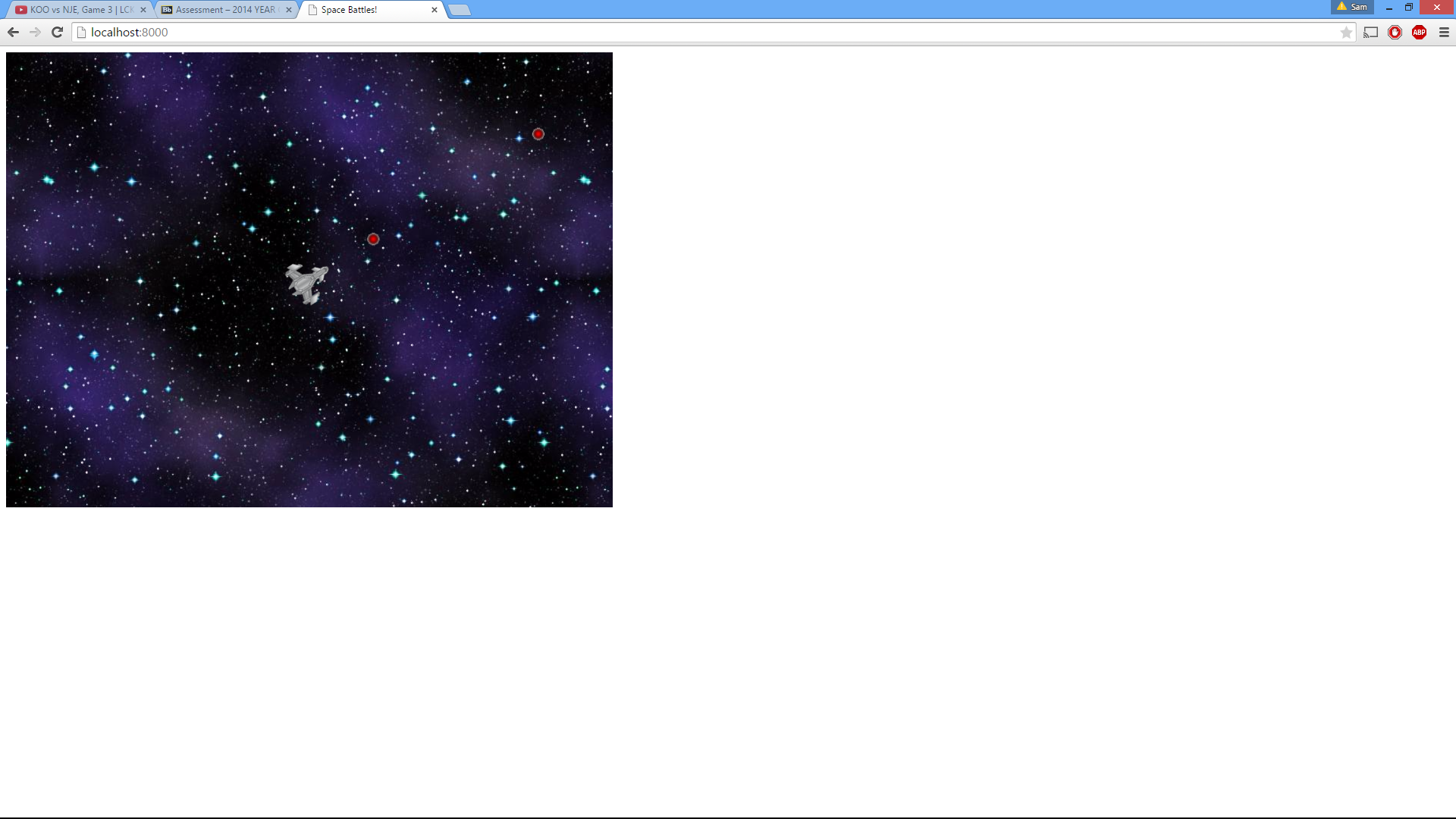
Internet Gaming Design and Development Documentation

Space Battles!

The simple real-time *super* **awesome** spaceship battle game!!



SB684 - 12845703

By Samuel Bowen

Table of Contents

[Design Process 3](#_Toc420618261)

[Documentation 3](#_Toc420618262)

[Client.js 3](#_Toc420618263)

[Server Functions 3](#_Toc420618264)

[The Ship Class 4](#_Toc420618265)

[Phaser Game 4](#_Toc420618266)

[Server.js 5](#_Toc420618267)

[Conclusion 5](#_Toc420618268)

[Installation 7](#_Toc420618269)

[Necessary Software 7](#_Toc420618270)

[Installation 7](#_Toc420618271)

[Running the program 7](#_Toc420618272)

[Key bindings 7](#_Toc420618273)

[Resources 8](#_Toc420618274)

# Design Process

The final product is not what I initially had in mind, at the start of this semester Oliver and myself were originally thinking about designing an RTS (real-time strategy) game which would consist of two players, one creating building to defend their base, and another creating enemy units to attack their opponents base.

Due to personal reasons my group member has deferred and will not be submitting this coursework, therefore I have almost entirely created this project on my own, causing me to go in a different route when designing the game play. As I was lacking man power and felt I didn’t have enough time to create the initial idea on my own, so I decided to make a much simpler game, a more common style of game.

I kept the game real-time as they are much more fun to play and in the end decided to go with an arena spaceship battle game, where the player control’s his or her ship and flies it around the map killing other players! I felt this type of game is much easier to create, but also quite fun, with lots of references and material already available to get ideas from.

# Documentation

I have tried to the best of my ability keep the code clean and readable, I’ve also commented almost everything to make sure that the reader knows what is going in, thus lots of my documentation is actually in the code files so I won’t repeat unnecessary parts here.

## Client.js

Client.js is the file which contains all of the client code, it is used to create the player and his enemies, handle all of the input and deal with all of the game logic, Client.js also creates some server functions which allow them to communicate efficiently. Once Client.js has finished processing the game logic and inputs, it sends all of the updated client information to the server which distributes it back to all of the other clients.

Client.js consists of 3 major parts; the server functions which are created to allow the server to send information to the other clients, the Ship class which is used to create the player and give them all of the necessary variables, and the main body of the game code.

### Server Functions

To handle server communication from the client side I have used a simple Nodejs bidirectional remote procedure call layer called eureca. Eureca allows you to call server side functions from the Nodejs client, and vice-versa.

To set up the eureca server all that was needed was to create an instance of a eureca client, calling the ready functions returns the proxy from the server, allowing us to create an instance of the eureca server. With the eureca server proxy I could now call functions which have been defined on the server side, such as .handshake() and .handleKeys().

On the client side eurecaClient.exports.setId() is created to be used to return the unique identification value of the client that has just connected onto the server. When a player connects to the server this function is called, it returns the ID of the client, then creates the game for the client, handles the server-client handshaking and sets a ready variable to true, allowing the player to start playing the game.

The functions eurecaClients.exports.kill() and eurecaClients.exports.spawnEnemy() are called from the server side to handle new players connecting and disconnecting from the game. When a new player connects and the handshake function passes the spawnEnemy() function is called to create the new enemy in the clients game. When a client disconnects from the server the server calls the kill() function from the client side to kill the player’s ship that has just disconnected.

The final function that is created on the client side is to allow for the current client to receive the position and angle information from another client connected to the server. In essence this function is what update all of the enemy players around you.

### The Ship Class

Within the Phaser create() function the player is initialized by creating an instance of a Ship, the ship constructor takes the ID of the client, the Phaser game variable and a ship variable. Within the constructor the Ship class defines all of the player’s variables and properties. Most of the variables are straight-forward, such as currentSpeed, fireRate, nextFire, x, y, etc… The ship class also handles all of the bullets that the player has shot, holding them in a Phaser group, which makes it easier to detect for collisions.

The constructor also calls the sprite which has been preloaded in the preload Phaser function, and gives the sprites some properties. The Ship class also holds onto state and input variables which are used to detect for key input.

The Ship’s update() function is used to update the position of the ship by handling player input, if there has been a change in the player’s input varibles the updated input is sent to the server to be handled and the new position of the current client is sent to all of the other clients. The update function will then check for player input by using a simple Phaser keyboard handler, game.input.keyboard.createCursorKeys(). After checking for all of the key bindings the position of the player is updated using another Phaser physics function but takes the speed and rotation of the sprite, using its current velocity to figure out where it should move to next.

The Ship’s fire() function uses some simple math to determine the fire rate of the player, if the current game time is greater than that of when the player is next able to fire, the player is allowed to fire. The next fire time is updated and a bullet is created using the Phaser getFirstDead() function, the bullets position is reset to the player’s coordinates and using the way the player’s ship sprite is facing works out the direction the bullet should travel.

### Phaser Game

I have decided to use Phaser, a free open source framework for Canvas and WebGL browers games, seeing as I have decided to use Java Script. Phaser provides a game framework which is quick, easy and simple to use, it’s also very efficient. The framework consists of 4 main functions, preload, create, update and render. Render has only been used for debugging purposes during this project.

The preload() function is used to load in all of the necessary files and assets that are required to play the game. In my case all that I needed to import are the assets and a simple sound file, these are preloaded and then called making asset management much easier throughout the game.

The create() function builds the game world, I use it to set the world’s size and give it some simple properties, here I define the background sprite by calling the preloaded “background” sprite, I can then move it around the game, or fix it to the camera using background.fixedToCamera = true. I create the array of ships that are in the game and then define the audio file. After that I create the player, as explained before the player is an instance of the Ship class, I parse in the variables and then define some game variables using the new player. I set the camera to follow the ship and gave it a deadzone and then created the keyboard input handler, cursors.

Next up is the update() function which is called around 30 times a second, updating the game at around 30fps. At the start of every update call I check to see if the client is ready to have their game updated, this variable is changed when the player connects to the server in the first server function. I then check for player input, the player has been defined as a Ship which contains the input variables, these variables are set to be equal to the keyboard inputs. Later on if these player input variables are different to the ones that have been previously stored the game knows there has been a change in the client’s state, telling the client to update its position on the server.

After the input has been registered I loop through all of the ships to update them, while updating them I also check for collisions between the client’s bullets and the other client’s ships. This collision detection is done using the Phaser overlap function which checks if two sprites are overlapping and then calls functions accordingly. If a player dies then he is randomly spawned someone across the map, and the killer gets credited with a kill.

## Server.js

Most of the Server.js file has been commented to explain exactly what is going on between the client and the server.

On the server side of things there are 2 functions which have been created to detect for clients connecting and disconnecting, and 2 more functions which handle sending positional and input data between the clients and server. The server also holds a list of all of the clients, keeping track of their id’s so that the server can then contact them when ever.

Just like other socket servers, the eureca server extension onConnect() function detects when a user has joined the server, it is called when a new client connects to the server’s IP and port number, in this case I’ve set the port to 8000. When a client connects the server retrieves the id of the client and then uses the client side function setId() to return the unique Id to the client, this allows the server and client to identify each unique player.

The next function is onDisconnect(), it is called every time a client disconnects from the server, once again the id of the client that has disconnected is recorded and another client side function is used, this time kill() to remove and delete the disconnected player’s ship from the client’s game. The client Id is also removed from the server’s list of Ids.

The next function is handshake(), this function is also called when a new client connects as it is called during the setId() function on the client side. It returns all of the currently connected clients to the game to each client, allowing them know where the enemy ships are on the client side. Handshake() calls the client side function spawnEnemy() which is used to create all of the enemy ships on the client.

The last function on the server side is the handleKeys() function, this function is called when a client presses or releases a key, it is used to update the position of that client on all of the other clients. Calling the client side function updateState() the client is able to update the position, angle and cursor state of every other client in the game.

# Gameplay

The game consists of players controlling their own personal spaceship, the game runs in real time and player movements are sent through to each client. The aim of the game is to kill the other players. Using the arrow keys you are able to speed yourself up, slow yourself down and rotate left and right, with the spacebar you are able to shoot directly in front of your ship.

There is a killing spree tracker which updates when you get a kill, also an announcer will tell you that you are on a killing spree when you kill 3 or more players! I attempted to get the announcer to tell all of the other clients that there is an enemy player on a killing spree as well by parsing the id of the player to the other clients, but couldn’t get it to work properly.

# Conclusion

I believe that I have created a pretty average game, it is quite simple but still fun and engaging at the same time. I would have been much happier if I had been able to complete the game we had originally thought of designing at the start, but I am happy with the way the server/client communications have been created and are handled. If I had more time I would have added many more features that could have made this project a really fun game!

# Installation

## Necessary Software

Node - <https://nodejs.org/download/>

Node Package Manager (NPM) – Comes with the Node installation.

*Node is used to run the server.*

## Installation

Open up the command prompt

Navigate to the Game directory (e.g. C:\Users\sam\Desktop\Game)

Enter “npm install”

*Node package manager will install all of the dependencies you’ll see in Game\package.json.*

## Running the program

When in the Game directory enter “node server.js”

Open your browser and go to “localhost:8000”

*The server is listening on port 8000.*

## Key bindings

Up Arrow – Increase Throttle  
Down Arrow – Decrease Throttle  
Left Arrow – Rotate Left  
Right Arrow – Rotate Right  
Spacebar – Shoot!

# Resources

Background Sprite - <https://phaser.io/examples/v2/tile-sprites/tiling-sprite>

Spaceship Sprite - <http://fractalsoftworks.com/forum/index.php?topic=4062.0>

Killing spree sound - <https://www.youtube.com/watch?v=7DsiwBUgmwk>

Enemy killing spree sound - <https://www.youtube.com/watch?v=hDIoDogNGtk>

Example code - <https://phaser.io/examples>