Team 22 : Memoji

Design Document

Team Members

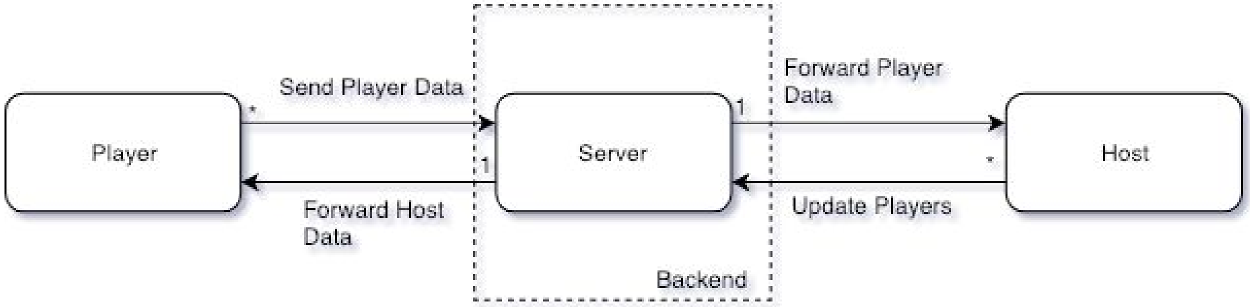
Jonathan Poholarz (jpoholar@purdue.edu)
Maxwell Jones (jone1268@purdue.edu)
Manoj Polisetti (mpoliset@purdue.edu)
Andrew Ring (amring@purdue.edu)
Delun Shi (shi272@purdue.edu)

Purpose

As proven by the success of the Jackbox™ Party Packs©, there is a market for engaging multiplayer experiences incorporating digital technology into traditional sit-down board game entertainment. We believe there is an opportunity here to develop additional games which delve into this fairly unexplored design space incorporating users' phones as controllers for a game hosted with a computer. We are designing a Jackbox™ style party game with the main theme revolving around emojis in order to provide entertainment to friend groups and families with the goal of being as accessible and entertaining as possible. Utilizing a short 4 letter code displayed on the computer will allow players to easily connect to the host computer using their own computers or mobile devices. Without requiring the complex networking details such as IP addresses, we hope to make our game more accessible and easy to jump into for a diverse audience.

Contents

Design Outline



The overall design of our game system can be broken down into three interacting parts. We will have the Host computer managing the game session and the Player devices participating in a game session. Both of these groups function as clients in a client-server model with our Server component.

The Player class and the Host class interact with the Server class and not with each other directly. A Player can send answers, vote, choose a username and avatar, and exit the game by communicating to the Server which then relays the actions to the Host the Player is connected to (using the lobby code the player connected to). The Host receives information from the Server and can react accordingly. Similarly, the Host can send information to the Server such as new prompts for players to answer, answers for players to vote on, or if the round of gameplay has ended. The Server forwards this information to the respective Players connected to that Host's lobby code.

- Player - each individual connection to a game session which answers prompts and submits votes on answers
  - Player will be uniquely identified by a UserID in the host's Player dictionary
  - Player will be uniquely identified by a UserID and LobbyID on the server
- Host - the game session which generates prompts for the players to answer and controls the overall game logic
  - Host will be uniquely identified by a 4 character room code (LobbyID) generated by the server
  - Host handles game logic and forwarding state changes to client players.
- Server - the online networking component which directs net traffic between the hosted servers and the connected players but does not process game logic
  - Stores Host IP/Port and Room Code pairs
  - Maintains a list of Players connected to each Host as well as the Player IP/Port

Design Issues

Functional issues

1. Do we need an account system for players?
    a. **Option 1: Users do not need to go through any account creation or login steps**
    b. Option 2: Users need to create accounts to host and play games

    Decision: Fundamental to this style of game is its ease of access in which only one player needs to own the game and anybody can join quickly from their phones. For this reason, we will not be using an account system. In practice, users would not be able to host without purchasing the game (which then allows the download of the actual hosting software).

2. Do the users use their own mobile keyboards to input emojis, or do we implement our own method of input?
    a. **Option 1: Include an input method exclusive to the application**
    b. Option 2: Use the mobile keyboard

    Decision: We decided to use our own implementation to input emojis. This will allow us to avoid conflicts due to unicode across different devices and the difference in the selection of emojis offered between iOS and Android devices. Different devices or platforms render emojis differently, and must avoid ending up with a series of rectangles. Instead, we will use our own input system which will allow for more direct customization by our development team and bypass the unicode difficulty by rendering our own set of emoji images consistently across platforms.

3. Should the player device have a timer displayed on it which counts down until there is no more time to answer prompts?
    a. Option 1: Include a countdown timer on each player device.
    b. **Option 2: Only display a timer on the hosting screen.**

    Decision: There will only be a timer on the host screen because it is assumed that players will all be in the same room as the screen or the screen is available to be viewed online. Even without a timer on their own devices, players should be able to see the countdown. We have chosen this option mainly because syncing the timers across multiple devices may pose notable challenges with the networking system involved, and the players do not technically need to have this timer on all devices.

4. How should we handle networking between the host computer and phones?
    a. Option 1: Use Godot directly, having the host computer host the server.
    b. Option 2: Create a separate server.

    Decision: By using a separate server, it will be easier for hosts to set up game sessions as they will not need to port-forward from a personal router in order to host the game. Instead, all information can be routed through the server, which reduces the setup barrier to entry and makes the game more accessible to less tech-experienced players. The separate server will also allow us to create a simpler structure for our networking API.

5. How should we handle more people wanting to join when the lobby is already full of players?
    a. Option 1: Not let them connect.
    b. Option 2: Have them join the "audience."
    c. Option 3: Add them to a queue to waiting players to join later games.

    Decision: Adding additional clients will put them into the audience pool. This additional grouping of players will be able to contribute to voting sessions but not answer the prompts directly. Adding this extra group accommodates groups of players that exceed our game limit, again supporting our goal of developing a fun game for as many players as possible.

6. Should there be a limit to the emojis that can be placed on the canvas.
    a. Option 1: Limit number.
    b. Option 2: No limit number.

    Decision: Because we plan to send our emoji answers across our networking setup, we do not want players to be able to abuse our system by spamming emojis. This would potentially result in costly net traffic for our server(s). In most cases, the players will not need to go over the generous limit we provide them of around 25 emojis. They should be able to convey their answers within this limit.

7. Should the players who answered the prompt be able to vote for the winner of that prompt?
    a. Option 1: Yes.
    b. Option 2: No.

    Decision: In most cases, players will only vote for themselves when given the option. Voting for the other player could cost someone a round and isn't worth the risk in the case of close voting. Instead of this, as votes will most likely translate directly into points, we can instead have the players who authored the answers vote for something else trivial such as which player they believe is going to earn the most votes.

Non-Functional issues

1. What game engine should we use?
    a. **Option 1: Godot**
    b. Option 2: Unity

    Decision: We decided to use Godot over Unity because most of our team does not have experience with Unity, and, compared with Godot, the entry learning curve for Unity is much steeper. Godot will be easy to learn, requires much lower hardware specs to develop in, is a significantly more lightweight platform, and offers great source control integration. We also expect it to be much easier to iterate ideas in than Unity.

2. What Backend should be used?
    a. **Option 1: NodeJS**
    b. Option 2: Python

    Decision: We decided to use NodeJS to set up the server because we would not be storing any information permanently and will be using the server to just route data from the host and the client to each other. After discussion, we felt that our team's experience in NodeJS would also support this decision.

3. What platforms will the game support?
    a. Option 1: Windows
    b. Option 2: MacOS
    c. Option 3: Linux
    d. Option 4: Android
    e. Option 5: iOS

    Decision: All listed platforms will be exported to and supported. Accessibility is extremely important to this format of party games, so as many platforms as possible should be supported. Godot is capable of exporting to many platforms, so we expect to support all of the most common systems.

4. What graphical form should we utilize?
    a. **Option 1: Pixel-based Images**
    b. Option 2: Vector-based Images

    Decision: Although vector-based images work well with varying resolutions such as the diverse set of mobile device screen resolutions, pixel-based images are easier to code. The added complexity of using vector-based images does not seem to outweigh the extra effort that would be required to create those assets and the extra effort needed to integrate them into Godot.

Functional Requirements:

1. **As a player**, I would like to connect to a game lobby using my phone through the game application
2. **As a player**, I would like to choose my in-game username and avatar emoji
3. **As a player**, I would like to quit the game in-between rounds if desired, releasing my spot in the player list for a new player to join
4. **As a player**, I would like to receive instructions and prompts from the host computer screen during play
5. **As a player**, I would like to view a prompt response screen on my device for each assigned prompt in each round
6. **As a player**, I would like to view a voting screen on my device for each round of prompt voting
7. **As a player**, I would like to view a waiting screen on my device in-between tasks in which the game requires no further inputs from me
8. **As a player**, I would like to input emojis in my responses to prompts easily via a straightforward GUI
9. **As a player**, I would like to edit my response before submission - adding, moving, or deleting emojis with the GUI
10. **As a player**, I would like to vote for answers to the prompts using the GUI
11. **As a player**, I would like to select multiple answers during the final round when needed
12. **As a player**, I would like to view the results of the votes on the host screen
13. **As a player**, I would like to view round summaries on the host screen
14. **As a player**, I would like to view visual confirmation on the host screen when my inputs and votes are properly received
15. **As a player**, I would like to reconnect to a running game after disconnecting and be able to continue playing as the same player with the same username and same avatar
16. **As a player**, I would like to experience a diverse set of prompts such that at least two consecutive games have little overlap in the prompts chosen
17. **As a player**, I would like to view the remaining time for a question on the host screen
18. **As a host**, I would like to create a game lobby and display the lobby's server generated letter code on screen
19. **As a host**, I would like to begin the game once enough players have joined the lobby
20. **As a host**, I would like to display a lobby screen while waiting for players to join
21. **As a host**, I would like to display a waiting screen while players are inputting answers to prompts
22. **As a host**, I would like to ignore/skip players who have not submitted answers to prompts within the allotted time and continue the game
23. **As a host**, I would like to display a results screen for each prompt of each round
24. **As a host**, I would like to display a total results screen at the end of each round
25. **As a host**, I would like to display a final results screen and winner/credits screen after the game completes
26. **As a host**, I would like to play additional games after the first game without forcing all players to disconnect and reconnect to a new lobby

27. **As a host**, I would like to exit the game properly, disconnecting all players and blocking further connections to the defunct game lobby
28. **As a host**, I would like to allow audience members to join at any time after the main players have connected

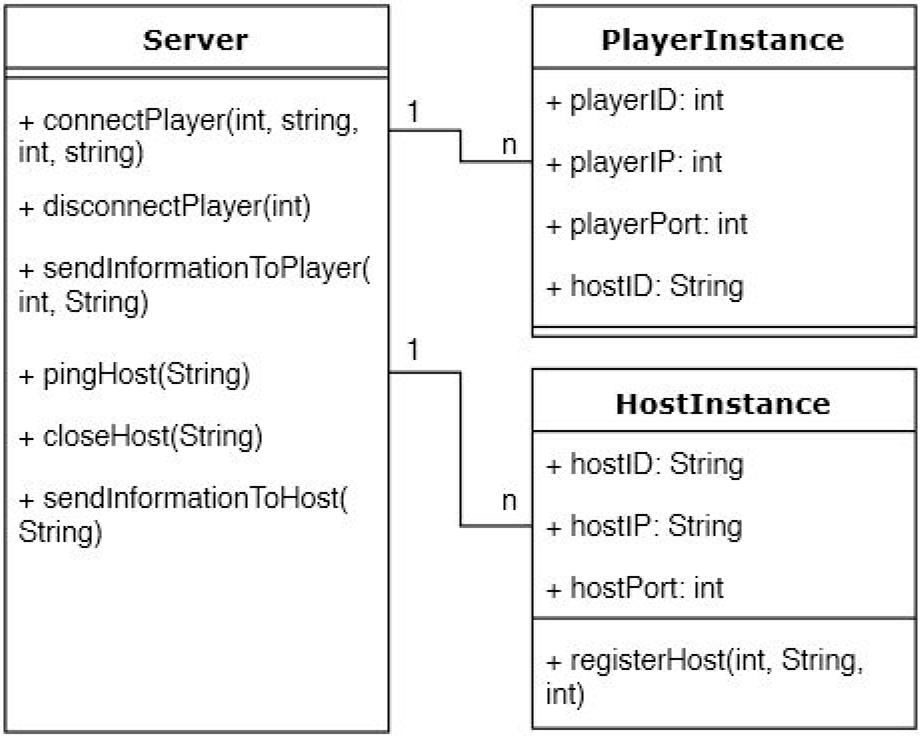
Non-Functional Requirements:

1. **As a developer**, I would like to maintain a server for which host computers can obtain a letter code for their game lobbies
2. **As a developer**, I would like to maintain a server to facilitate communication between the host computer and player devices in each lobby
3. **As a developer**, I would like to maintain a server which will connect player phones to the correct host computer lobby for each respective letter code
4. **As a developer**, I would like to remove inactive/unresponsive lobbies from the table of hosts on the server
5. **As a developer**, I would like to handle mid-game disconnects such that players can resume their games
6. **As a developer**, I would like to view crash messages in case of errors when possible

Design Details

Descriptions of Classes

We can break down each of the main components (player, host, server) into the classes used to construct them. Due to the way scenes are constructed in Godot, the different pieces used to construct the Host and Player, called nodes, are organized in a hierarchical fashion. Ideally, the child nodes are built to function without accessing their parents directly, and information is passed to the parents through signal processing. Some nodes, marked [SINGLETON] are referenced essentially globally where needed.



Server - the component maintaining the hosted lobbies and directing net traffic from hosts to their respective players

PlayerInstance : object representing a connected player to a host, stored in a table

playerID - ID of the player given by their host

playerIP - IP address of the player device

playerPort - Port number of the player device

hostID - ABCD code representing the lobby that player is connected to

HostInstance : object representing a hosted lobby on the server (host has ID of 1 for a lobby), stored in a table

hostID - ABCD code given by server to identify that host *hostIP* - IP address of the host device

hostPort - Port number of the host device

registerHost(hostID, hostIP, hostPort) - setup a host lobby to receive connections from players

connectPlayer(playerID, IP, Port, hostID) - connect a player to a host

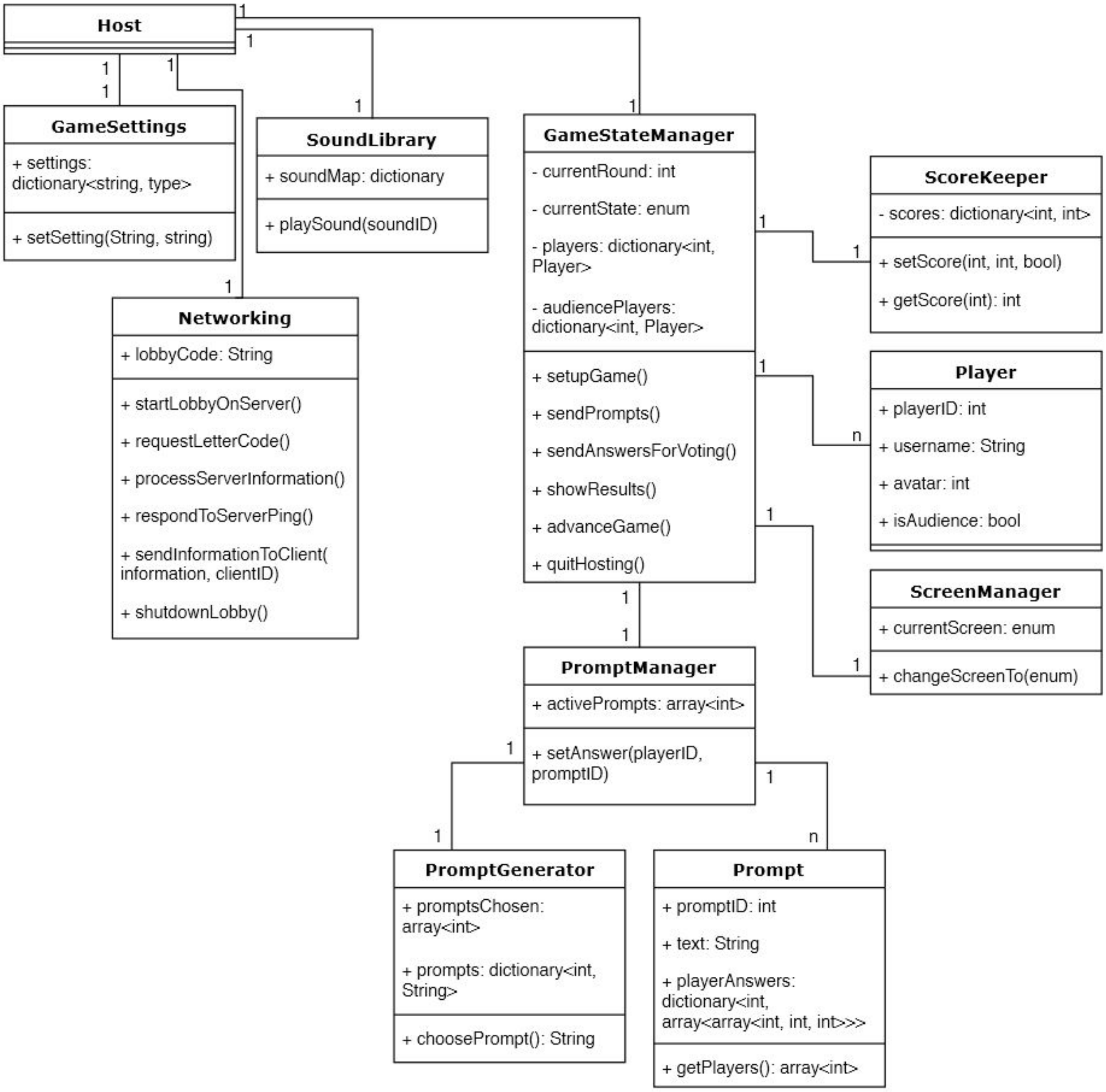
disconnectPlayer(playerID) - disconnect a player from a host

sendInformationToPlayer(playerID, hostID) - pass information from a host to a player

pingHost(hostID) - send a host a ping request to ensure their lobby is still active

closeHost(hostID) - close a host lobby and prevent further connection from it

sendInformationToHost(hostID) - pass information from a player to a host



Host - the component handling the overall game logic

- **GameSettings [SINGLETON]** : keeps track of customizable game settings such as the volume, maximum player count, or if the audience is enabled
  - *settings* - dictionary mapping setting names to values
  - *setSetting(name, value)* - sets the value of a given setting to a given value
- **Networking [SINGLETON]** : handles incoming and outgoing network messages to and from the server
  - *lobbyCode* - 4 letter code generated by the server to identify the hosted game session
  - *startLobbyOnServer()* - attempts to register a hosted lobby on the server for players to join
  - *requestLetterCode()* - gets a letter code from the server which players will use to join the game
  - *processServerInformation(information)* - decodes a message from the server and calls the necessary functions to process it
  - *respondToServerPing()* - inform the server that the game lobby still exists to prevent its deletion on the server
  - *sendInformationToClient(information, clientID)* - sends data to the server to be forwarded to a particular client player such as a prompt or time expired notification
  - *shutdownLobby()* - tell the server to disconnect all players and end the game lobby session
- **SoundLibrary [SINGLETON]** : manages the sound effects
  - *soundMap* - dictionary which maps soundIDs to sound files
  - *playSound(soundID)* - plays a given sound effect on demand
- **GameStateManager** : controls flow of the game, processes game information, directs screens when to show
  - **Player** : object representing a given player
    - *playerID* - unique identification number for a given player
    - *username* - chosen username for a given player
    - *avatar* - chosen icon for a given player
    - *isAudience* - boolean keeping track of a user being an actual player or just a member of the audience
  - **ScoreKeeper** : keeps tallies of each player's score
    - *scores* - dictionary of playerIDs mapped to score values
    - *setScore(playerID, value, relative)* - sets a score of a player to a new value or relative to the old value by the value parameter
    - *getScore(playerID)* - returns the score of a given player
  - **PromptManager** : controls which prompts will be sent to which players, which answers belong to which players
    - **PromptGenerator** : chooses which prompts to use from the prompt files
      - *promptsChosen* - array of promptIDs already used this game, to prevent duplicates
      - *prompts* - dictionary mapping prompt IDs to prompt strings
      - *choosePrompt()* - obtain a new prompt string
    - **Prompt** : object representing a chosen prompt
      - *promptID* - the ID of the prompt
      - *text* - the actual words of the prompt
      - *playerAnswers* - dictionary of playerIDs mapped to their answer of a prompt

getPlayers() - returns an array of playerIDs who are to answer this prompt

activePrompts - array of prompt objects currently in use

setAnswer(playerID, promptID) - enters an answer for a given player into the respective prompt object

ScreenManager : changes to the appropriate screens when necessary

TitleScreen - UI for the screen seen before the game starts

InstructionsScreen - UI for the screens providing directions at the start of a game

AnswerPromptsScreen - UI for the screen shown when players are given time to answer

VoteOnAnswersScreen - UI for the screen shown when players are given time to vote

ResultsToVotingScreen - UI for the screen shown after voting rounds end and the voting tallies are displayed to the players

CumulativeResultsScreen - UI for the screen shown between rounds, summarizing the total points earned by the players

CreditsScreen - UI for the screen shown after the game ends which displays the names of the people involved in creating the game

currentScreen - maintains a record of what the current screen shown is

changeScreenTo(screen) - changes the screen on demand to the requested screen

currentRound - variable to keep track of what round of prompts the players are on

currentState - variable to keep track of what part of the game the players are in

players - dictionary of player IDs to player objects representing the connected players

audiencePlayers - array of player IDs of players currently in the audience and not answering any of the prompts

setupGame() - initialize all data necessary for the game to begin

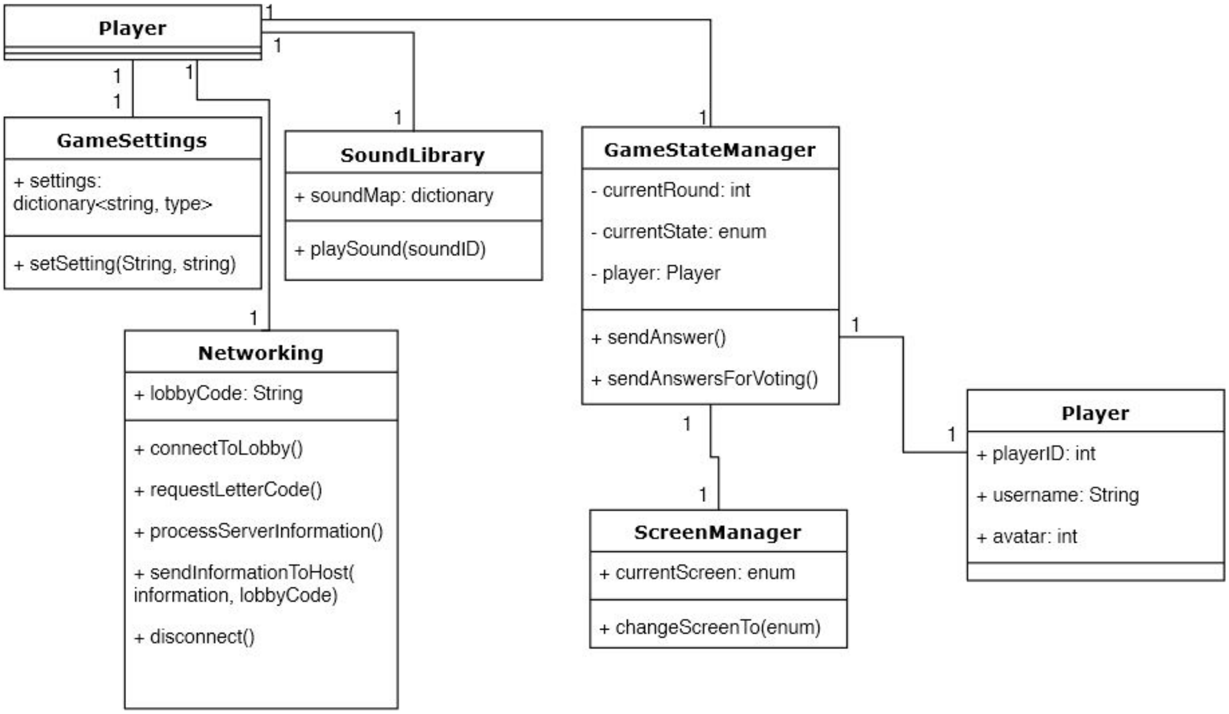
sendPrompts() - obtain new prompts and send to all players

sendAnswersForVoting() - send the answers from a given prompt out for voting

showResults() - display the results on the host screen

advanceGame() - move game along to the next phase of play

quitHosting() - ends the lobby session



Player- the component representing each connected client trying to play the game

GameSettings [SINGLETON] : keeps track of customizable client settings such as the volume or font size

settings - dictionary mapping setting names to values

setSetting(name, value) - sets the value of a given setting to a given value

Networking [SINGLETON] : handles incoming and outgoing network messages to and from the server

lobbyCode - 4 letter code generated by the server to identify the hosted game session

connectToLobby() - attempts to connect to a hosted lobby on the server

requestLetterCode() - gets a letter code from the server which players will use to join the game

processServerInformation(information) - decodes a message from the server and calls the necessary functions to process it

sendInformationToHost(information, lobbyCode) - sends data to the server to be forwarded to a particular hosted game session such as a vote or an answer to a prompt

disconnect() - tell the server to disconnect the player from a given lobby

SoundLibrary [SINGLETON] : manages the sound effects

soundMap - dictionary which maps soundIDs to sound files

playSound(soundID) - plays a given sound effect on demand

GameStateManager : controls flow of the game, processes game information, directs screens when to show

Player : object representing a given player

playerID - unique identification number for a given player

username - chosen username for a given player *avatar* - chosen icon for a given player

ScreenManager : changes to the appropriate screens when necessary

JoinGameScreen - UI for the screen seen before connecting to a host

EnterNameAndAvatarScreen- UI for the screen where players enter a name and avatar before joining the server

WaitingForStartScreen - UI for the screen shown when the user is waiting to being playing

WaitingScreen - UI for the screen shown when the user does not need to give input

AnswerPromptsScreen - UI for the screen shown when players are given time to answer

EmojiCanvas - the dropping point for users' emojis when answering promts

EmojiPalette - the container holding emoji inputs for the player

VoteOnAnswersScreen - UI for the screen shown when players are given time to vote

VoteOnFinalRoundScreen - unique UI for voting for the final round

ResultsToVotingScreen - UI for the screen shown after voting rounds end and the voting tallies are displayed to the players

CumulativeResultsScreen - UI for the screen shown between rounds, summarizing the total points earned by the players

CreditsScreen - UI for the screen shown after the game ends which displays the names of the people involved in creating the game

currentScreen - maintains a record of what the current screen shown is

changeScreenTo(screen) - changes the screen on demand to the requested screen

currentRound - variable to keep track of what round of prompts the players are on

currentState - variable to keep track of what part of the game the players are in

player - player object representing the connected player

sendAnswers() - send the answers to a given prompt back to the server

sendAnswersForVoting() - send the answers from a given prompt out for voting

Interactions Among the Classes:
For our Server, there is one main Server object controlling all of the Player and Hosts connections. As it receives networking messages from Players and Hosts, it creates and deletes records of these connections in Player and Host tables. The rows of the tables do not perform any real operations.

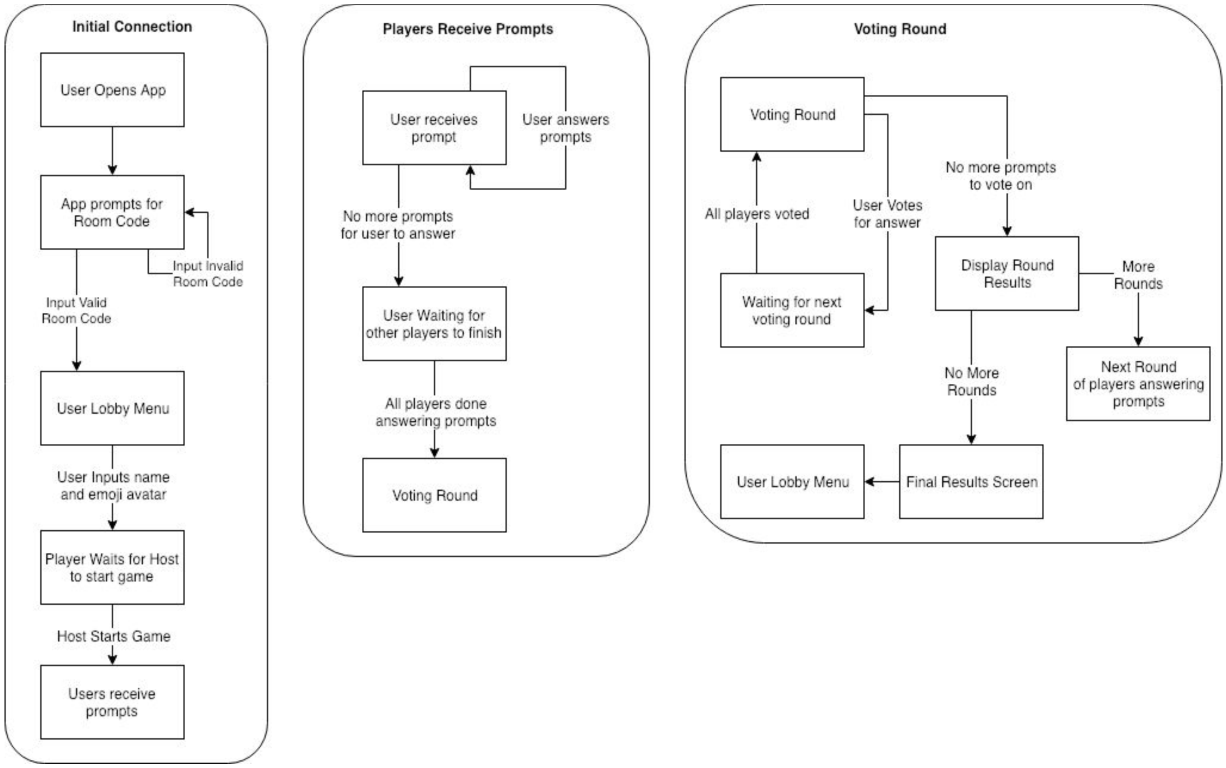
The Host and Player components behave fairly similarly to each other. The parent nodes named Host and Player own all of their respective children and delegate most of the operations down to them. Each possesses singleton classes of GameSettings, SoundLibrary, and Networking which are specialized to specific functionalities. GameSettings stores all of the settings which may impact many parts of the game. SoundLibrary maintains all of the sound effects which may be played. Networking contains an API for communicating externally with the server. Mainly, these classes will interact with the GameStateManager class which controls most of the flow of the game.

GameStateManager will direct the ScreenManager class to change screens when needed, and it subscribes to the signals generated by the UI elements such as submit buttons in order to react to human interactions with the application. In this configuration, the lowest level UI nodes and other helper nodes managed by GameStateManager do not need to do a lot of active processing. Instead, GameStateManager will only react to what it needs to and minimize the coupling between the classes.

GameStateManager also has a few other classes as children such as the ScoreKeeper class which stores player scores and the PromptManager class which generates and stores the results from different user answers. Player objects are also stored in a dictionary for the Host and in a variable for the Player which are used to maintain information about the different players such as their usernames and their emoji avatar choices.

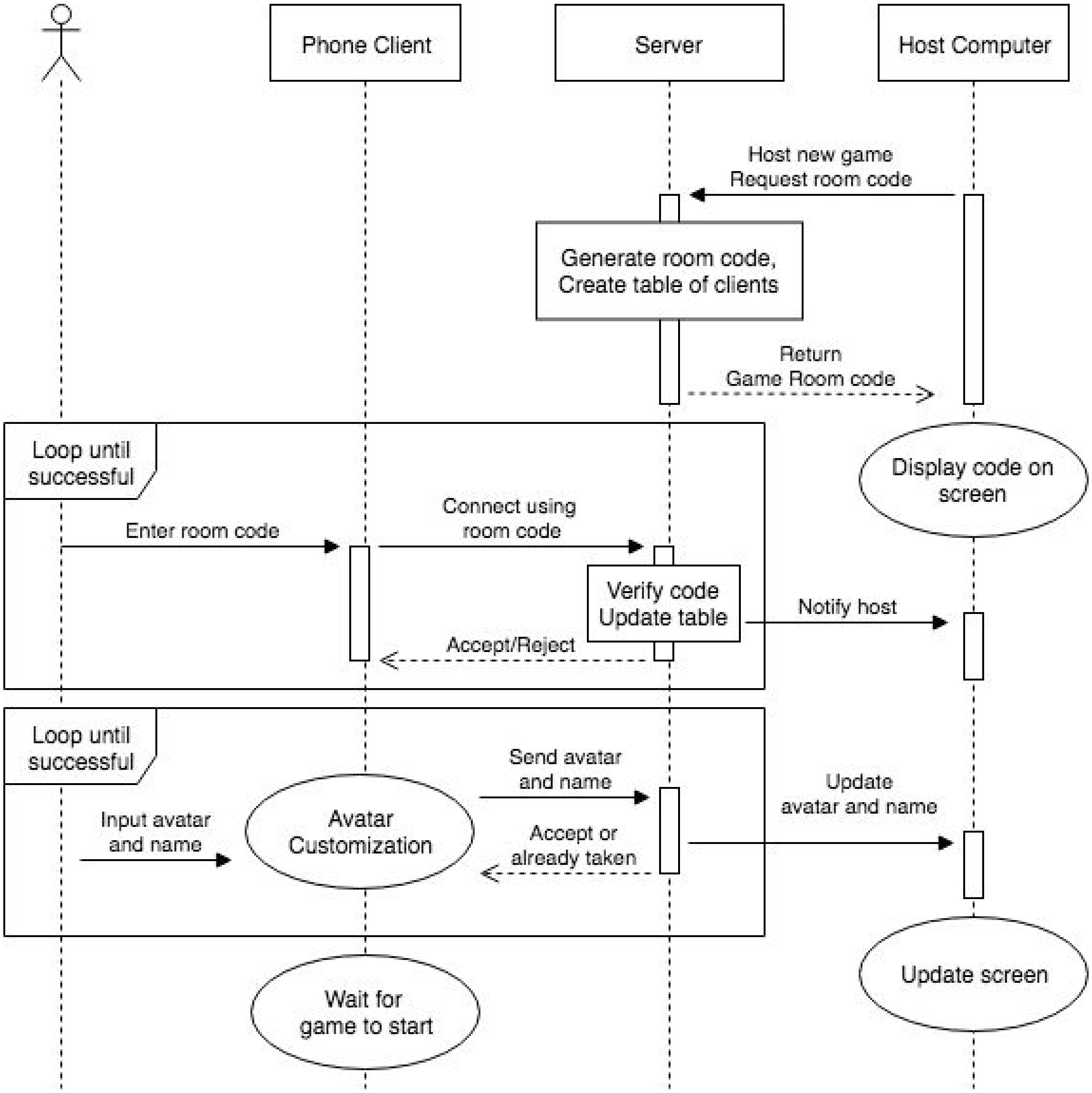
State Diagram:

Game Logic:

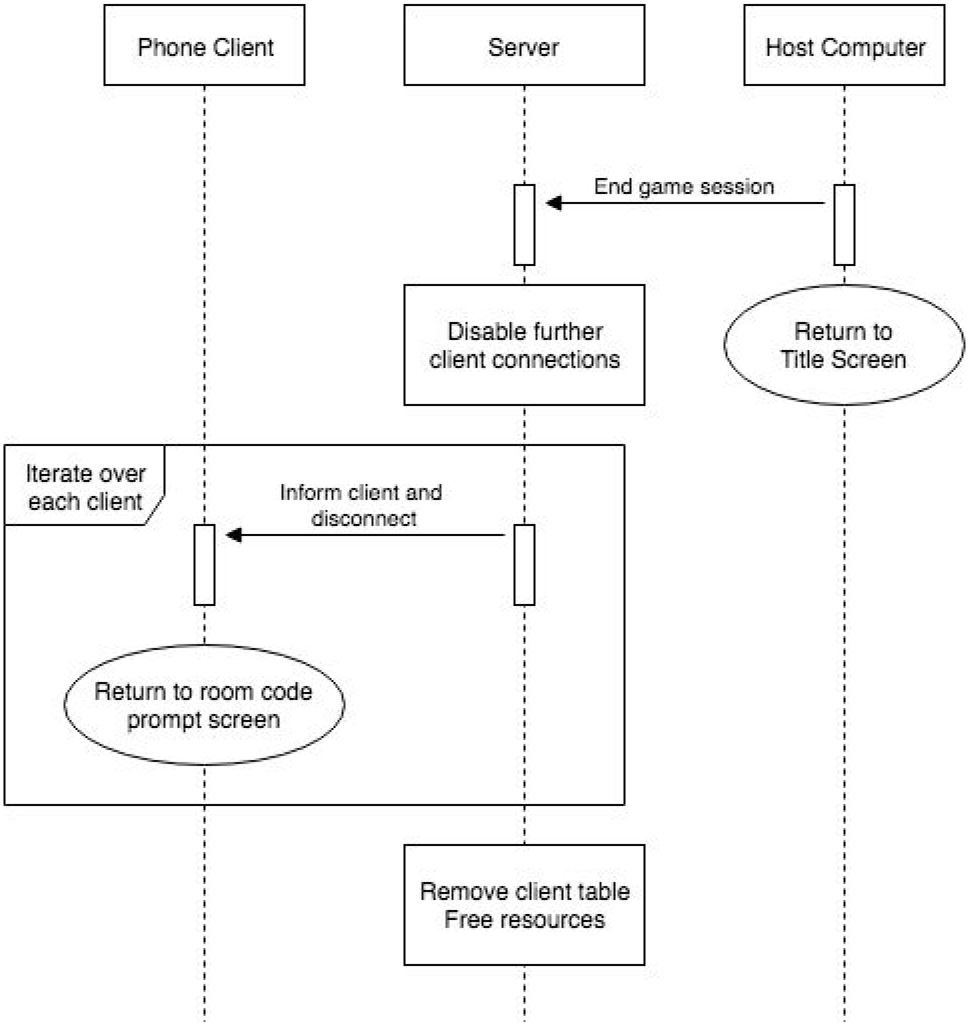


Sequence Diagrams:

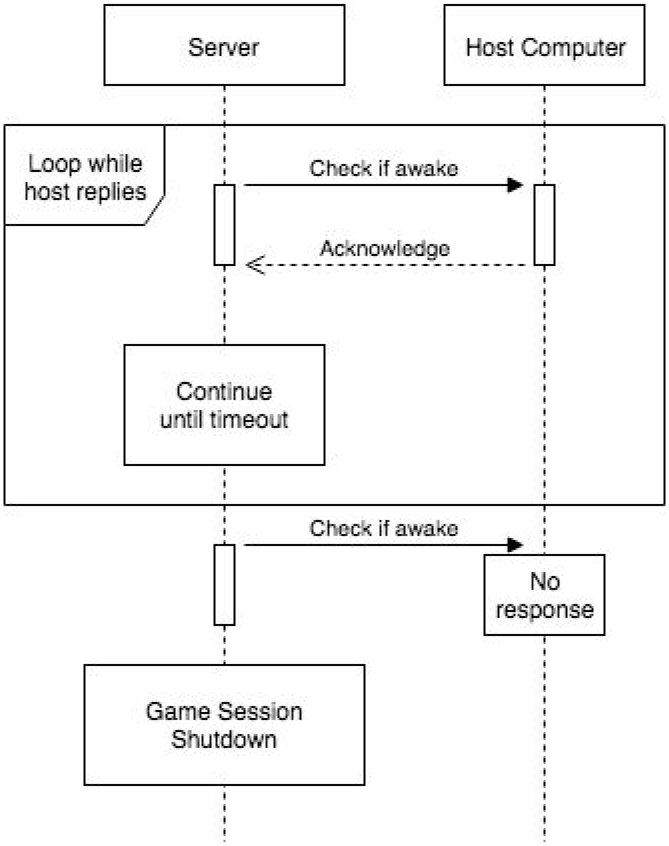
Lobby Initialization:



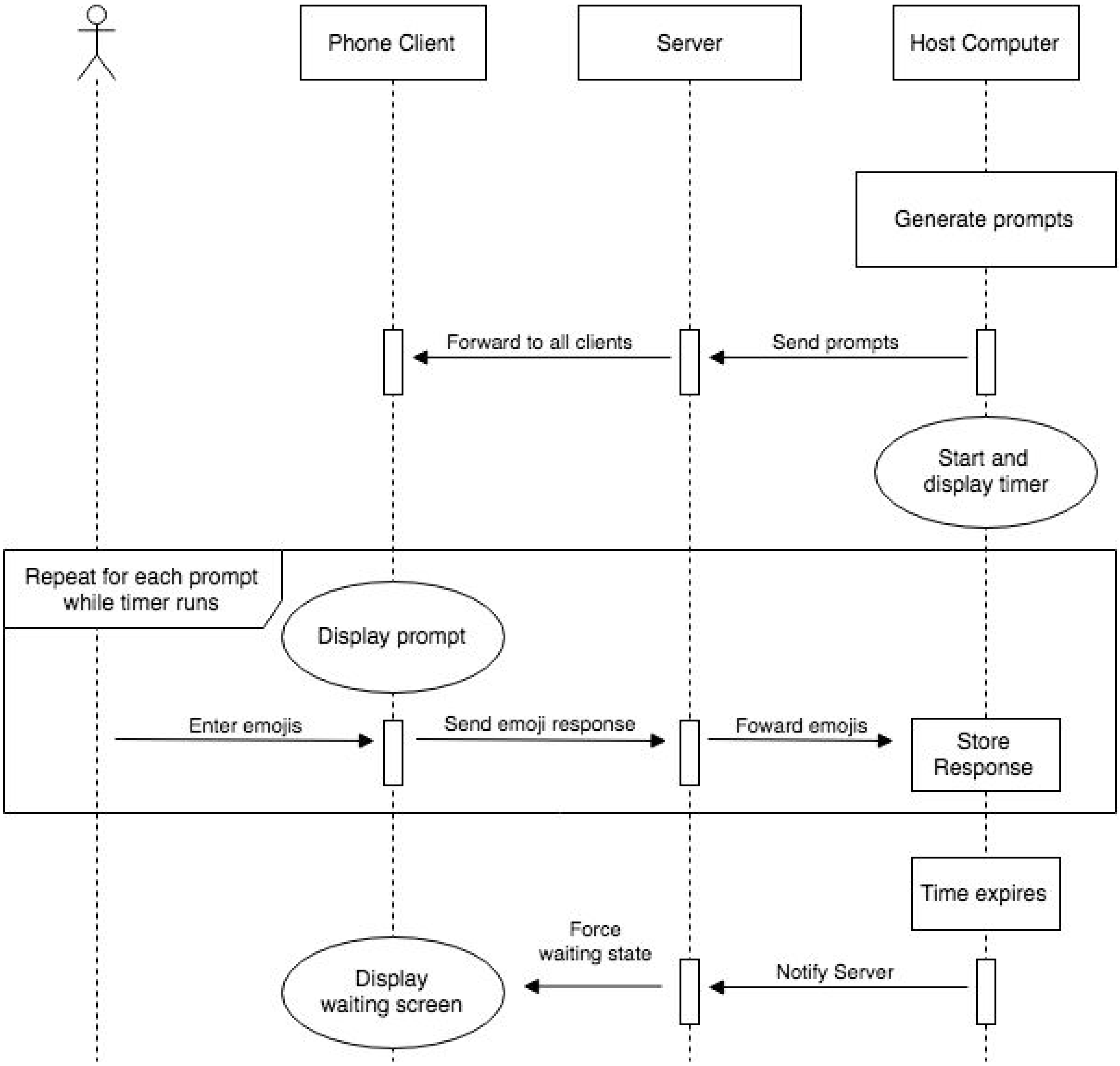
Game Session Shutdown:



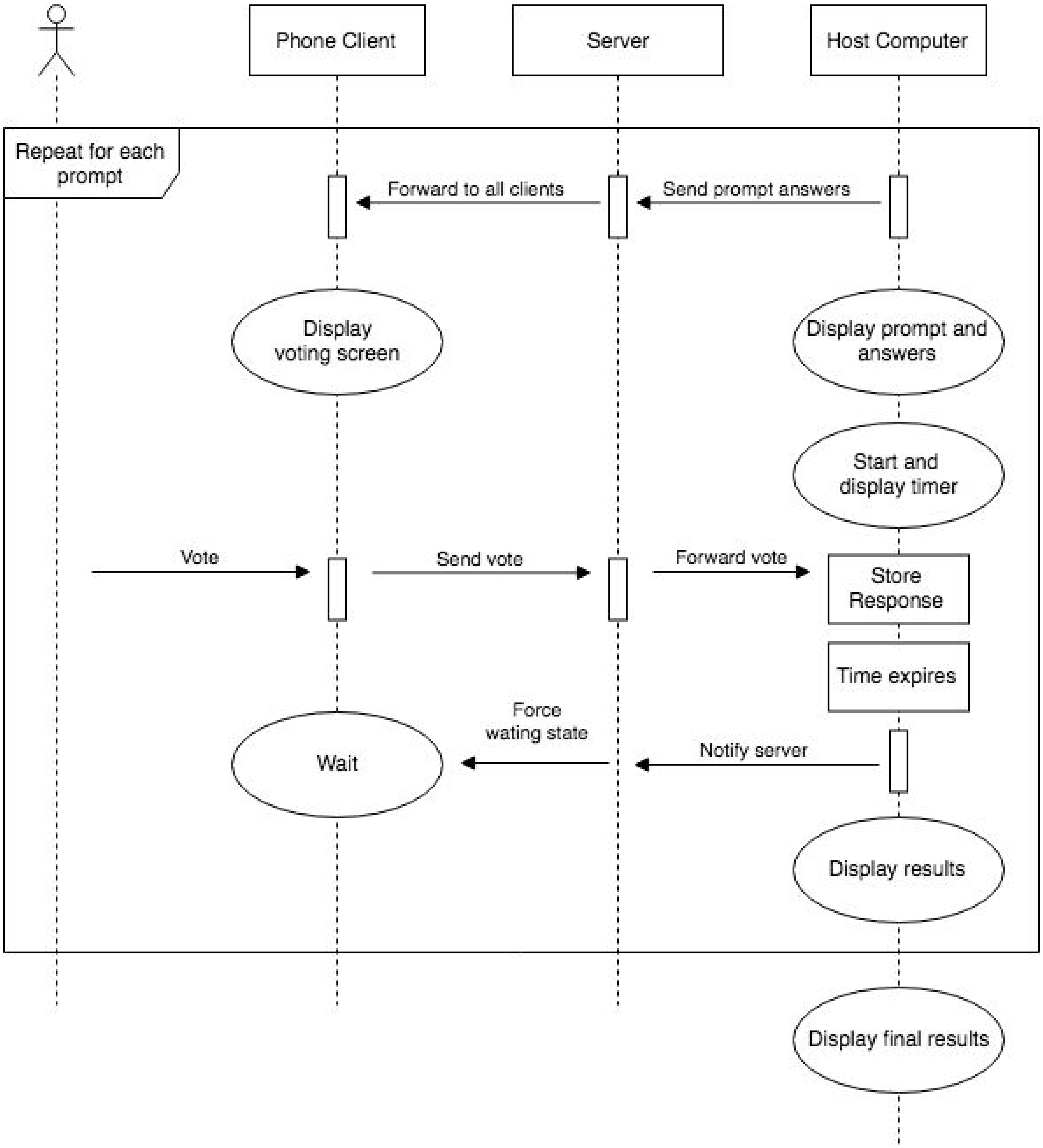
Host Timeout:



Answering Prompts:

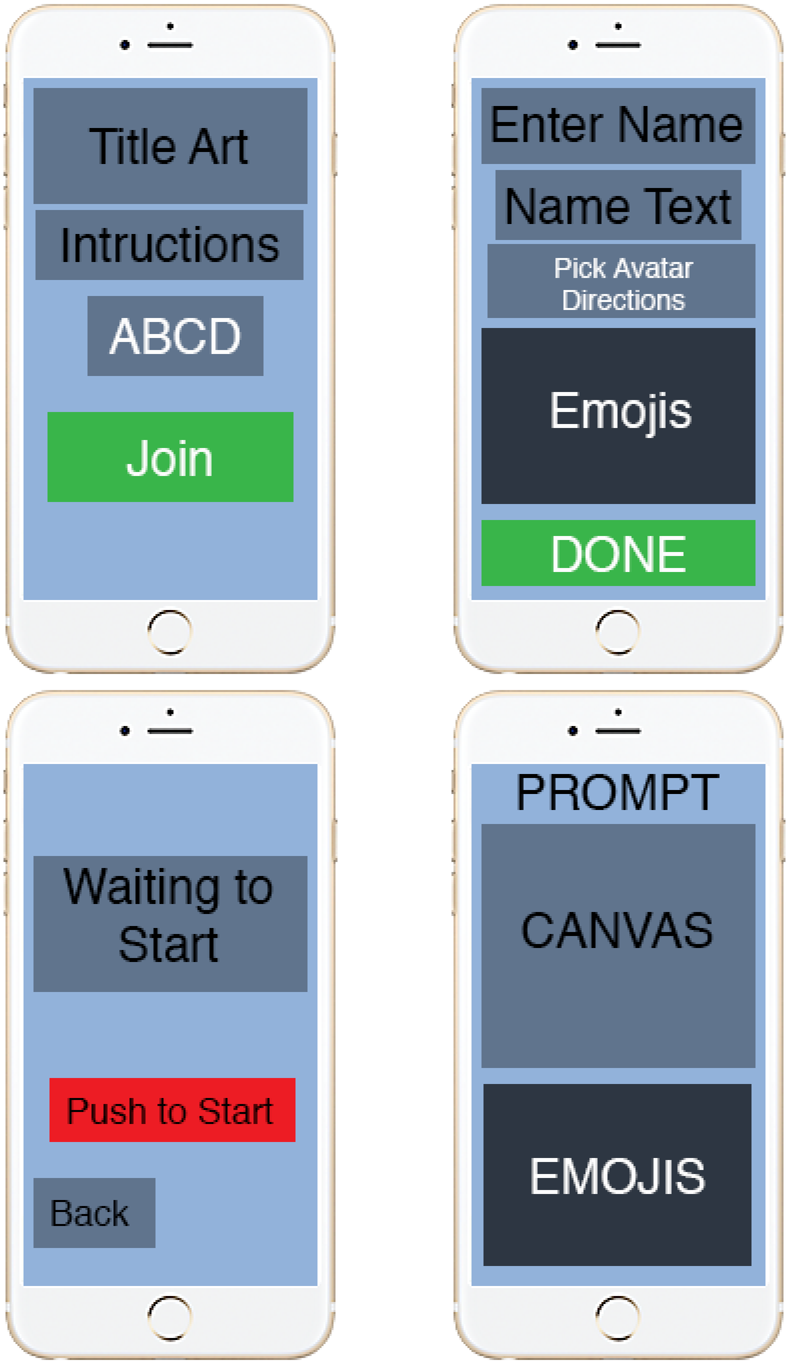


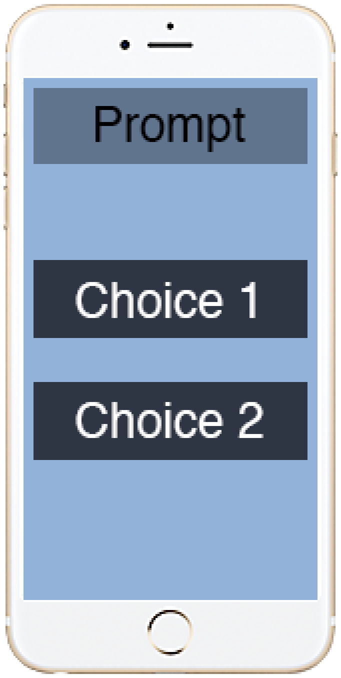
Voting on Answers:

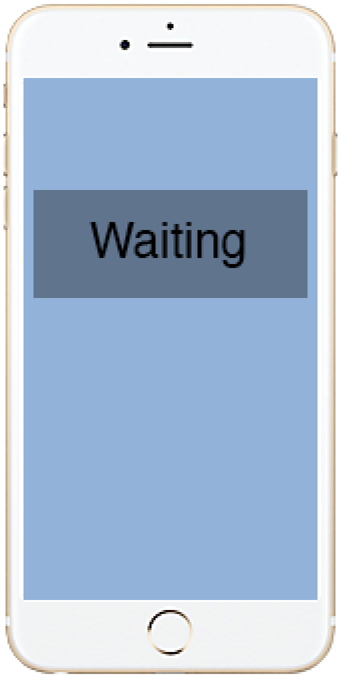


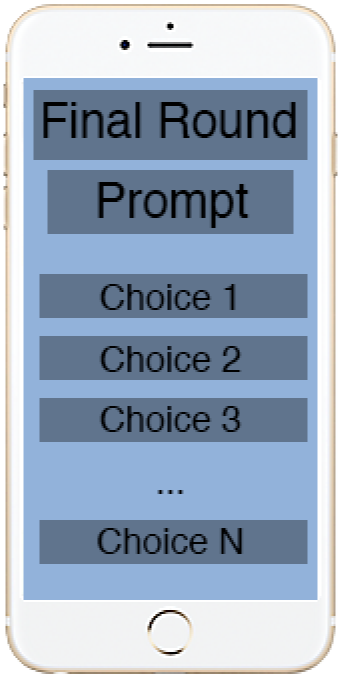
UI Mockups:

Phone:

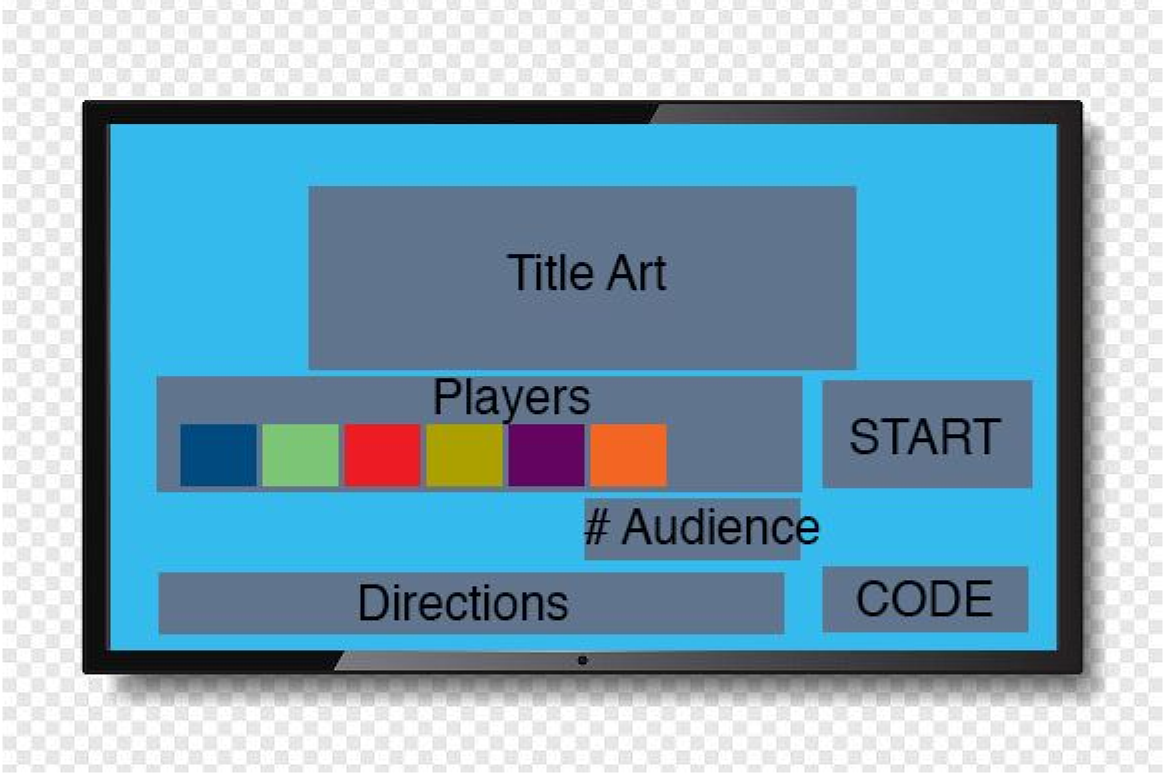


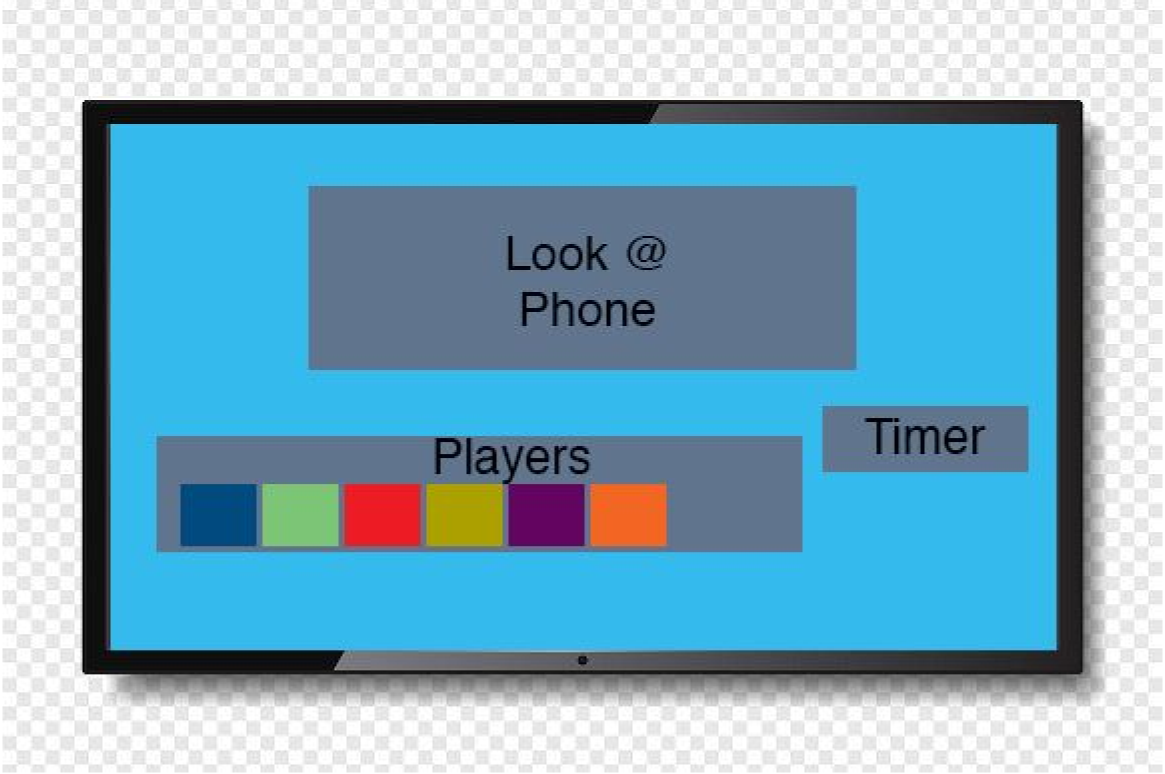

Prompt
Choice 1
Choice 2

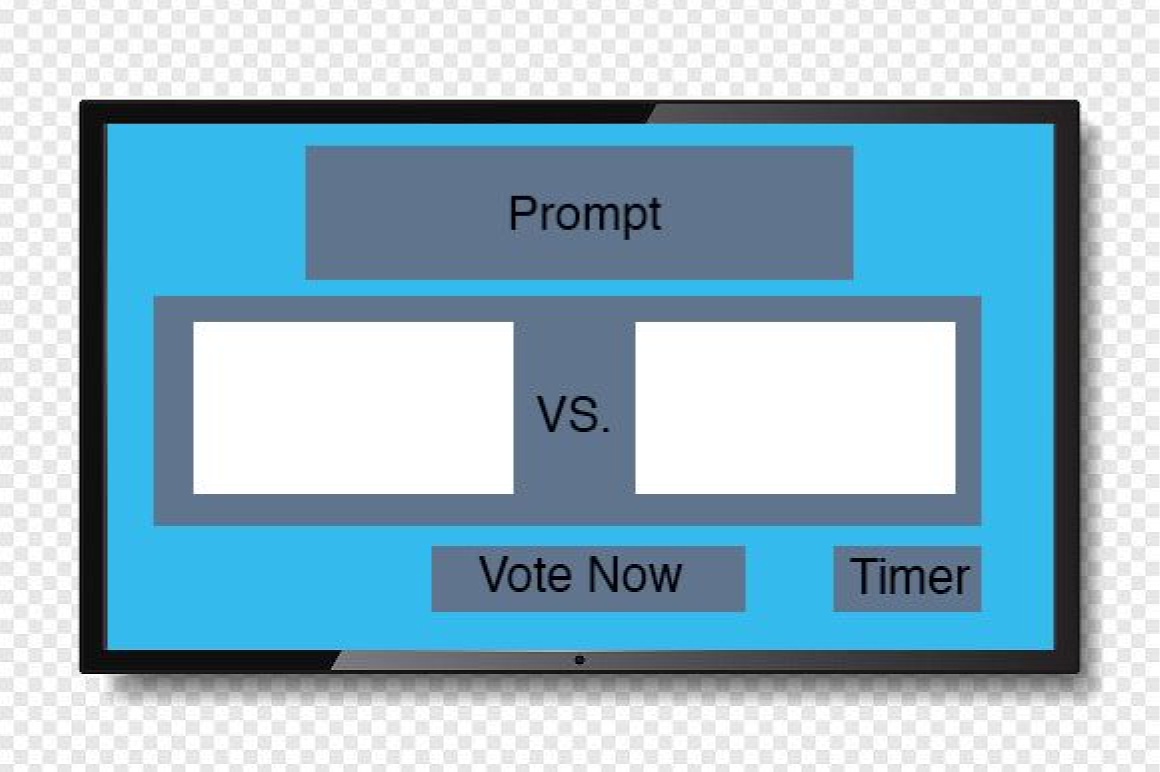

Waiting

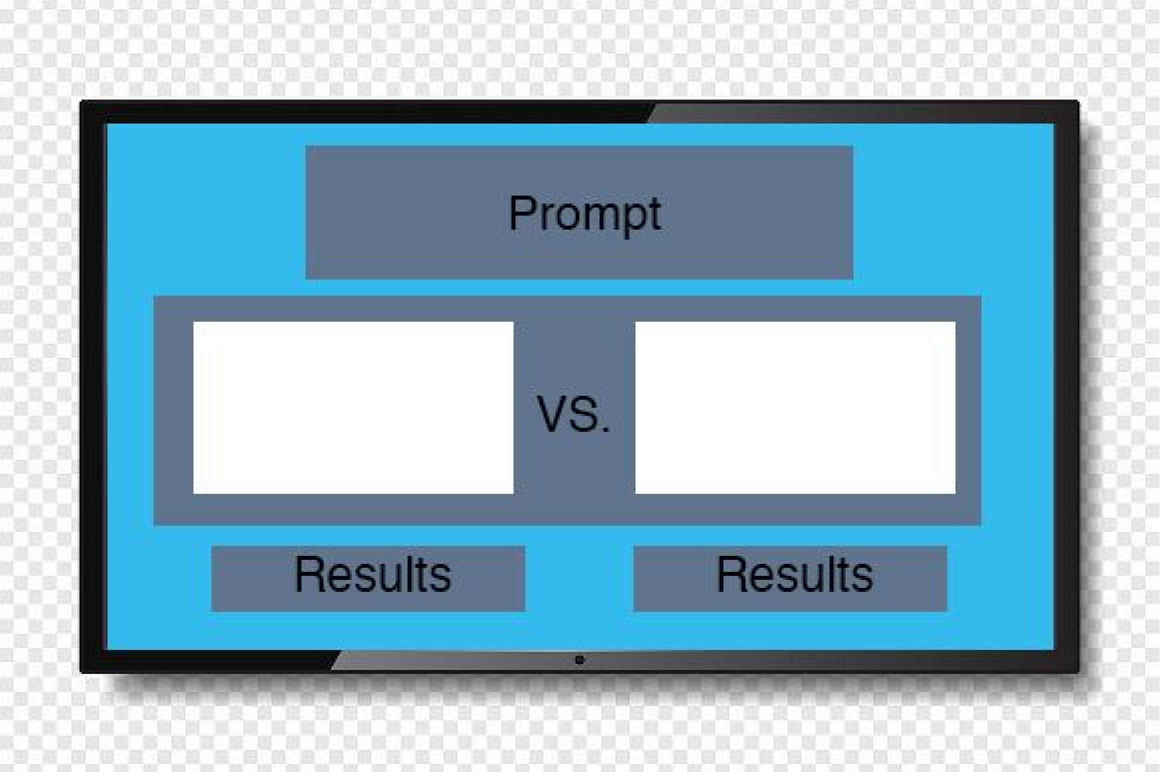

Final Round
Prompt
Choice 1
Choice 2
Choice 3
...
Choice N

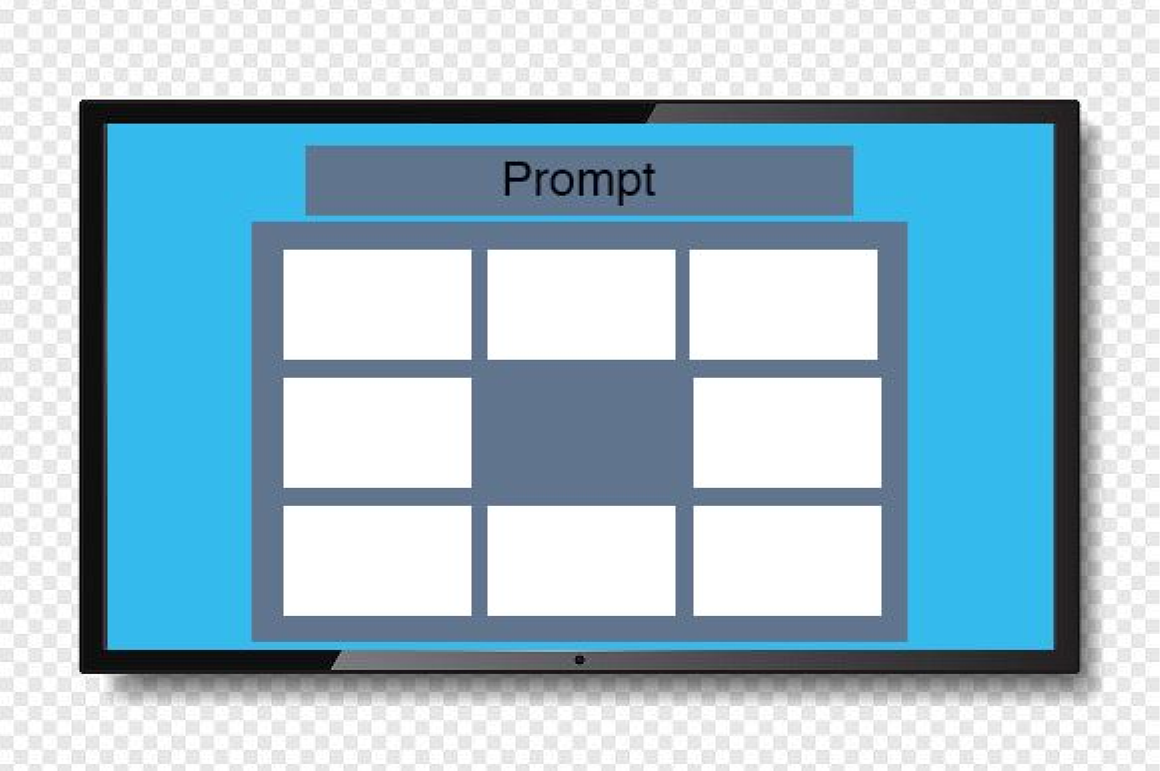
Host:


Title Art
Players
START
# Audience
Directions
CODE

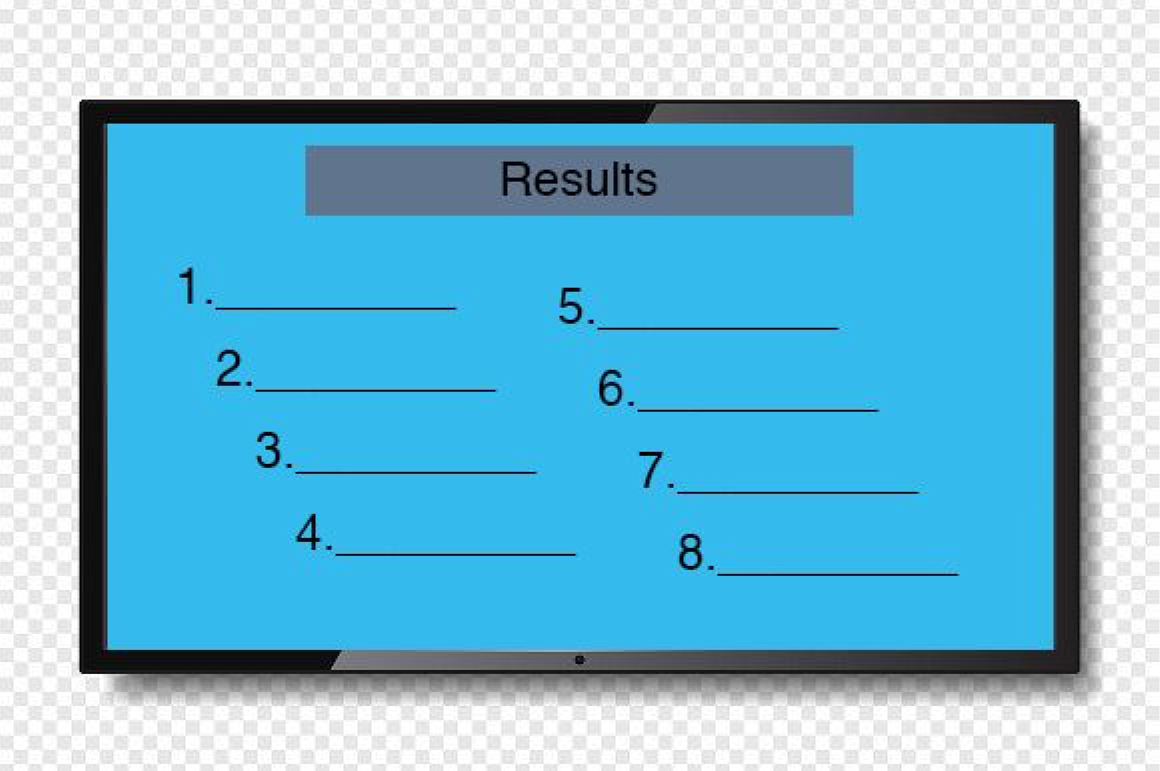

Look @
Phone
Timer
Players


Prompt
VS.
Vote Now
Timer


Prompt
VS.
Results
Results



Prompt



Results

1.____________

2.____________

3.____________

4.____________

5.____________

6.____________

7.____________

8.____________