Promised Land Journey Game
Developed by Robbie Errico and Oussama Rahmouni
For Ancient Path Adventures, Talpha Harris

Software Design Document

Github Repository

Repository Link: https://github.com/ousrahm/Promised-Land-Game

Website Repository Link: https://github.com/ousrahm/Promised-Land-Journey-Website

Phaser 3.0

Phaser 3.0 was used to create this project.

Phaser Website: https://phaser.io/phaser3

Phaser Documentation: https://photonstorm.github.io/phaser3-docs/index.html

Firebase Documentation

Firebase Realtime Database is the only database for this project.

Firebase Hosting is used to host this project:
https://promised-land-journey-game.web.app/

Firebase Setup Instructions: https://firebase.google.com/docs/web/setup

Firebase for JavaScript:
https://firebase.google.com/docs/reference/js/firebase.database.Reference

Firebase Console is used to manage Hosting and Realtime Database:
https://console.firebase.google.com/

Files & Software Architecture

All questions are stored in a single csv file called questions.csv, with each question making up a row of the csv file with the following categories:

- Question: Text for the question
- Correct Answer: Text for the correct answer
- Wrong Answer (x3): Text for each of the three wrong questions
- Stage: Which stage this question should appear in
    - Note: This functionality is unused in the current version of the game due to not having enough questions to give each stage its own unique pool of questions, which is why all questions have been placed in stage 0.

The index.html file represents the file that automatically loads whenever a user opens the game website. The index file loads fonts, jQuery, a jQuery CSV extension, and

Firebase-specific resources. It creates the HTML textboxes, buttons, and checkboxes that are needed for the Naming scene, the Join scene, and the backgrounds setting. Lastly, it loads all of the JavaScript files that make up the game, including a compressed version of Phaser 3.0.

The scenes folder holds the JavaScript files for all of the classes that implement Phaser scenes in the game, as well as the GameState and Questions classes, which handle the state of the game and the questions, respectively.

When the game first loads, the game.js file needs to operate with the existence of the rest of the Phaser scenes already loaded, and so it must always be listed at the bottom of the JavaScript files. In game.js, a connection to the Firebase Realtime Database that stores the game objects is established. A configuration object is created to set the parameters for the game window and scenes. Multiple listeners for the HTML elements described above are included in game.js. An instance of GameState is created and an instance of Questions, using the CSV on file, is created from game.js. Once the window loads completely, a new instance of the Phaser.Game class is created with the configuration object passed in as a parameter.

Because super("bootGame") is called in the LoadingScene class from the loadingScene.js file, that will be the first scene to open after the Phaser Game is created. An instance of the LoadingScene class is created using the reference, "bootGame," meaning, to open the LoadingScene, one must use this command in another scene: this.scene.start("bootGame"). The string that is provided to the super() call in the constructor of every scene class defines the name by which to open that scene.

Upon loading a new instance of a scene class, there are three functions that will be automatically be called:

1. preload()
    a. Called one time as soon as the scene loads
    b. Used to load pictures, videos, and other elements for use in any scene using "this.load…"
        i. It is recommended to load elements in a scene prior to the scene(s) in which those elements will be displayed.
2. create()
    a. Called one time as soon as the scene loads
    b. Used to add loaded pictures, videos, and other elements using "this.add…"
3. update()
    a. Called 60 times per second on average
    b. Can be used to add loaded pictures, videos, and other elements or adjust those elements sizes and positions

Throughout playthroughs, the game alternates between various *scenes* to provide different functionality based on what is goin on. Players' game engines begin by creating an instance of the menuScene class, where they can choose to start a game. From there, depending on whether they choose Host or Join, they will enter hostScene or joinScene, respectively. In hostScene, Player 1 will enter their name and start a game, which will trigger the creation of a database for that game's playthrough, which will then take them to lobbyScene. In joinScene, players will join an existing host's game by entering the unique four-letter code for the game they wish to join; if the code they enter is able to be found in the database, they will join the host in lobbyScene, where they will be assigned a player number 2 through 4 based on what order they join. When all players have joined and have assembled in lobbyScene, the host can then begin the game, which takes all players to triviaScene. This scene is the main scene where gameplay takes place, drawing a question and four answers, which it only permits the player whose turn it is to answer. Should the player select the correct answer, they are taken to correctScene, which reveals that they got the question correct. From there, if the player has answered three correct questions in their current stage, they are moved to newStageScene, which notifies them of which stage they are moving onward to. While here, the game checks to see whether that stage has been reached by any other players yet. If it has not, the game moves to storyLineScene, which will play a storyline (although because the storylines for this game have not yet been given to us, this scene has been skipped in the current version). After the storyline has either played through or been skipped by the host, the game goes to nextPlayerScene, changing whose turn it is to the next player in line. If the stage HAS been reached previously in newStageScene, then the game simply goes straight to nextPlayerScene from there. If the player gets an answer incorrect, they go from triviaScene to incorrectScene, which reveals the answer the player selected to be wrong and shows which answer was the correct one, continuing to nextPlayerScene afterward. From nextPlayerScene, the game continues back to triviaScene, and this cycle continues, making up the majority of the gameplay. At the end of the final player's turn each round, the game checks to see if any players have won; if EXACTLY one player has won, players will go from newStageScene to victoryScene, where the game will congratulate the winning player before taking players back to menuScreen on click of a reset button. If multiple players have won, the game instead enters tieScene, which proclaims which players have tied before bringing them back to triviaScene to compete in a knockout tiebreaker round, which transitions between scenes in the same way as before. If there are still multiple winners at the end of 5 cycles, the game is proclaimed a true tie and proceeds to trueTieScene, which congratulates all winners who have made it that far.

<u>Database Structure</u>

- Four-letter Game ID
    - Player number: number of players to play in current game
    - P1: Player 1's in-game name

- P2: Player 2's in-game name
- P3: Player 3's in-game name
- P4: Player 4's in-game name
- turn: keeps track of which player's turn it is
- joined: keeps track of the number of players that have joined the game
- started: marks whether the game has started
- retrievedQuestion: marks whether the host has generated a question so joined players can retrieve questions from the database
- question: text for the current question
- answers: set of texts for each of the four answers
  - A: text for answer A
  - B: text for answer B
  - C: text for answer C
  - D: text for answer D
- correctAnswer: letter of the correct answer
- selectedAnswer: which answer has been selected by a player
- skippedStoryline: tracks when the host player skips the storyline

Multiplayer & Data Synchronization

A GameState class is used to coordinate the functionality of the game. The GameState keeps track of every part of the game besides the questions. Questions are kept track of in the Questions class. An instance of GameState is created for every player as soon as they join the game. Every instance of GameState on every player's front end will be identical except for the *myPlayer* property, which keeps track of what that front end user's player number is (the host/player1 is 0, player 2 is 1, etc.).

By including database node listeners in various places throughout the front end code, multiple devices can stay synchronized by displaying scenes at the same time. The code in each player's front end scenes keeps their local instance of the GameState class up-to-date. Through the database node listeners in various places throughout the front end code, multiple devices stay synchronized by displaying scenes at the same time.

Information is synced through Firebase Realtime Database. See the database schema in Database Structure. Instructions on how to set up Firebase Realtime Database and Firebase Hosting are linked in Firebase Documentation.

Every time a player joins the lobby, the number in the *joined* field of the database is incremented by 1. When *joined* reaches the correct number of players (2, 3, or 4 depending on how many players the host has chosen), a listener will reveal a READY button for only the host's lobby scene. Once the host presses this READY button, the host's front end will modify the *started* field in the database from **false** to **true**. All

players, including the host, listen for this *started* field to change before opening the Trivia scene for the first person simultaneously on everyone's front end.

When it is time for a player to answer a question, the host's front end, referred to here as the host, populates the *question* and *answers* fields in the database with a question and four answers. The host also populates the *correctAnswer* field with the correct answer to that question. Once the host adds this question to the database, the host will change *retrievedQuestion* in the database to **true**. All other players listen for *retrievedQuestion*'s value to change to **true** before retrieving the question from the *question* field and answers from the *answers* fields in the database. This occurs before every player's trivia question, regardless of whether it is the host player's turn or not. Players' names are sent to the database following the Naming scene and are stored locally in every players' front end. The turn field in the database is used to coordinate the game engine with the database, and is adjusted only by the host player's front end.

When a player selects an answer for a question, that player's front end will modify the *selectedAnswer* field in the database, changing it to the text for that answer. All players listen for *selectedAnswer*'s value to change before calling the a function that will compare *selectedAnswer* to *correctAnswer* and opens the Correct or Incorrect scene accordingly. Code in Correct and Incorrect scenes adjusts the various counters for stages, number of questions answered correctly, etc. in the instance of the GameState class. On the host's front end, the code will reset *selectedAnswer* and *retrievedQuestion* back to **false** and an empty string, "", respectively.

When the Storyline scene is displayed, a SKIP button will only appear for the host. All players' front ends are listening for the *skippedStoryline* field to be changed from **false** to **true** before opening the next scene simultaneously. (Note: The storyline functionality was removed from the most recent version of the game due to not having received the proper cutscenes that should be displayed.)

The listeners that are placed in the lobbyScene, TriviaScene and the StoryLine classes keep the game synchronized on multiple devices.

Design Choices

We decided to go with Firebase Realtime Database instead of Firebase Cloud Firestore Database due to the simplicity of our database needs (more information on Firebase is included in the Firebase Documentation section). We did not need advanced querying, sorting, transaction-making capabilities from our database. As it says on the Firebase website (firebase.google.com/docs/database/rtdb-vs-firestore), "[Realtime Database is] an efficient, low-latency solution for … apps that require synced states across clients in real time," which exactly described our needs for this project.

We chose to design the game so that the host's front end would be making the majority of the calls to the database. By limiting other player's interactivity with the database, we minimized the number of calls that would be made during a game and enforced simplicity in the software design. All players, including the host, have functions that listen for changes in the database, so that as soon as the host makes a change, all of the listener functions in every players' front end would be triggered at the same time.

A very important note to remember is to always turn off database listeners after a player leaves a scene. Without turning off the listener, the listener will continue to trigger in a previous scene while the game has moved past that scene.

We had originally hoped to have pools of stage-specific questions for each stage. Due to not having enough questions to create large pools for each stage, all questions have been placed at stage 0 in the Questions class. Functionality to make questions stage-specific is commented out on lines 21-30 of the Questions class in the questions.js file.

We have created a file and class for implementing storylines after a player has been the first to reach a stage (StoryLine class in storyLineScene.js). Due to a lack of content to place in these scenes, we have removed them from the flow of the game. The file still remains in the repository. The script tag for the file remains in the index.html file but is commented out. All calls made to open the StoryLine scene have been commented out.