Matthew Dolecki, Matthew Thomas, Andrew Melis
ENAE380

Final Project Design

Introduction

Over the course of the fall semester of 2021, the authors of this report developed an online space-themed three-player video game and implemented that game on a custom web server. This report outlines the inspiration behind the game design as well as the software architecture used in its implementation. For access to all code used for the project, visit https://github.com/TomCassey42/System17c.

Inspiration

Our goal for this project was to implement what we had learned about programming throughout the course by making a competitive online multiplayer game. All of us enjoy strategy games and wanted to make an original game to play on a website. We used classic strategy board games such as Catan and Risk as inspiration for our game's mechanics and popular modern online games such as Stellaris as inspiration for the game aesthetics. The theme for the game was chosen because of our collective passion for space exploration. For the game mechanics, we wanted there to be aspects of player interaction, resource generation, and the end goal of controlling the solar system. Making our game on a website allows for multiplayer capabilities and more accessibility. This also served as a good learning experience to better understand data requests between server and browser, as well as script management.

Architecture

As stated above, the primary objective of this project was to develop and host an online game through a web server. For our implementation, we chose to run a Node web server on a remote AWS EC2 instance. In using AWS to host our website on the cloud, we avoided having to set up special network rules through the university IT department, as well as the chore of purchasing and maintaining dedicated hardware for the task. By using a lightweight, non-blocking software like Node to run the server backend over a more traditional industry-standard alternative, such as Apache, we reduced the number of cpu cores as well as the amount of memory required by the EC2 instance, allowing us to run everything on a free-tier EC2 template. A high-level flowchart representation of our architecture is given in Figure 1.

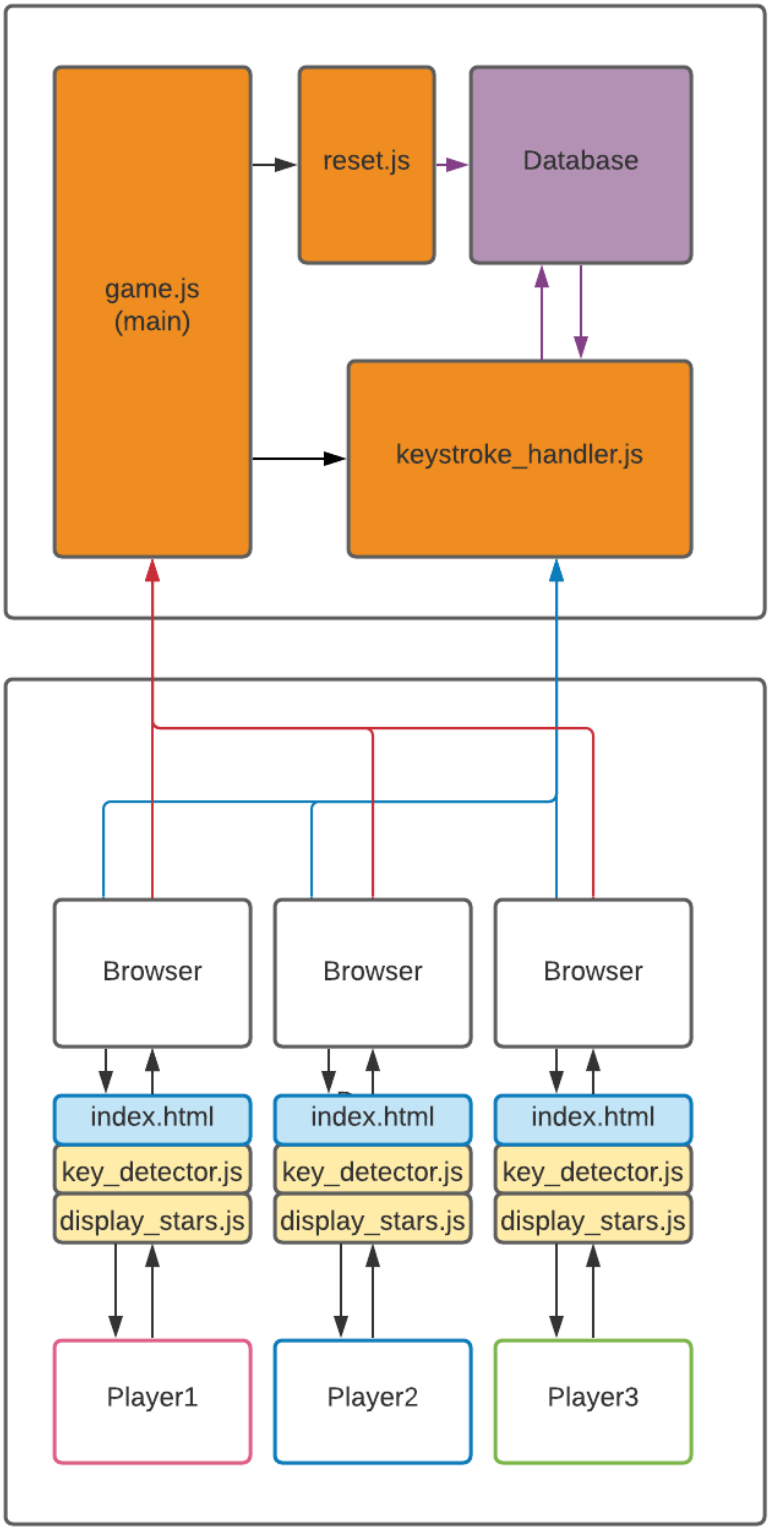
reset.js
Database
game.js
(main)
keystroke_handler.js
Backend
Browser
Browser
Browser
index.html
key_detector.js
display_stars.js
index.html
key_detector.js
display_stars.js
index.html
key_detector.js
display_stars.js
Player1
Player2
Player3
Frontend

Figure 1. System Architecture

The above figure depicts interactions between primary processes using arrows and organizes those processes by runtime environment, with those processes which take place in the backend appearing in the upper box, and those processes which take place in the front end appearing in the lower box.

The Backend

Data flow in the backend is facilitated by four major processes. The first is the game.js script seen in orange in figure 1. This is our primary node script and is what actually runs the node web server. The web server run using this script is configured to listen for GET requests on port 80 and then respond in one of three ways depending on the nature of the request: If the request contains no arguments, then the website's html script is returned to the browser that made the request (this is what happens by default when you type a website's domain, or in our case the EC2 instance's ip address, into the browser). If the request contains the word "reset", then the game.js file spawns a child process, namely reset.js, which resets all game data to pseudo-random initial values. And if the request contains a file name, a file of that name contained on the instance is returned to whoever sent the request. This last response allows us to source resources stored on the EC2 instance from the html script initially being sent to the browser. The game data mentioned above is stored in a MariaDB database and is represented in purple in Figure 1. MariaDB is a fork of the well-known open-source MySQL. This database was chosen because of its compatibility with the node scripts running on the backend (Node has a variety of native sql functions that make connecting to and altering the database trivial). In addition to the reset.js script, the keystroke_handler.js script, spawned by game.js on start and again depicted in orange in figure 1, has the ability to alter game data stored in the database and also acts as a middleman between players and the database during game play. This script listens for websocket connections on port 7071 and alters the game state according to the messages received over said websocket. Messages may include such things as movement commands, buy commands, and requests for data. For specifics on gameplay, see Documentation.

The Front End

While the primary function of the backend is to handle game logic and coordinate the different processes required to host the web server, the primary function of the frontend is to handle direct interaction with the user and then relay that interaction to the backend. In the case of our project, the entirety of the frontend is run in the user's

browser. The scripts required to run the front end are served to the user from the node server running through the game.js discussed above, and include the index.html, key_detector.js and display_stars.js files depicted in light blue and yellow in Figure 1. The index.html file is what is first served to the browser and is interpreted to produce the general layout of the website. It also sources the auxiliary scripts, key_detector.js and display_stars.js mentioned above, from game.js. Both of these scripts communicate directly with the keystroke_handler.js file running on the backend over the aforementioned websocket connection on port 7071. The key_detector.js script makes use of a common open source javascript library, jquery, to respond to keystroke events produced by the user. Information about these keystroke events, including the keystrokes themselves as well as the corresponding intended game actions, are relayed to the keystroke_handler.js scripts which then appropriately updates the game data stored in the database according to the logic of the game. Following any change made to the database, the keystroke_handler.js script will in turn send a message over the websocket relaying these changes to the display_stars.js script. This script then takes this updated information and uses it to update the image being displayed to the user's screen, allowing each player to track the real time movement of other players and planet attributes.

Future Development

In the future we would like to enable massive-multiplayer-online (MMO) capabilities by upgrading to another EC2 instance with more cpu cores and memory and by switching over to a multithreaded web server architecture (Node is single-threaded). We would also like to add new game mechanics like relative planet motion, the ability to upgrade production rates, and the ability to fortify planets against other players. Overall, this project gave us the opportunity to apply our ENAE380 knowledge in a fun and creative way, and we look forward to applying this knowledge in future endeavors.