Tumble Rumble! Final Report

By ComputersAreHard

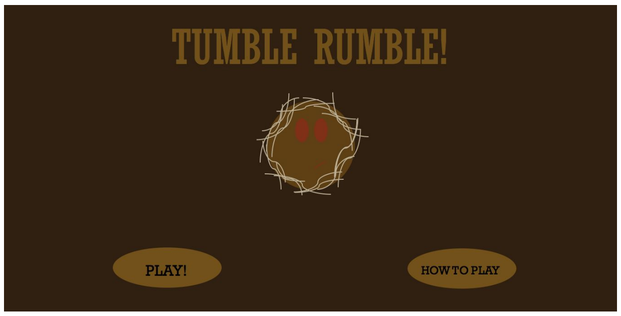
Meaningful Play Specification:

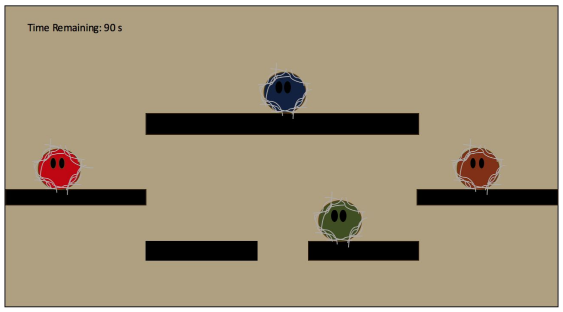
Our team name is ComputersAreHard, and our game is Tumble Rumble! The premise of our game is based on two fundamental concepts: simplicity and challenge. We wanted it to be simple, so the users clearly understand the rules and controls necessary to play this game successfully and easily. We also designed it to be challenging, with multiple fatal obstacles in a fast-paced environment, while avoiding other players.

The objective of our game is straightforward: players need to stay alive by avoiding cactus walls, other player's swords, and going out of the screen's bounds. The players must concentrate heavily in order to avoid getting injured or instant death from the previous mentioned obstacles. The instruction panel in the side margin of the website provide a clear and constant objective for the players to complete, but it is up to the skill and focus of the player to achieve victory in our game.

Throughout our numerous and extensive user studies, we have discovered a pattern: determination and focus are required in order to achieve victory. The results of these studies can be found below. Because our game is simple and challenging, our players find meaningful play with every round.

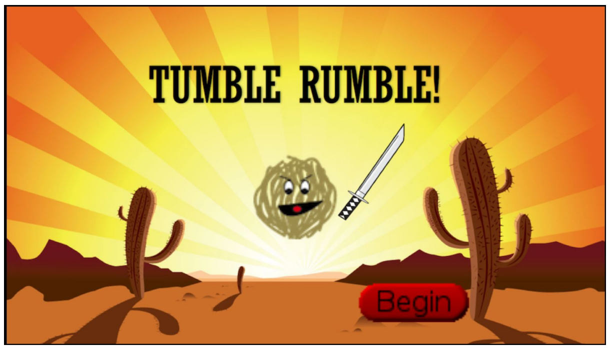
Old Mockups:

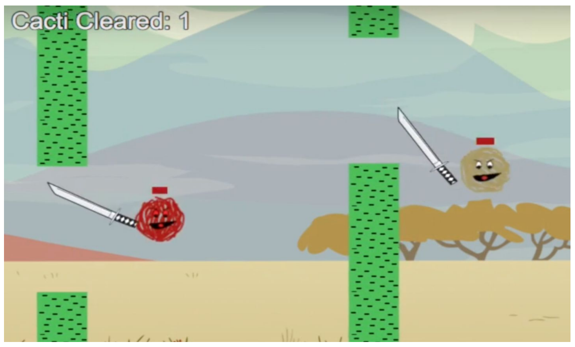




↓

New Mockups:





Mockup Discussion:

When we first began development, we had a clear idea of our game's states and progression. Throughout our deliverables, we kept the general flow of our game states, but iteratively updated the graphics several times. Our original and final product is shown above, with the most noticeable change being from the change from the Arena-style fighting game to the flappy-bird fighting game. With this change, our game became much more fast-paced and (in according to our user studies) more enjoyable as well.

Scenarios:

There were two primary scenarios we used throughout the development of our game. The first scenario consisted of 2 players, and the second consisted of 3 or more.

In the first scenario, we had one player win and one player lose. The objective of this scenario was to test the game's fighting mechanics. We had two players fight to the death using swords. In our earlier game's phases, this was done on a static stage without any obstacles. The results were disappointing, with our users complaining about lag and difficulty winning, to the point of the game being unenjoyable. After our game's conversion to the dynamic cactus stage, however, we repeated this scenario with the same previous users. The results drastically different with overwhelmingly positive reviews. Our players found the fast-paced environment much more enjoyable, now that the game was challenging and winnable.

Our second scenario also featured similar results and changes. The only difference this time is we had 3 players instead of 2, to test our game's capability of supporting multiple users. Our server - being relatively light compared to a heavy client - performed flawlessly. The data we gathered from this scenario supported the data we gathered from the other scenario.

These two scenarios make for a successful multiplayer game as they keep a competitive multiplayer experience, as well as, a survival-type one player experience.

High-Level Software Design:

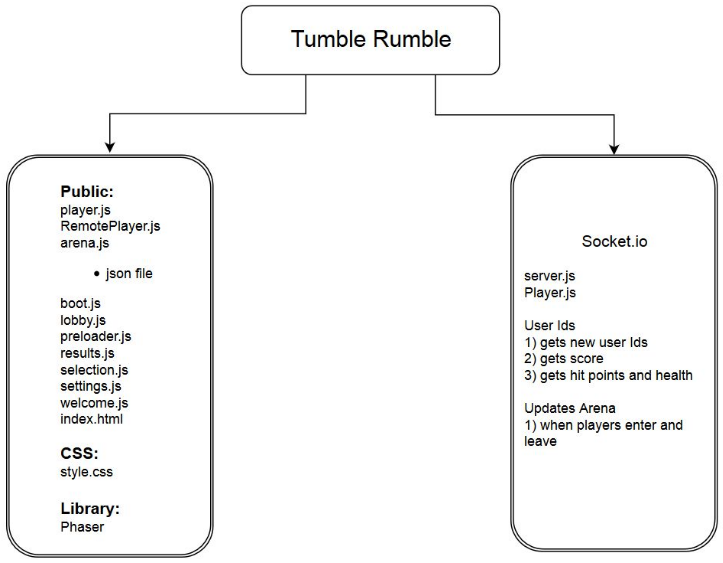
- Major Components and Modules:
    - **Game States**: Our game now has significantly fewer game states. We originally had 10 (Boot, Preloader, Main-Menu, Game Settings, Lobby-Create, Lobby-Join, Character Selection, Arena Selection, Arena, and Results), but I realized that a lot of the functionality 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.
            - cacti: 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:

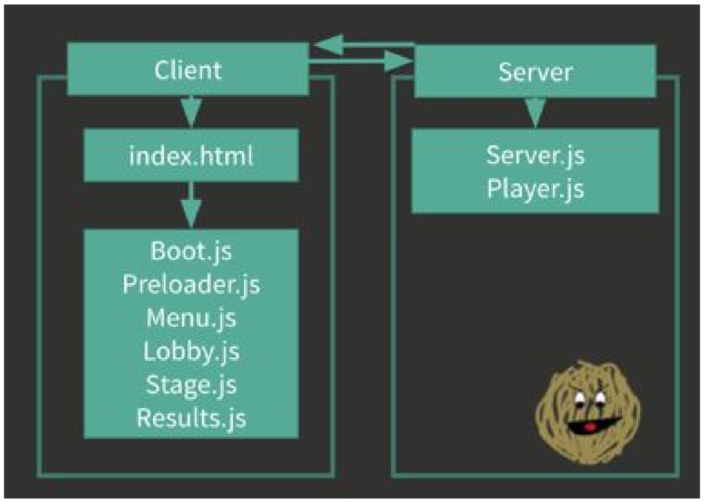
    - Buttons:
        - We currently only have 1 button, which starts the lobby state.
    - Sound:
        - Currently we have a placeholder track, but we will soon implement a western-style soundtrack taken from a public-domain website and put on loop, fading in and out at the beginning and end. This 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.
- Server/Client Communication Design
  - Relationship between Server and Client:
    - Both the server and client were written in JavaScript
    - We used Node.js and Socket.io for Server/Client Interaction, and we used Phaser as our game library.
    - A couple of primary socket events and their resulting actions:
        - New player – all clients create a new sprite
        - Move player – all clients move a certain sprite to a coordinate specified by the event's data
        - Damage player – all clients show a sprite has been damaged
  - Server:
    - Our server is very light
    - We have 2 server classes: Server.java and Player.java.
        - Server.java handles the socket events, while Player.java holds player's positions and various effects, such as damaged or dead.
  - Client:
    - Our client holds the bulk of the game code

- Once a client emits an event, the server will then broadcast that event to all other clients, which then update their sprites accordingly

Represented below is a visual of our original server/client architecture design:



And here is a visual representation of our latest server/client architecture design from our Presentation Slideshow:



Iterative Design Process:

- **Problems in our Design:**

  Throughout our iterative game development, we encountered many problems. Many of these were mechanical, in that the player was able to continually hold the up arrow key and fly upwards indefinitely. This resulted in many players accidentally killing themselves by flying off screen. Another mechanical problem was and the fact that the sword hits were initially one-hit kills, seriously decreasing the average game time.

Besides the mechanical problems, we faced many issues with our server not updating the player's position frequently enough, resulting in a laggy experience when playing with multiple people. We also had several bugs, such as players not dying when their health dropped below 0, or flying through cactus walls without taking damage, and various other minor issues that were solved relatively quickly.

The majority of our issues, however, came from communicating with our server and client. It wasn't until around the 6th deliverable did we realize the most efficient and practical method of communicating. Before then, we had problems with all our events being received and properly distributed between the other clients. Once we grasped a firm understanding of socket.io, we were able to fix all of these at once.

Perhaps the biggest problem we faced was making our game actually enjoyable. As explained previously, our game's first iteration lacked a dynamic stage and received mostly negative results from our user study. This major problem hoarded most of our time and effort.

- **Data used to identify and solve problems:**

  Most of the data that helped us identify and solve these problems came from personal tests, which we performed frequently and thoroughly as we added code. This helped us solve many of the mechanical and software design problems.

  As far as the problem of making the game enjoyable to the users, our user study was our primary source of data. Much of this data is cited above in the beginning of this document.
- **Changes to design to solve problems:**

  Our changes based on the received data is abundantly evident. When it came to fixing the mechanical problems, we added a timer to the player's jump function, so that the player doesn't fly off the screen. We also changed the sword damage to only subtract 10 from the player's default 100 health.

  As far as software design changes go, we managed to fix most of the problems rather easily, but one of the harder ones that perhaps could still be improved was player movement being laggy across clients. We managed to reduce the lag by increasing the amount of times the event is fired by a client, and then tweening each remote player on each client to the new position rather than transporting them there. With tweening, the tumbleweeds move towards the new position. This improved the game's appearance drastically.

  Lastly, when it came to making the game more enjoyable, we updated our static arena to a dynamic stage with moving cactus walls. This is discussed earlier in the document, but we will briefly summarize it here anyway: we scrapped the arena idea and switched it for the dynamic cactus walls that force players to move through the gaps to survive.
- **Lessons Learned:**
  - How would your approach change in the future?

In the future, we would definitely put more focus on the design documents. We supremely underestimated the importance of prematurely designing the game's architecture. This would have saved us hours of time and effort.

- What advice can you give others based on your experiences?

  To anyone else aspiring to develop a web-based multiplayer game, our advice is simple: take the designing process seriously and maintain an agile environment, since you will constantly be updating your game to meet user expectations.

Improvements:

Some changes that we would have implemented had we more time are:

- add more lives for each player to keep the game going for a longer average time
- add a stall once players first enter the game so that they have more time to get ready
- add a button at the game over screen prompting the player to decide if they want to quit or play again
- add a way for the player to attack and move, similar to the left and right keys on desktop version, on the mobile version of our game
- add a statistics screen at the end of the game reviewing how each player did.

Overall, we, as a team, were very successful. Our overall game design improved drastically as time passed and more deliverables were completed.