Networking Final Project: .io Game

Amanda Goodridge and Will Schottler

November 29, 2019

When thinking of a game for our final project, we decided to create an .io game. The definition of an .io game is that it is free to play, browser-based, very casual with multiplayer components, minimalist graphics and very few mechanisms. The important features of an .io game is that it is free and browser-based and even though there may be a lot of browser-based games out there, there aren't many multiplayer ones. This is what sets .io games apart from other browser-based games. Creating a single-player game that has the mechanics similar to a game like Agar.io isn't very difficult. What is difficult is the networking aspect and deployment.

In order to create an .io game, you have to create dynamic rooms and channels for players to join. In addition, you cannot have thousands of players join in one room; you must divide the players into channels and figure out how to properly network the data between each room and the server without conflict. Keeping track of the player data within each room and updating it is just one small part of the project. You also have to worry about performance and lag as well as ensure that the data is being processed efficiently. Then, when it finally comes time to deploy the .io game, you have to make sure that the deployed version on the Internet works just as well as your local version. During this time, a lot of crashes and bugs can occur as you begin to deploy your game.

Now that I have covered the networking aspects and complexity of .io games, I will go through the implementation of an. io game from scratch. All you need is some knowledge of Javascript including the "this" keyword and promises. The game we created is fairly simple but does require knowledge of networking. You control a ship in an arena with other players and while your ship automatically fires bullets. The goal of the game is to try to hit other players with your bullets while avoiding theirs.

To implement this game, we used Express, the most common web framework for Node.js, to power the web server. We also used socket.io, a websocket library, to communicate the browser and the server. Finally, we used webpack, a module bundler. In order to set up our game, we researched some basic .io games and followed their implementation process because it was our first time creating a networked game. To conclude, we found an example located on Github and followed the same steps as them.

Next, we began the actual game code and to start, we needed an index.html file, which is the first thing the browser will load when someone visits our website. After we implemented that file, we created a networking.js file in which we used the common socket.io library to communicate with the server. This file takes care of all communication with the server. It is important to note that the connectedPromise method resolves only once we've established a connection to the server, if the connection succeeds, we register callbacks for messages we might receive from the server, and we export play() and updateDirection() methods for other files to use.

In addition, we download all the images we need in order to create imaging on the screen. We do this by creating an assets manager called assets.js. Implementing this wasn't too difficult of a task. The main idea of this file is to keep an assets object that maps a filename key to an image object value. Then, once an asset is finished downloading, save it to the assets object for easy retrieval later. After this and once each individual asset download has resolved (meaning all assets have been downloaded), we resolve downloadPromise.

Now that we have downloaded the assets, we began rendering using HTML5 canvas to draw our webpage. Since the webpage is simple, we only need the background, our player's ship, other players in the game, and bullets. This is all implemented under the render.js file. In

addition to the render.js file, we needed an input.js file to make the game playable. This is also very simple as we only need the mouse to control the direction of movement. The final piece to our game is the client state and it is used to give the user the client's current game state at any point in time based on game updates received from the server.

One improvement we could make later down the road is a better implementation of the naïve implementation. The naïve implementation is essentially the worst-case scenario when it comes to lag. If a game update arrives 50 ms late, the client freezes for an extra 50 ms because it's still rendering the game state of the previous update. This causes frustration for the player because of the random freezes and unstable feeling of the game.

However, before we completed our project, we made a couple simple improvements to the naïve implementation of our game so that the lag wouldn't be quite so bad. The first thing we did was use a render delay of 100 ms, meaning the "current" client state will always be 100 ms behind the server's game state. The other improvement we made is using linear interpolation. Because of the render delay, we will usually be at least one update ahead of the current client time. This solves the frame rate problem and we can now render frames as often as we want/need.

When it comes to the backend powering of our game, we focused on client entrypoints, networking, rendering, input, and state. We created a Game class that contains the most important server-side logic and it has the two jobs of managing players and simulating the game. We also created an object.js file with an Object class that includes players and bullets that are actually quite similar due to them both being circular, moving objects. Having the Object class is very beneficial as it prevented us from repeating code.

The final part of our project (and the most difficult) was figuring out collision detection. We had to implement an applyCollisions() method that returns all bullets that hit players. We did this by using a simple math equation. If two circles only "collide," the distance between their centers is less than or equal to the sum of their radii. However, we had to be cautious when making sure a bullet cannot hit the player who created it and a bullet only "hits" once in the edge case when it collides with multiple players at the same time.

Overall, the game implementation took quite some time and a lot of testing but with enough research and online resources to help, we were able to figure it out. We learned a lot about networking fundamentals as well and how to make sure every player who joins is playing on the same network. Personally, I found this part of the project to be the most interesting as it was my first time creating a multiplayer game.

Bibliography

.io games. (2010, February 4). io-games.io. Retrieved from http://io-games.io/.

Lin, A. (2016, December 20). How To Build A Multiplayer Browser Game (Part 1). Retrieved November 21, 2019, from https://hackernoon.com/how-to-build-a-multiplayer-browser-game-4a793818c29b.

Lin, A. (2019, March 19). How To Build A Multiplayer Browser Game (Part 2). Retrieved November 24, 2019, from https://medium.com/hackernoon/how-to-build-a-multiplayer-browser-game-part-2-2edd112aabdf#.imv6b3b33.

NightMayor. (2019, March 13). What are io games? Retrieved November 18, 2019, from https://www.addictinggames.com/what-are-io-games.

Zhou, V. (2019, June 14). How to Build a Multiplayer (.io) Web Game, Part 1. Retrieved November 21, 2019, from https://victorzhou.com/blog/build-an-io-game-part-1/.

Zhou, V. (2019, May 3). How to Build a Multiplayer (.io) Web Game, Part 2. Retrieved November 29, 2019, from https://victorzhou.com/blog/build-an-io-game-part-2/.