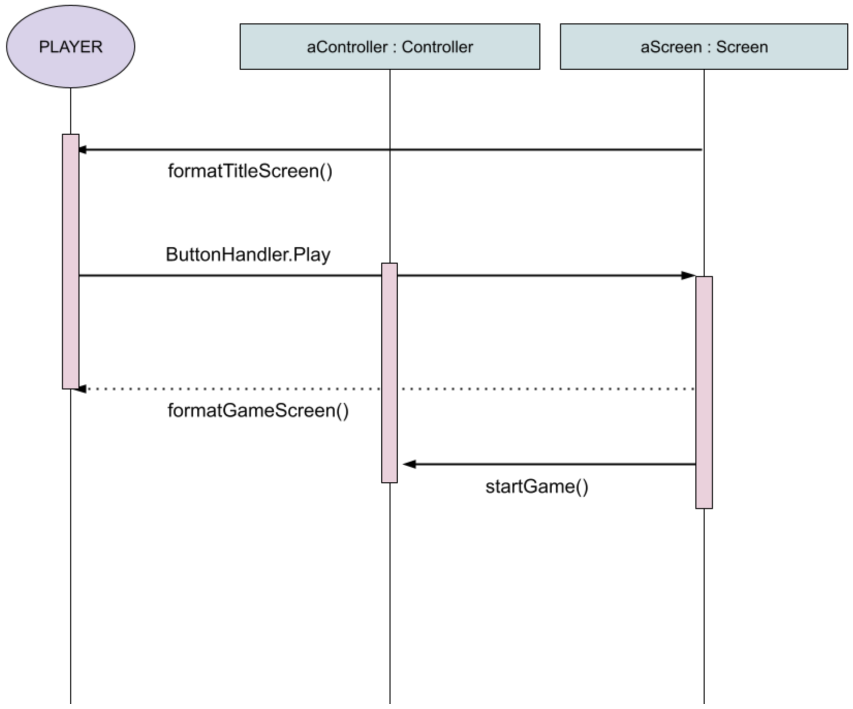
*****

Activity diagram provides top level overview of the game logic. Configuration is done from the title screen, after which a game instance is begun. The user may choose to pause or exit at any time. Assuming the player is successful, they are given the option to enter a username for a high score and return to the main menu to play again.

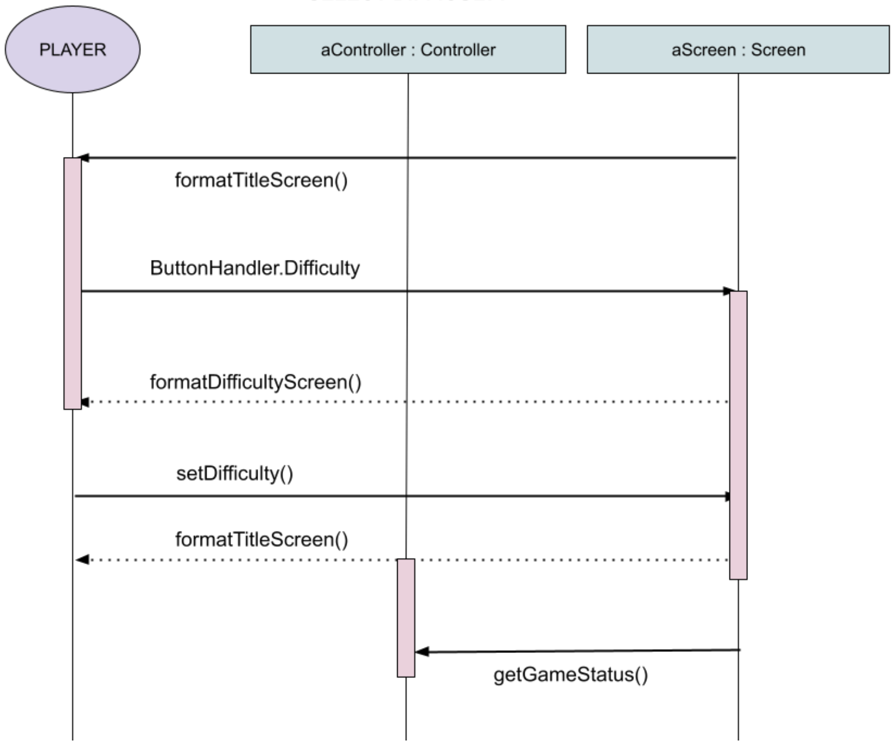
*****

SEQUENCE DIAGRAMS

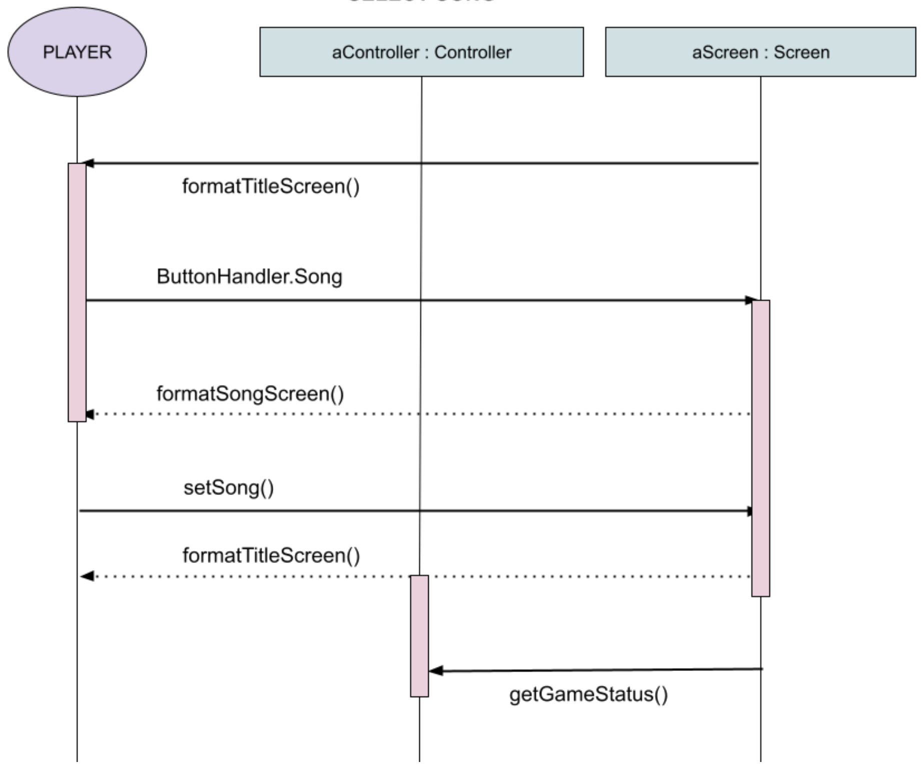
PLAY GAME



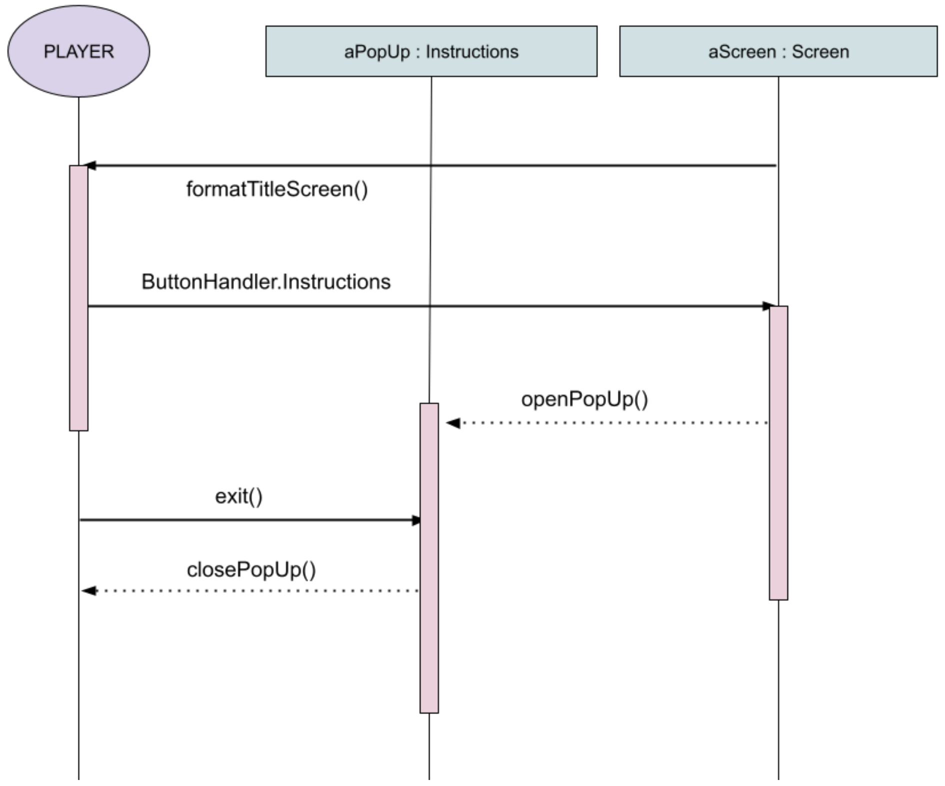
SELECT DIFFICULTY



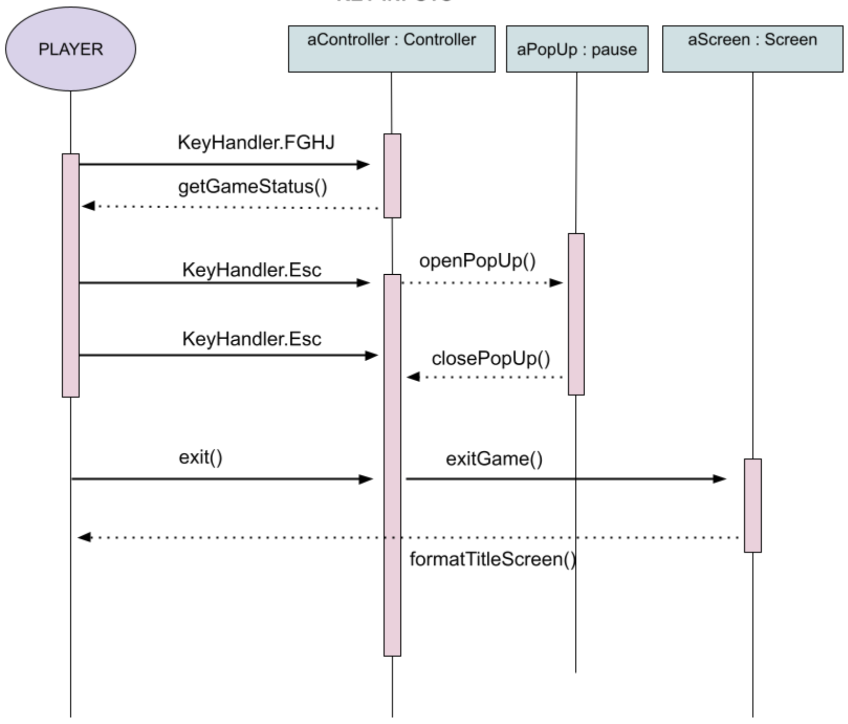
SELECT SONG



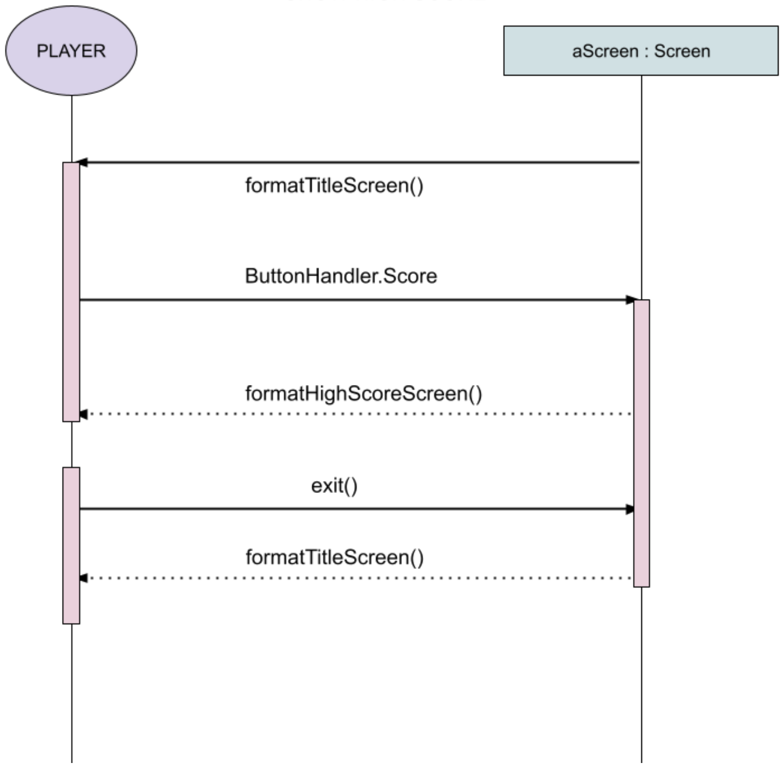
INSTRUCTIONS



KEY INPUTS



SHOW HIGH SCORE



USE CASES

Use Case Name: Open Game

Actors: Player

Preconditions: Game must be closed and waiting to be opened.

Description: When starting a new game, the GUI will open and display the "Start Game menu". There, the player will be given the option to start a new game, view the high scores, select a song, or view instructions.

Exceptions:
File corrupted/nonexistent: If the file doesn't exist on the computer or is corrupted, the program cannot open

Post-Conditions:
The program will stay in the Start Game menu waiting for the player to choose an option.

Use Case Name: Exit Game

Actors: Player

Preconditions: The program must be open and in a menu screen (Start game menu or Pause game menu)

Description: When the "Exit game" option is selected, a prompt will pop up asking the player if they are sure they want to exit the game. If the player is in the middle of a game, a prompt will pop up informing the player that their data will be unsaved within the "Are you sure?" prompt.

Exceptions:
Unexpected exit: If the game is exited by simply closing the application, the "Are you sure?" prompt will not appear, and all the data will be unsaved (including high score).
In-game exit: If the player exits from the In-game menu, their data will be unsaved (including high score).

Post-Conditions: If the user was actively playing, the screen will return to the main menu. If the user was in the menu screen, the program will close completely.

Use Case Name: View High Scores

Actors: Player

Preconditions: The program must be open and in a Start Game menu screen

Description: When the "View High Scores" option is selected, a list of high scores will appear from highest to lowest. Then the player will be presented the option to go back to the Start Game menu.

Exceptions:
No high scores: If the user has never completed a full run, there will be no high score to display. In which case the leaderboard will display empty slots.

Post-Conditions: The program will remain in the leaderboard screen until the player chooses the option to back to the Start Game menu screen.

Use Case Name: Play Game

Actors: Player

Preconditions: The system must be waiting in the Start Game menu.

Description: When the player chooses Play Game, a new game will begin with the preset difficulty and song options.

Exceptions:
No Difficulty: If the player did not choose a difficulty, the default difficulty will be used.
No Song: If the player did not choose a song, the default song will be used.

Post-Conditions: The program run the game until the player loses, the song is over, or the player exits the game.

Use Case Name: Key Inputs

Actors: Player

Preconditions: The program must be open and running. The program is always waiting for user key inputs

Description: If the player is currently in a game, the key inputs may be used as commands so select what string to play.

Exceptions:
Key not bound: If the key that the player pushed is not one corresponding to a block, the player's score will decrease.
Not in Game: If the player is not currently in a game and a key is pressed, nothing will happen.

Post-Conditions: The program will execute the command related to the key pressed. The program will continue to wait for other key inputs.

Use Case Name: Set High Scores

Actors: Player

Preconditions: The player must finish playing a game with a higher score than the previous high score.

Description: When the player gets a new high score, the leaderboard will update showing the new highest scores.

Exceptions:
Incomplete game: If the player currently has the highest score, but has not finished the game, and decides to exit, that high score will be lost.
Not High score: If the player does not reach or mass the highest score, the leaderboard will not be updated.

Post-Conditions: The leaderboard will be updated, and the program will return to the Start Game menu

Use Case Name: Instructions

Actors: Player

Preconditions: The program must be waiting in the Start Game menu

Description: When the player chooses the Instructions option, pop-up window will appear showing the player how the game is played.

Exceptions:
Not in Start Menu: If the program is not in the Start Game menu, there is no way of accessing the Instructions option

Post-Conditions: The instructions will display until the user decides to exit this window.

Use Case Name: Select Difficulty

Actors: Player

Preconditions: The program must be waiting in the Start Game menu

Description: When the player chooses the Select Difficulty option, the player will be presented with a list of difficulty options he may choose from. When the player chooses an option, the game will change to the according to the selected difficulty (e.g., speed will increase)

Exceptions:
Not in Start Menu: If the program is not in the Start Game menu, there is no way of accessing the Instructions option
No Difficulty Chosen: If the player does not choose a difficulty, a default difficulty will be used.

Post-Conditions: The difficulty level will be saved, and the program will return to the start game menu waiting for the next input

Use Case Name: Select Song

Actors: Player

Preconditions: The program must be waiting in the Start Game menu

Description: When the player chooses the Select Song option, the player will be presented with a list of song options he may choose from. When the player chooses an option, the game will load the selected song and return to the start game menu.

Exceptions:
Not in Start Menu: If the program is not in the Start Game menu, there is no way of accessing the Instructions option
No Song Chosen: If the player does not choose a song, a default song will play

Post-Conditions: The selected song will be saved, and the program will return to the start game menu waiting for the next input

USE CASE TESTING

OPEN GAME

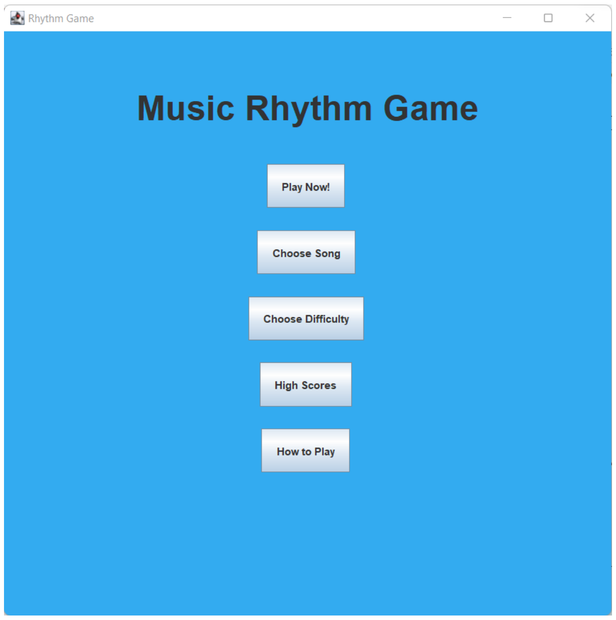
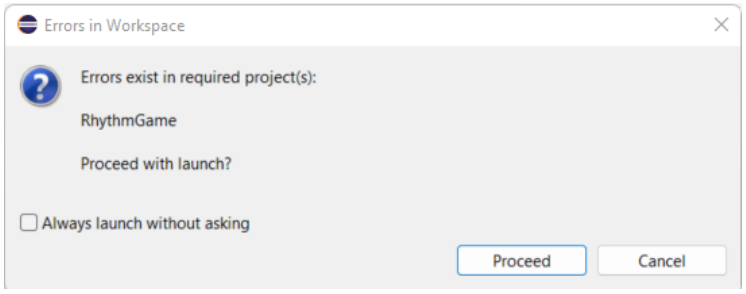


Fig. 1.1 Screenshot of the Main Menu, which appears upon opening the game.



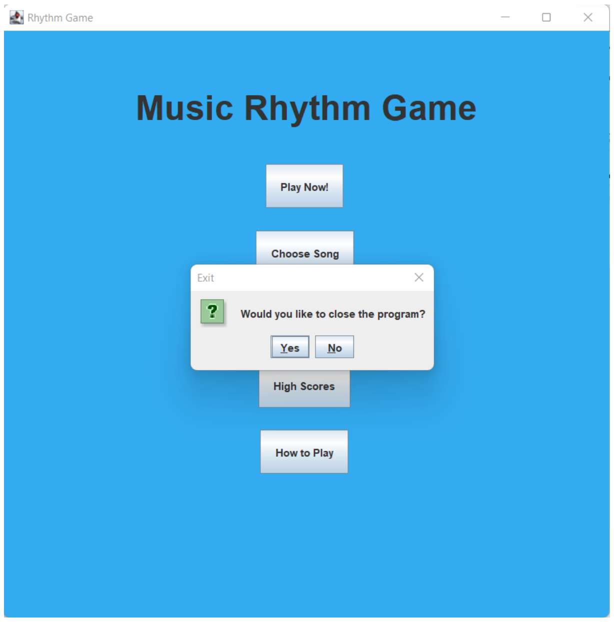
Errors exist in required project(s):

RhythmGame

Proceed with launch?

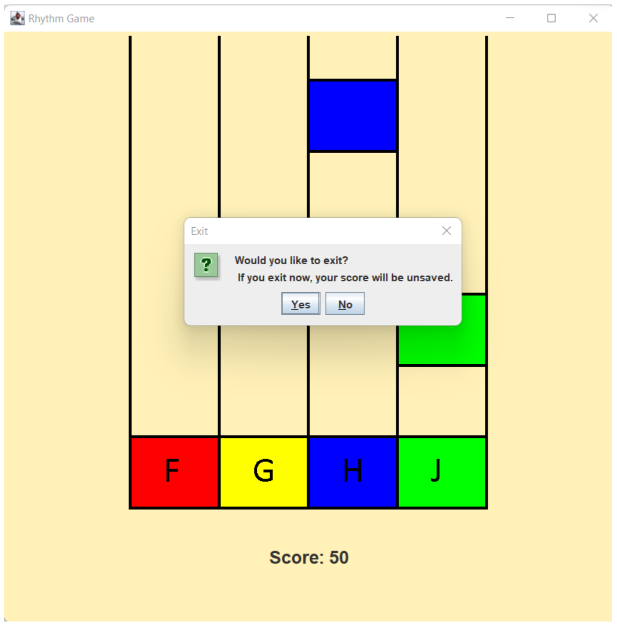
Fig. 1.2 Screenshot of Eclipse's error pop-up. If the files are not correctly loaded into a workspace, the program will not execute.

EXIT GAME



Would you like to close the program?

Fig. 2.1 Screenshot of the "Exit Game" pop-up window when exiting from within a game.

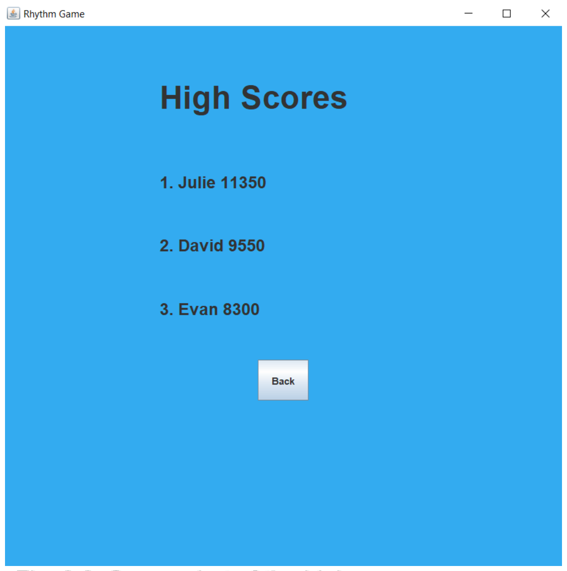


Would you like to exit?

If you exit now, your score will be unsaved.

Fig. 2.2 Screenshot of the "Exit Game" pop-up window when exiting from the Main Menu

VIEW HIGH SCORES



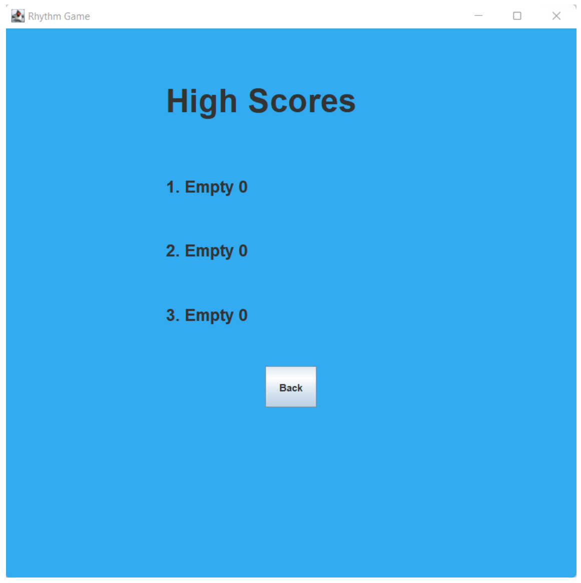
High Scores

1. Julie 11350

2. David 9550

3. Evan 8300

Fig. 3.2 Screenshot of the highscores screen behaving regularly.



High Scores

1. Empty 0

2. Empty 0

3. Empty 0

Fig. 3.2 Screenshot of the highscores screen. This is the way the exception is handled when there are no high scores to display.

PLAY GAME

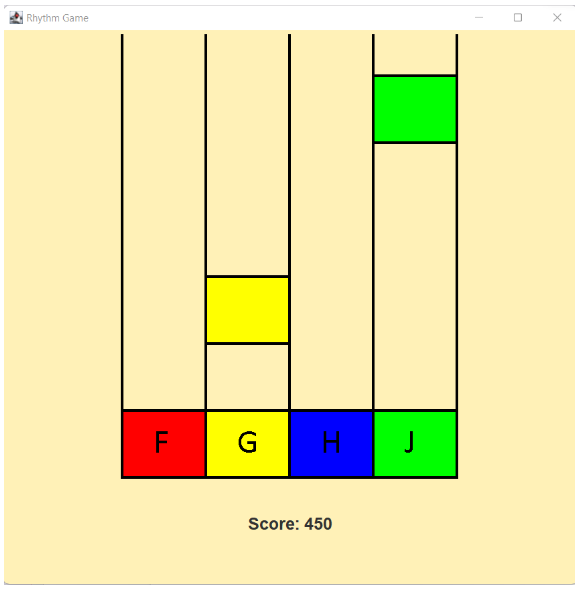


Fig. 4.1 Screenshot of the in-game screen working correctly.

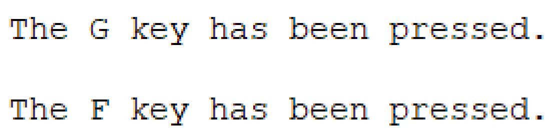
```
Using Defaut song: "sayItAintSo"
```

Fig. 4.2 Screenshot of the Eclipse console. This shows how the “No Song Chosen” exception is handled; a default song will play.

```
Using Defaut difficulty: hard
```

Fig. 4.3 Screenshot of the Eclipse console. This shows how the “No Difficulty Chosen” exception is handled; a default difficulty will be used.

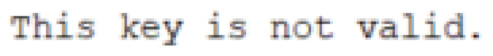
KEY INPUT



```
The G key has been pressed.

The F key has been pressed.
```

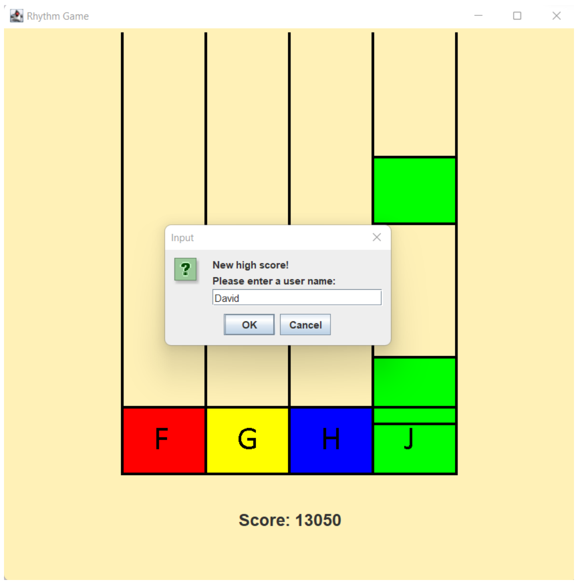
Fig. 4.1 Screenshot of two outputs in the Eclipse terminal. This shows how key inputs are being received by the program correctly



```
This key is not valid.
```

Fig. 4.2 Screenshot of the Eclipse terminal. This shows how the "Key not bound" exception is recognized by the program.

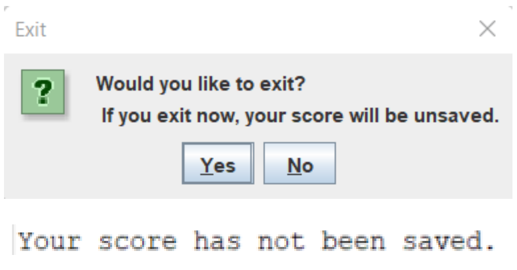
SET HIGH SCORE



New high score!
Please enter a user name:
David

Score: 13050

Fig. 5.1 Screenshot of the "New High Score" pop-up window that appears at the end of a game when the player has reached a new high score.

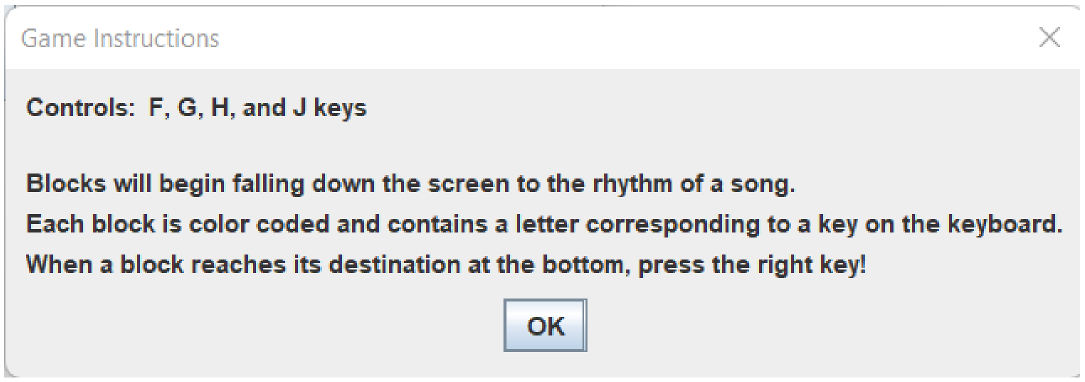


Would you like to exit?
If you exit now, your score will be unsaved.

```
Your score has not been saved.
```

Fig. 5.2 Screenshot of the "Are you sure you want to exit" pop-up window, as well as an Eclipse console output. This illustrates how the "Incomplete Game" exception is handled within the program.

INSTRUCTIONS


Game Instructions

Controls: F, G, H, and J keys

Blocks will begin falling down the screen to the rhythm of a song.
Each block is color coded and contains a letter corresponding to a key on the keyboard.
When a block reaches its destination at the bottom, press the right key!

Fig. 6.1 Screenshot of the instruction window accessed from the Main Menu

SELECT DIFFICULTY

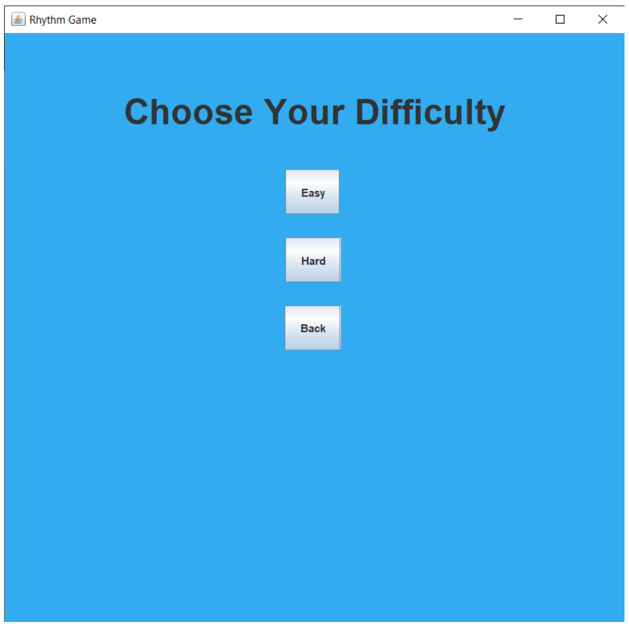


Fig. 7.1 Screenshot of the "Select Difficulty" menu accessed from the Main Menu. Fig 4.3 illustrates how exceptions for this case are handled

SELECT SONG

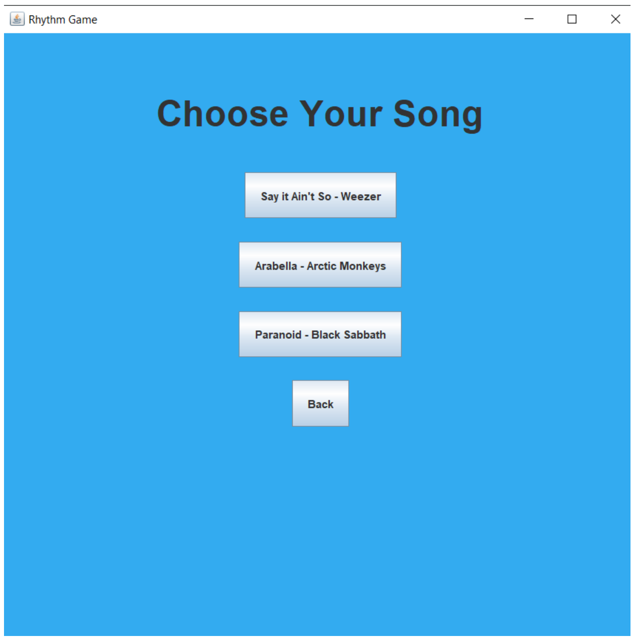
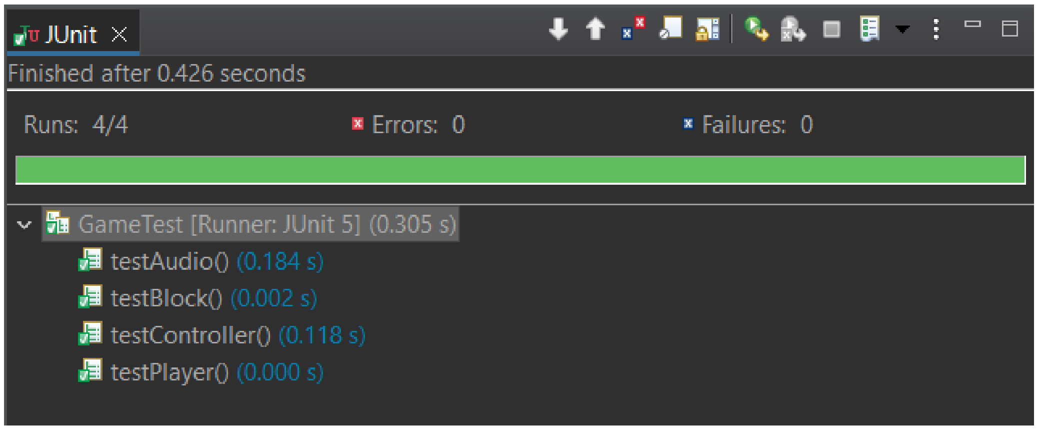


Fig. 8.1 Screenshot of the "Select Song" menu accessed from the Main Menu. Fig 4.2 illustrates how exceptions for this case are handled

UNIT TESTING

A JUnit file was used to test individual classes and their methods. All classes were tested, excluding the GameDriver and Screen classes. The GameDriver and Screen classes are used for the implementation of the GUI, so it was more appropriate to evaluate them during the use case testing. Otherwise, all other classes and their methods were verified in the JUnit test.

JUNIT TEST OUTPUT



```
Finished after 0.426 seconds
Runs: 4/4        Errors: 0        Failures: 0
GameTest [Runner: JUnit 5] (0.305 s)
    testAudio() (0.184 s)
    testBlock() (0.002 s)
    testController() (0.118 s)
    testPlayer() (0.000 s)
```

Fig. 9.1 Screenshot of the JUnit test output. All class tests passed every case with no failures.

To run this test yourself, you can right-click the "GameTest.java" file in the RhythmGame.Debug package and select Run As -> JUnit Test. This will run the JUnit file and provide you with the output shown above.

Revisions

Over the span of this project, there were many enhancements and corrections made to the game. The final iteration features the addition of graphics, colors, and bug fixes. Our milestone review comments requested more visual user feedback, greater complexity, and more comments in the code. Below are the items that were added in the final version of our project in response to these comments:

- Graphics and the animation of falling blocks were added to provide a more visual and enjoyable user experience.
- Background colors were provided to give a more comfortable feel.
- Block borders now light up when the user presses the correct key. This is to provide more feedback to the user.
- Lingering issues with the game's timer, which caused numerous frustrating bugs, have been fixed. These issues were affecting the scoring, the speed of the falling blocks, and the amount of times popup windows were appearing. These issues are no longer present in the final version.
- The DestinationBlock and MovingBlock classes have been removed from the project, and the Block class is no longer abstract. While learning about generating graphics, the team decided to simplify and restructure these classes, as the graphics required for the program work differently than what was previously thought.
- A third song was added to the game to give more variety.

Project Challenges

While implementing the required materials for this project, numerous challenges had to be overcome. With only a rudimentary understanding of Java GUIs and animations, the team had to learn how to move an unknown amount of falling blocks down the screen while also implementing logic for which keypress each block corresponded to. This challenge was solved by randomly generating rectangles of different color/letter combinations and adding them to an arraylist. The program would then iterate through the list and generate the graphics and logic for each block.

Another challenge that the team faced was adding sound to the project. This was something that no one in the group had experience in. Quite a bit of background work went into the creation and execution of audio files. This included generating the files properly in Java, calculating the song lengths, and determining when the songs would need to start/stop and pause. Additionally, the timing of the blocks needed to be close to the beat of a song. This took a lot of trial and error in implementing the timer class to get good results.