Tumble Rumble Code Documentation

Revised Functional Specification

Our game has – being developed in an agile environment – undergone several drastic changes. A link to our original Functional Specification can be found here: https://docs.google.com/document/d/1S14ztRxnAhwNIQ1p6idqvPS1UneUU8B11ZKj5RVOgxM/edit?usp=sharing

Below, I will outline the revisions made to the original document.

High Level Design:

- Major Components and Modules:
    - **Game States**: Our game now has significantly fewer game states. We originally had 10 (Boot, Preloader, Main-Menu, Game Settings, Lobby-Create, Lobby-Join, Character Selection, Arena Selection, Arena, and Results), but I realized that a lot of the functionality of the separate states can be more efficiently accomplished in 1 game state. Thus, we now have only 6 game states: boot, preloader, menu, lobby, stage, and results.
        - Our boot state is called from the index.html file. From here, we take care of general things, and load the preloader.
        - The preloader will load all assets for all future game states, and will show a loading bar at the bottom of the screen so players know how much has been loaded.
        - Then the menu state starts, and players can enter the lobby state at the click of a button.
        - In the lobby state, the game waits for another player to join, and then automatically calls the stage state to start the game. Alternatively, players can click a button and play solo (good for practice). If anyone was to join while someone else was training, they would be thrown into the same game with them.
        - In the stage state, players interact with each other. After a winner is determined, the results state is called.
        - The results state simply tells the user if they won or lost. After a few seconds, the game is restarted.
    - **Handlers**: We have added a new major component, handlers. We have several handler objects. These are simply abstract classes called upon by various game states in order to cleanly handle socket events, stage development, and maps. (Note: Maps are not currently being used in our most recent revision).
    - **Assets:**

        - **Entities**: We have many entity modules.
            - LocalPlayer: This class creates the object for the local player, handling the user input and sending out socket events.
            - RemotePlayer: A class that creates a remote player. Essentially the same as above, but this class doesn't look for user input and instead waits for socket events to move the sprite.
            - LocalWeapon: Here we create the local weapon. This will create a weapon that can be controlled by the local player.
            - RemoteWeapon: Same as above, but this one is controlled by remote players via socket events.
            - Healthbar: A child sprite of each player that visually represents how much health they have left.
            - Tumbler: This is the base player sprite. This is called whenever a localplayer or remoteplayer is created. All the sprite physics and attributes are given here.
        - **Sounds**: Our sound directory stores all sound and music used by the game. Currently this is only 1 file, the background music for the menu.
        - **Textures**: Our texture directory stores all textures for GUI (which includes buttons, backgrounds, and healthbars), players, weapons, and world (things like cactus images and foregrounds).
        - **Tilemaps**: Currently not being used, but these consist of the various tilemaps, stored as .tmx and .json files. Each tilemap has its own folder, and within that folder, we also store all of its textures.
- Interface Elements and Media Assets:
    - Main Menu:
        - Buttons:
            - We currently only have 1 button, which starts the lobby state.
        - Sound:
            - Currently we have a placeholder track, but we will soon implement a western-style soundtrack taken from a public-domain website and put on loop, fading in and out at the beginning and end. This soundtrack will continue for the next several game states, up until the users reach the character selection.
    - Lobby
        - Text:
            - Waiting for players...
        - Buttons:
            - Start: The player can start solo if they wish
    - Stage
        - This is the primary section of the game. This will display the world, which is similar to flappy bird in that it has a perpetual cycle of randomly generated cactus walls with small openings that the user must fly through, all the while they are fighting and trying to kill each other.

        - Health bars for each player (these are visual bars with numbers inside them representing their health)
        - We still need to implement:
            - A timer at the top of the page
    - Results
        - After the fight ends, this game state is loaded and the winner for the round is shown.

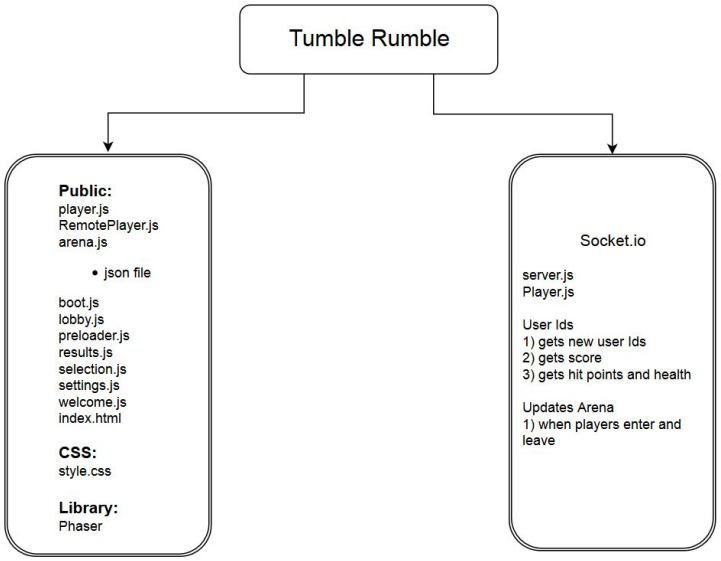
External Libraries and Frameworks: We are using Phaser as our gaming framework. We chose Phaser because it is up-to-date, works well, and one of our team members (Cory) has experience using it. Phaser will supply support for sounds, collisions, and physics.

Client-Server Interaction: We are using socket.io for client-server interaction. Our Server will be constantly listening for events from all sockets, such as "move player" or "damage player" and "remove player." These events are triggered when a client announces them. Once the server is aware of the event, it sends the new data to all the clients, which are also listening for events such as these. This allows each client to move their local player, and have that movement reflected across all other clients. In addition to just moving the players, we can also activate their weapons, allow them to be damaged and die, and respawn.

Database Management: We aren't storing player profiles, usernames, or anything related to that.

Hosting: A live demo of our program (hosted by Heroku) can be found here: https://tumblerumble.herokuapp.com/
Alternatively, we can host our program locally or on the TAMU servers.

Diagrams and FlowCharts



Code Explanation

At the forefront of our program, we have 2 directories: client and server. When the program is started, the files in the server directory (Server.js and Player.js) are run. Server.js allows clients to connect, and Player.js allows the server to store necessary information about each player. Things are much heavier on the client side of the program.

Within the client directory, we have our index.html file; a directory for the game's assets, states, and handlers; a directory for the website style, and a directory for the vendor (Phaser). When a client connects, they are shown the index.html file. This file will use the .css files in the style directory to style its content. It will also create a canvas and load the game's first game state, boot.

The relationships between the game states was already explained above under the Functional Specification Revision tab, so I won't discuss them here. What I will focus on, however, is everything that goes on during the actual gameplay.

So when the Stage state starts, a few things happen. Since we are using Phaser, the first thing that happens is whatever we have inside our create() function. This is where we spawn all local and remote players, as well as build the world. I'll go into more details later. Next, the game will continually call the update() function until the game ends. In our game, that is defined as whenever there is just 1 player left, and someone has died. The one player remaining is declared winner, while all other clients are declared loser. The only other function we have in this game state is destructor(), which simply clears memory. This is necessary for our game to restart, otherwise some of the values will be skewed.

Going back to the initial create() function, the first thing that happens is we assign the current game instance to a global variable. This allows separate classes to manipulate data belonging to the local player's game instance. Without this, our game is buggy and unreliable.

The next thing create() does is straightforward: it starts the music.

Then it builds the world. Being only 2 lines of code, this may appear simple, but a lot of things are happening when the StageHandler instance is created. In StageHandler.js, we build the world. This is done by adding the background as a tilesprite and moving it left constantly to give the feel that the player is moving right. We also have a perpetual line of cactus walls appearing and sliding through the stage, forcing the player to jump through holes or die. The holes themselves are randomly generated, so in order to have all clients show the same holes, we had the server send a random number to all the clients. Whenever a client receives this number, it creates a wall with the hole at the specified number.

After the world is built, we spawn the player. The player class contains attributes such as location, health, and whether or not the player is alive, among other things. It also holds a sprite,

which is stored inside an instance of the Tumbler class. The Tumbler class is where the physical nature of the sprite is stored. We check for collisions and handle things like gravity here.

After the player is spawned, we move to the update() function, where the local player, world, and remote players are updated. This is done by calling the update method of each object, and constantly checking for events sent out by the server.

We also check for a winner with each loop. If someone has died, and there is no more than 1 player left, that player is declared winner. If only one person was playing, then there is no possible way for them to win. The game will continue to go on forever until they die. We will probably add a timer to provide a way for them to win, but that hasn't happened yet.

Functional Iteration Summary (what we need to do for the changelog)

Proposed Changes: There were quite a few changes proposed from our user study data. Many of these changes were related to the *current* gameplay and its mechanics, such as adding more weapons, a timer, lives, and some point-tracking system. While these were all good suggestions, we decided a complete overhaul of our game could serve us better. So, the primary proposed change and the only one we actually implemented - from the user study - is this: changing our game from a plain fighting-style game with a static stage to a dynamic flappy-bird style game with a moving stage. With this change, many of the other changes become irrelevant. However, we did eventually add a timer and scoring system, as these changes could serve our new game as well.

Report of Changes: It actually wasn't that hard to implement this game overhaul. Our game's software architecture provided an easy way of updating our game with the new stage and character attributes. We only had to create one class, StageHandler.java, which contains about a hundred lines of code that listens for a randomly generated number sent from the server at intervals of once every 0.7 seconds. This number is all the clients need to create the cactus walls, as we use this number to map where the gap in the walls should be. After each client receives this number, we instantly spawn the cactus wall and have it move left at a constant speed. Every client will show the same exact wall, allowing for synchronization. To make room for this change, however, we had to discard a rather large section of our game. This was the TileMapHandler.java class and its supporting directory, with 4 fully-functional and beautifully designed maps. We may have discarded these from the main code, but they still exist in the repository as ghost files, should we ever decide to reimplement them. Overall, we learned a lot from Deliverable 5's user study, and we believe we were adequately able to suit the user's requests with our overhaul.

With our new changes, we decided to upload our game to a public web server so our friends could play and give us more feedback. We used Heroku to accomplish this, and we cannot describe how much easier it is to get users to play. Before, we had TAMU students connecting to the TAMU network, which was a hassle since many of our friends weren't computer science majors, and needed help with setting up a VPN for off-campus gameplay. With Heroku, we were able to accomplish much more in a drastically smaller amount of time. And our

results from this separate user study were overwhelmingly positive. Our players actually enjoyed our game now that it posed a real challenge.