# Game summary

The game is based on the game agar.io which can be found at <http://agar.io/>. The player gets spawned into an appropriate server, one which requires players. A sever will require players when the mass of the food available added to the mass of all the players is less than a certain threshold. The player spawns in as a “blob”, which the objective to become the biggest “blob”. There is no winning sequence to the game, as the player’s aim is to stay at the top of the leaderboard for as long as possible.

The player grows in size by eating food which is spawn by the server in random places, or by eating other players and adding the mass of their body to their own body. The player’s body grows in proportion to its mass, which increases by the size of the object that has just been consumed.

The player moves by moving the mouse over the screen. The “blob” will follow the mouse at a speed that is proportional to the distance the mouse is from the centre of the player. The player can also press “mouse one” to have a speed boost: the player’s body moves much faster in the direction of the mouse, at the expense of losing a lot of mass in the process.

The player’s mass can be viewed at the bottom left of the screen. A player can shoot bits of its own mass at another player, causing them to grow in size, by pressing “SPACE”. If another player absorbs these mass bullets too quickly, they will split apart into multiple blobs and these will spread in multiple directions, and will be out of the game. These blobs can then be eaten as food by other players, again trying to avoid eating too quickly.

Once a player disconnects, their blob remains in the location they were in. Other players can then consume this blob. Players can also be consumed by this blob, so they need to be careful of it.

The main difference of this game to agar.io is the way players interact: players can give other players part of themselves, but players consuming parts of other players too quickly will risk exploding, which is indicated by a progress bar appearing below the player. Players also have the ability to speed-boost to catch other players; agar.io prefers to allow to split apart, with the split part being propelled at some velocity to catch a player.

# Storyboard

# Implementation specification

The game is composed of two parts: the server and the client. The server is also composed of two parts: an **ExpressJS** server that serves the static client files, such as JavaScript and HTML used by the browser to render the gam; and a WebSocket server. All of this server creation is done in src/server/index.js. WebSockets are a common way to develop real-time applications for a browser: “With this API, you can send messages to a server and receive event-driven responses without having to poll the server for a reply.” (Mozilla, 2020). This is the game server that the clients connect to in order to send and receive messages about the game state. This is all handled by the **ServerNetwork** class in the src/server/network.js file. It extends off the Node **EventEmitter** class and will emit events when a client connects, disconnects, etc.

There is also another class like it, **ClientNetwork**, in the src/client/network.js file. This will handle the client connecting to the server. Both the **ClientNetwork** and **ServerNetwork** classes queue messages when they are received in an array. These are then dequeued and processed when the main game update happens. Outgoing messages are also queued until the **flush** method is called, which will send out messages in the order they were queued. The **flush** method will be called on each server tick.

The bulk of the game code that runs on the server is contained in src/server/game.js. When the server is first started, the state of the world is initialised. This involves randomly placing food across the map. Food is randomly placed by the system until the total mass of the world has reached a threshold. The mass of the world is the sum of the all food, players and “projectiles” throughout the map. As the mass of the world approaches this threshold, the rate at which food is spawned slows down. The function **spawnFood** is where the implementation of this algorithm resides.

After the world has been initialised, the server waits for **SPAWN** messages from clients. Once received, a player object will be added to the state of the system. The player is given a random location, using an algorithm which repeatedly selects a random point on the map until it is not within a certain distance of another player (although at the start there are no players). The player is also spawned with a colour and name specified by the client, and with a certain radius in the server’s config. Once created, a **CREATE\_PLAYER** message will be broadcast to each client.

The server is also waiting for the **SET\_VELOCITY** message. This is the client asking the server to update its player’s velocity throughout the map. The server sanitises this and sets it to an appropriate value if too large. The final client message the server is listening for is **SHOOT** which tells the server to create a projectile from the player’s body. In this game, projectiles can be stationary or moving. For instance, when a player disconnects their remaining body is classed as a “projectile”. Projectiles can be consumed by other players, and if consumed too quickly will cause a player to explode. An explosion causes a player’s body to split equally into multiple pieces, which are given some velocity.

The rest of the server is focused on checking if the players are interacting with the world: eating food, other players or projectiles. If two players collide, the bigger one eats the smaller one. Players also lose a small amount of mass on each server tick.

The client code pretty much mirrors the server. The initialisation code for the client is in src/client/index.js. This is the entry point file used by Webpack to bundle all the required modules together. Using Phaser 3, the client functionality is modularised as scenes. The main scene in the client is located in the src/client/main.js file. This scene waits for messages from the server and reacts accordingly, e.g. creating a player blob after the **CREATE\_PLAYER** message is received. It also updates the visuals of the world when receiving messages from the server. The various user interface scenes are located in the src/client/ui folder.

# Network utilisation

Everything that happens in the game is controlled by the server – in other words, it is using an “authoritative” server model. The browser first connects to the server to pull the required static files to render the game. Once loaded, the client connects to the server via a WebSocket connection. The initial handshake is done by the browser and server, and once complete, the client can send and received messages to and from the server. Each client is given a unique identifier (uuidv4) when connected to the server.

All messages are bundled up as JSON data between the client and server. Messages are queued before they are sent and then flushed. When the server receives messages from a client, it adds it to a queue to be processed later, mapped by the unique id of the client that sent it. This id is then used later to update the correct “blob”. These messages are defined below:

|  |  |  |
| --- | --- | --- |
| **Message** | **Description** | **Send from: Client/Server?** |
| SPAWN | Creates a player object and broadcasts it to all connected clients | Client |
| SET\_VELOCITY | Set the velocity of the player | Client |
| SHOOT | Shoot a projectile from the player in the direction of the mouse | Client |
| STATE | Sends the whole state of the world to the client | Server to individual client (message not broadcasted) |
| CREATE\_FOOD | Creates a food object | Server |
| CREATE\_PLAYER | Creates a player object | Server |
| SPAWN\_PROJECTILE | Creates a projectile object | Server |
| UPDATE | Updates a player’s attributes: mass, location | Server |
| DESTROY\_PROJECTILE | Remove a projectile object from the game | Server |
| DESTROY\_FOOD | Remove food object from the game, causing a sound effect to play (to the client only) | Server |
| EXPLODE\_PLAYER | Tells the game to explode a player, causing a sound effect to play (to everyone) | Server |
| UPDATE\_PROJECTILE | Update the attributes of a projectile: position | Server |
| CURRENT\_WORLD\_MASS | Tell the client about the current mass of the system (debug purposes) | Server |

These messages are processed by the client and server in the order in which they were queued; in other words, there is no chance that one sent after another could be processed first. This is the basis of my client-server architecture. To keep the server and client in sync, all messages have to be processed in the correct order.

Client messages are triggered by user input. For instance, the **SET\_VELOCITY** message is triggered by the player moving the mouse somewhere on the screen. The velocity is then calculated by the distance of the mouse from the centre of the blob. Another message triggered by input is **SHOOT**. This happens when the player presses SPACE, causing the server to spawn a projectile in the direction of the blob’s velocity.

Once a player disconnects, a **DESTROY\_PLAYER** message is triggered, along with a **SPAWN\_PROJECTILE** message which essentially creates an uncontrolled blob where the player was originally.

# Critical Review

## Design and implementation

There are three reasons why the design and implementation of the system are good. The first is that the network design is simple: it uses JSON format to send messages, which is a very common format and JavaScript supports it very well; messages are queued and processed. If they were not queued, messages could be processed out of order or on the wrong tick. This could potentially cause discrepancies between what each player sees of the world; it is based on WebSockets, which all modern browsers now support, to transmit data between client and server.

The second reason is that the config of the client and server can quickly be changed by modifying a single file. If I want to adjust the maximum mass a world can have, I can adjust the MAX\_WORLD\_MASS variable. With minimal adjustments, each config variable could be defined by environment variables in the future, so different servers can have different configurations, for instance: allowing more food on one server than another.

The final reason that the

## Improvements

# Assets

explode.aiff (converted to .wav) = <https://freesound.org/people/bareform/sounds/218721/>

eat.wav = <https://freesound.org/people/pan14/sounds/263133/>

shoot.wav = <https://freesound.org/people/V-ktor/sounds/435418/>

# References

# Individual contribution