Space Hangman Game

Arianna Brown, Xochitl Arriaga, Aisis Julian, Warisa Siewsrichol

Table of Contents

Project Requirements

The purpose of this project is to create a functioning hangman game that supports both single-player and multiplayer gameplay. In order to allow multiplayer gameplay, the game will use socket programming to create a server that supports multithreading. This way, multiple clients will be able to connect to a server and play games. In order to make our game look more appealing, we will use JavaFX to provide a graphical interface for our game. Hangman is a visual game, so we feel implementing a visual aspect to our game is very important.

Much like a traditional hangman game, we will start with a ‘base’ image. Instead of a man, our base image is a spaceship. Players can choose to play single-player or

multiplayer games by clicking buttons. If a multiplayer game is chosen, players then must choose between a 2-player, 3-player, or 4-player game. Finally players must choose a game difficulty: Easy, which will consist of words that are less than or equal to 4 letters, Medium, consisting of words between 5 and 7 letters, and Hard, consisting of words greater than 7 letters.

Once a game has started, player must use the JavaFX keyboard we will create to choose a letter by pressing the button labeled with that letter. Once a letter is chosen, that letter's button is disabled for the rest of the game to prevent a user from guessing the same letter twice. Each time a player makes a wrong, guess, we add another component to our spaceship image. Each user can make 6 wrong guesses before they lose the game. If a user guesses makes 5 or less wrong guesses and is able to guess the word, they win. After a game is over, users will be taken to another screen which will display whether the user won or lost. They then will have the option to play again or exit.

Design Philosophy

For server-side design, we want to keep our graphical interface simple. Clients handle game play as well as most of the game logic, so all we need for our server is a way to import the desired port as well as an on and off button. Our graphical interface is heavily on the client side. There are a few key things that are necessary for providing a functioning and enjoyable experience for users:

- A welcome screen that displays the user's choices: Single-player vs. multiplayer, 2-player, 3-player, and 4-player games, and Easy, Medium, and Hard game modes
- A game scene that displays a keyboard the users can interact with to guess a letter
- A game scene that displays letters guessed correctly
- A game scene that displays the spaceship image, and correctly adds a new component to the image for each wrong guess
- A game over scene which displays whether the player won or lost

Entities

Client

- Purpose: A simple class that keeps track of the client's IP and Port. Extends ClientConnection class
- Methods

    - getIP()
        - Returns a String object for the IP for that client
    - getPort()
        - Returns and int object for the port for that client

Client Connection

- Purpose: Main driver behind client-side connection. Accepts serializable data from server as well as sends server data
- Methods
    - send()
        - Sends serializable data to the server
    - closeConn()
        - End the connection with the server
    - getIP()
        - Abstract method for getIp() in Client class
    - getPort()
        - Abstract method for getPort() in Client class
    - clientConnect()
        - An infinite loop that accepts and reads incoming data from server

ClientFX

- Purpose: The bulk of client-side program. Handles all JavaFX code as well as most client-side game logic
- Methods
    - start()
        - Set the primaryStage scene to the startScene. Also contains a few event handlers for the ‘connect’ button and ‘start’ button. When the ‘connect’ button is clicked, create a new client thread.
    - createClient()
        - Pass in a runnable as a parameter which allows the client to read in data from the client-server thread while still updating JavaFX elements in the application thread. This is where the bulk of server message interpretation happens. Depending on the data the client receives, different JavaFX functions are called.
    - startScene()
        - Sets up all JavaFX components in start scene
    - waitScene()
        - If a user selects a multiplayer game, they are sent to the waitScene until more players have joined the game and the game is ready to

        - start. This method handles all JavaFX components for the waitScene scene.
    - GameScene()
        - Sets up all JavaFX components in the gameScene scene
    - initWordDisplay()
        - To display the word and letters users have guessed correctly, we implemented a JavaFX to display each correctly-guessed letter as a block on the screen. This method sets up the wordDisplay component as a blank display.
    - updateWordDisplay()
        - Every time a user guesses a letter correctly, wordDisplay must be updated to reflect correctly guessed letter
    - initSpaceshipImage()
        - Creates and displays the initial ‘base’ spaceship image for the game
    - updateSpaceshipImage()
        - Every time a user guesses wrong, creates and updates spaceship image by adding one new component to the image
    - initKeyboard()
        - Initializes an interactive keyboard that displays each letter as a button
    - enableKeyboard()
        - Enables all buttons on the keyboard
    - disableKeyboard()
        - Disables all buttons on the keyboard
    - EndScene()
        - Initializes and displays the endScene, the scene that is called once the game is over

Server

- Purpose - sends and accepts from client, makes an instance for each client thread and most of the game logic
- Methods
    - Server()
        - Set port and initialize in the dictionary
    - initDictinary()
        - Inputs the text file, make all letters uppercase, input words in different dictionaries based on the length of words
    - getPort()
        - Returns the port

- startConn()
  - Start the connection with the client, updates the number of clients
- send()
  - Send serializable data to the client
- broadCast()
  - Sends every client the same message
- closeConn()
  - Close connection with client
- class ClientThread - inner class of Server to make instances of client thread
  - ClientThread()
    - Takes in a socket to make a connection through server
  - run()
    - Create the input and output streams, only runs if the client guesses a letter, starts the game if each player chooses the same level of difficulty, sets and shows the letter each client chooses
  - findGame()
    - Returns a game when the client is active and chooses the same level of difficulty
- Class Game - inner class of server that holds the game logic
  - Game()
    - Takes in client thread and makes an instance of the player, sets the difficulty and number of players
  - removePlayer()
    - Takes in Client thread and removes player in the array, updates number of players connected
  - addPlayer()
    - Takes in client thread and adds player to the array of players
  - startGame()
    - Makes an instance of an array of boolean for letters guessed, chooses a random word from dictionary based on level of difficulty and send it client, remove duplicate words
  - resetGame()
    - Clear array of players and update number of players to 0
  - evaluateGuess()
    - Takes in the letter the client pressed, see if the the character exists in the word, removes letter as a choice, checks if the

player wins, ie guessed all the characters, check if the player loses, if all lives have been used

- removeDup()
  - Removes letter if it in the word multiple times
- checkForLoss()
  - If the player has less than 0 lives, send client index for which player lost

ServerFX

- Purpose - Holds most of the GUI elements for JavaFx and some of the game logic for the server
- Methods
  - start()
    - Makes an instance of the server display, set the action for the server off button by closing the connection and displaying a different scene, set primary stage to start scene
  - class serverDisplay - inner class of serverfx to initially display the scene for the server
    - serverDisplay()
      - Set the background for the GUI, and on and off buttons for the server, the port box to input, and all the labels for each element including the header
    - displayServerOn()
      - Makes the port input and label not visible, and moves the buttons towards the top
    - displayServerOff()
      - Makes all ports and labels to be visible and resets the server to original display
    - createServer - event Handler
      - Gets the port that was in the input, create a new server socket, set the server to be on. If creation failed, the server is not on
      - Makes a task to display number of clients connected to the server and start connection
      - Makes a new thread with the task created
  - Class clientDisplay - inner class of ServerFx used to display the inclient information once the server in on
    - clientDisplay()

- Updates the server display if the client is connected, display the GUI elements of the scene such as the different labels of the screen elements, and sets the client index

Benefits, Assumptions, Risks/Issues

One benefit to our program is that the game is easy to play. A difficulty all of the group members found with Project 2, the Pitch game, was that many people did not know how to play the game. This provided additional issues when creating the game because not only did group members have to handle game logical andiffed graphical interfaces; They also had to ensure they understood the rules of Pitch enough to correctly implement the game logic. All of the group members already know how to play Hangman, so this is a hurdle we did not have to overcome when implementing game logic with this project.

One risk is that there are most likely hundreds, if not thousands, of hangman games already created and available for use on the internet. As a group, we tried to overcome this hurdle by choosing to go with the more unique theme of alien hangman versus regular hangman. We felt that this provided a more creative, fresh take on the game and would provide an opportunity to create a more niche, immersive visual experience than just regular hangman.

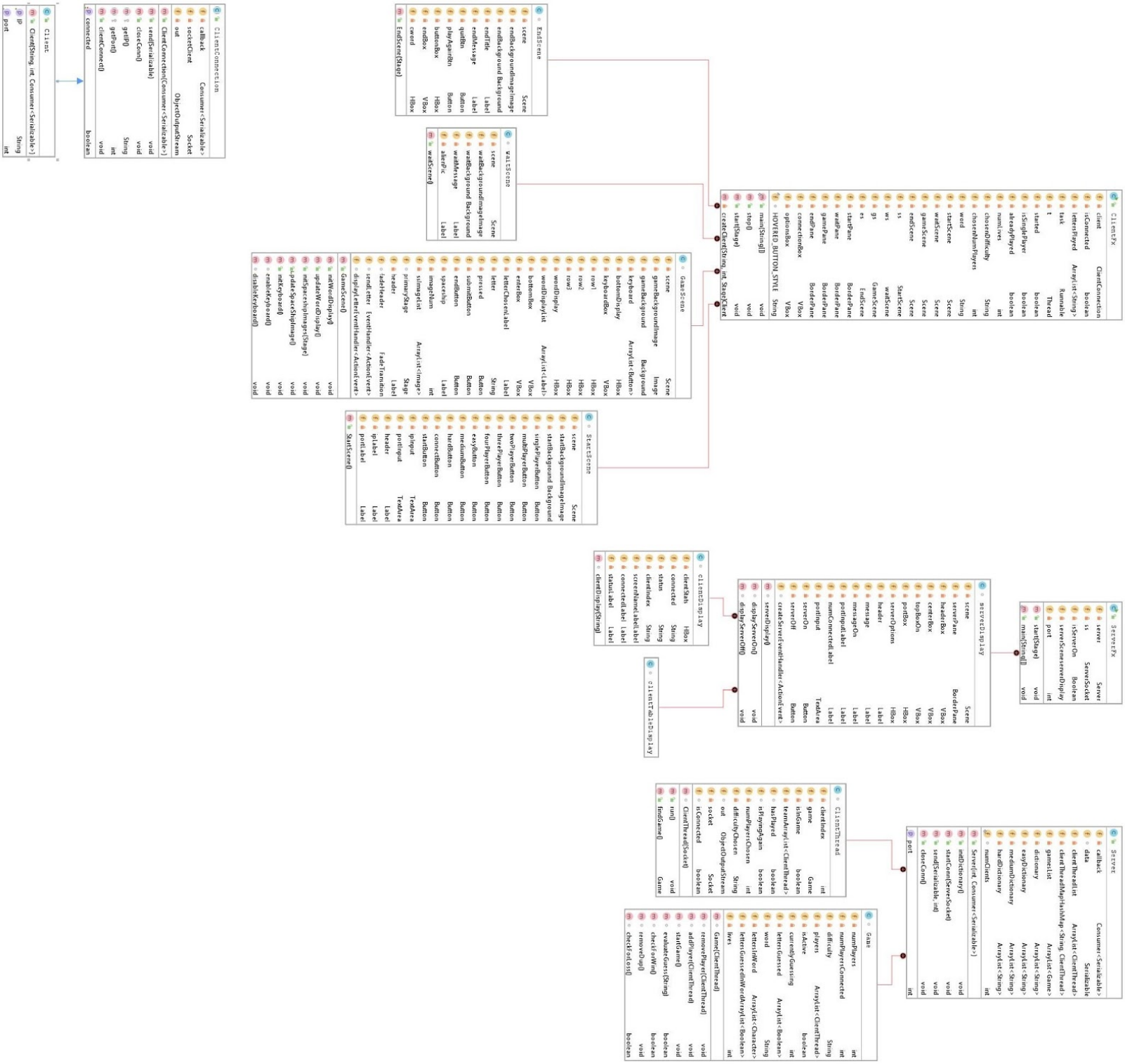
Test Plan

The graphical representation of our game is just as important as the game logic, and we plan to test the JavaFX components of our program to ensure a natural flow as well as the game logic. One element to pay attention to is ease of use: Is the keyboard fun to use? Is it annoying? Do buttons enable and disable at appropriate times in the game?

Of course, game logic must be tested and tested again. The key points of our game logic are:

1) Does the server select a word that matches the difficulty the user selected
2) Is the word the user must guess correctly sent to each client in the game
3) Does the wordDisplay update when a user makes a correct guess
4) Does the spaceship image update when a user makes an incorrect guess
5) Does the game end when the users have run out of lives
6) Does the game end when users have correctly guessed the word
7) Once the game ends, is the endScene displayed

UML Diagram



Activity Diagram