Technical Guide

Alexander Cahill - 15321711
Liam Ó Chearbhaill - 15384941

Multi User Boids

From a user perspective

User enters MUBS via browser.

User has the option to enter a game room.

Matchmaking will pair two users together at a time

Users then can use touch or mouse cursor to repulse the birds from their own screen and watch as they flock to their opponent's screen. Notice how the birds do not clash with each other and move smoothly, similarly to real birds' flocking behavior.

From a developers perspective

Clone the repo available here https://gitlab.computing.dcu.ie/cahila23/2020-ca400-cahila23-ocearbl2/

This is a Node.js project and therefore you will need a version of node to run this program. We have developed this on Node 12.16.3 and thus you will need this version or newer to avoid versioning issues.

To install all dependencies for development purposes, please run "npm install" before partaking in any work to avoid any other dependency/versioning errors.

start server using 'sudo npm start'

start tests using 'sudo npm run test'

For collaboration get in contact via git.

Link to video walkthrough: https://youtube.com/watch?v=sHIF71n-R1Q
This walkthrough is also available in the video walkthrough section of the repo as requested in the spec. Also, link to that here:
https://drive.google.com/file/d/1dNbQtP33SOT2OoEedj7t-72F0HE3F3in/view?usp=sharing

Motivation

When doing research for the project we were both interested in using software development for graphical displays. A use of programming in graphics that piqued our interest was the “boids algorithm”, originally developed by Craig Reynolds in 1986. The simple rules resulting in a complex, lifelike and impressive display of flocking birds served as inspiration for the project. We wanted to incorporate the boids algorithm, or something similar, and controlling large amounts of graphical objects with simple rules into our project.
Since we already had an interest in developing web-applications and API’s and also an interest in gaming, we felt making an online browser game was the natural direction to go in.

Research

We stumbled upon the boids algorithm while we were researching multiple FYP ideas. We found that this tied in well to our third year project (ant colony simulation) and we were happy to stay on the theme of simulating natural behaviours as we both learned a lot the last time as well as enjoying ourselves along the way.

When researching the boids algorithm we referred alot to Craig Reynolds’ (inventor of the boids algorithm) own webpage on boids[1] to find out how to implement the code and also to Conrad Parker’s website[2] for explanations and some pseudocode. Thanks to these resources we were able to implement our own version of the boids algorithm for our project using Javascript and Three.js, based on the 3 fundamental rules of the algorithm; Separation, Alignment & Cohesion.

Tools

We did some research into what frameworks and tools we would use. We chose to write the program in JavaScript as we both agreed that this was a language we would like to become more proficient in. Python is dynamically typed meaning the type is checked at runtime compared to JavaScript which is type checked at compile time. We both have a lot of experience working with Python and so we both agreed on the point that we would like to gain experience in other languages. Also, it simplifies things to help developers to a point of near complacency which we would like to overcome. It is a simple enough programming language and it is widely used for web applications and scripts online and due to our past experiences we knew it wouldn’t be a large learning curve in order to be able to implement this project.

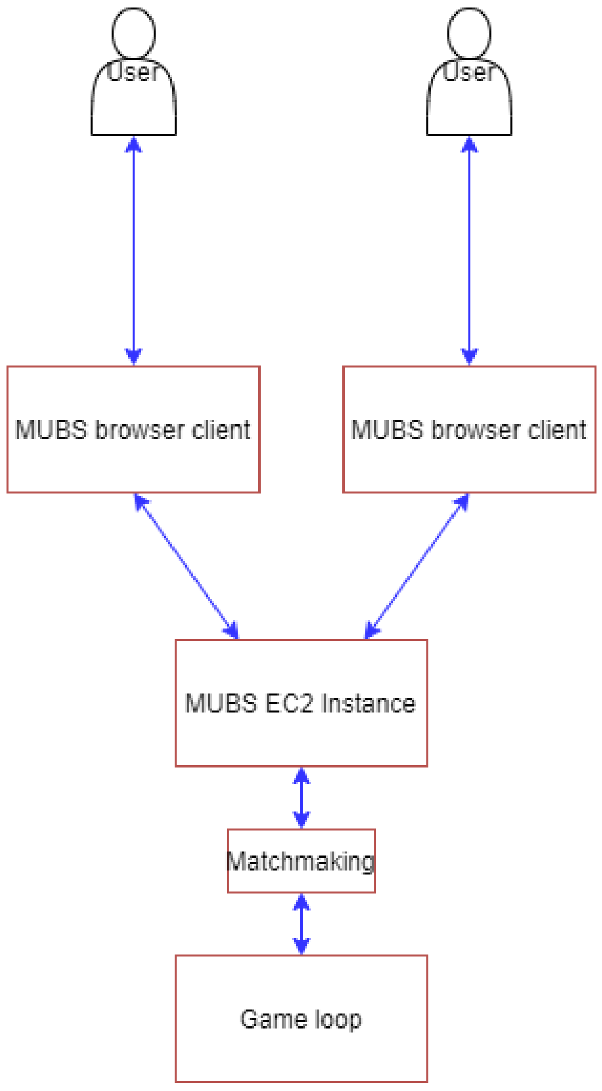
IDE’s we used include JetBrains IntelliJ which is great as it is developed by the same team who created PyCharm and thus the IDE was familiar to us due to the similarities between the two. However, we quickly learned that IntelliJ doesn’t support JavaScript for free, in order to get that support you must pay for the “Ultimate edition” which costs 500 Euros per user. Of course this was a cost neither of us was willing to part with and so we sought out other alternatives. We settled on our own personal preferences which ranged from Atom, - git’s take on a lightweight IDE, and VSCode which is Microsoft's IDE which makes use of user created plugins in order to be able to support almost every form of modern software development.

AWS was used to expose a live IP for the server, a simple EC2 instance was used which was relatively easy to setup and use.

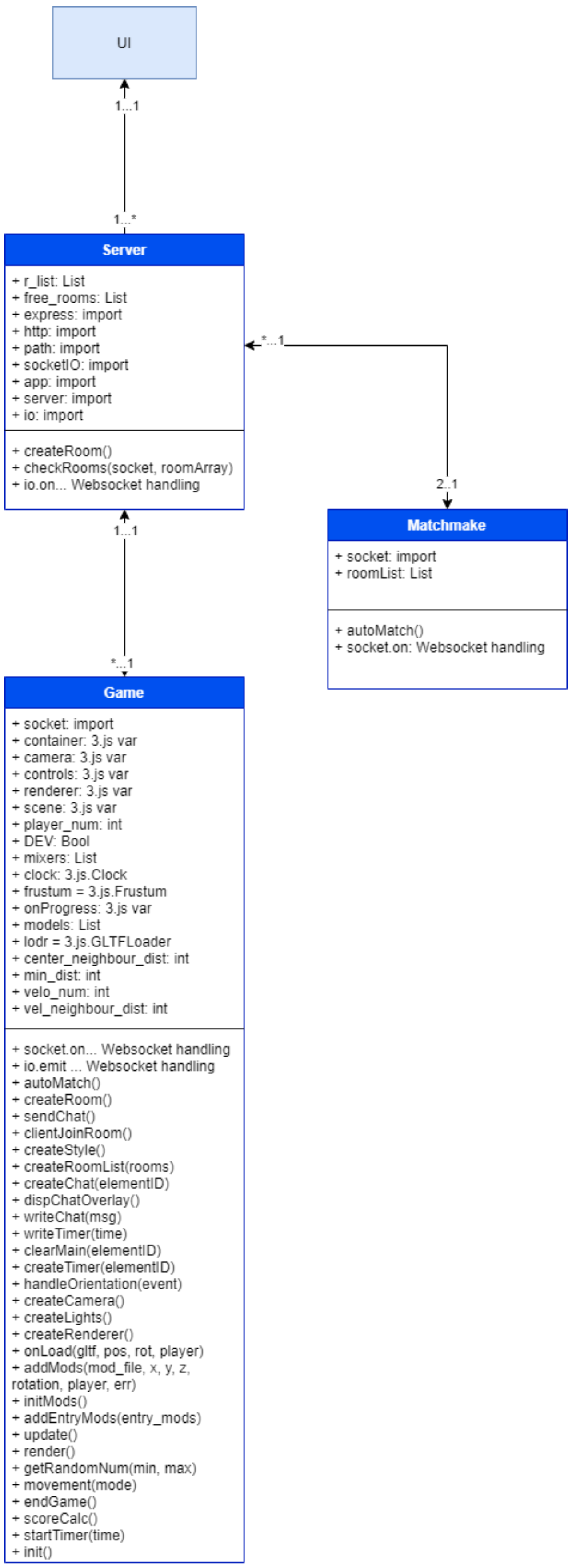
Android Studio is another IDE by JetBrains and it is the industry leader when it comes to Android development nowadays. This is because it allows you to mock an Android device and test your project as if it were any type of Android device. This makes it much easier to test your functionality on a wide variety of devices without having to buy each individual device and manually test on them. This also has the added benefit of reducing hardware costs. This was handy for testing the accelerometer/touch functionality on various devices.

Design & Implementation

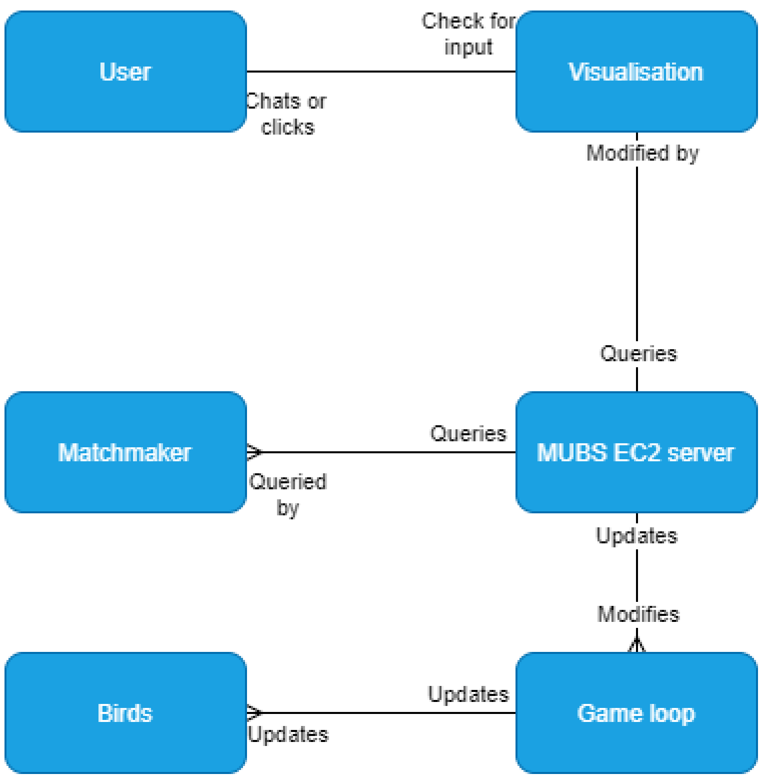
Below is a high level system architecture diagram depicting the relationship of modules with blue arrow-headed lines



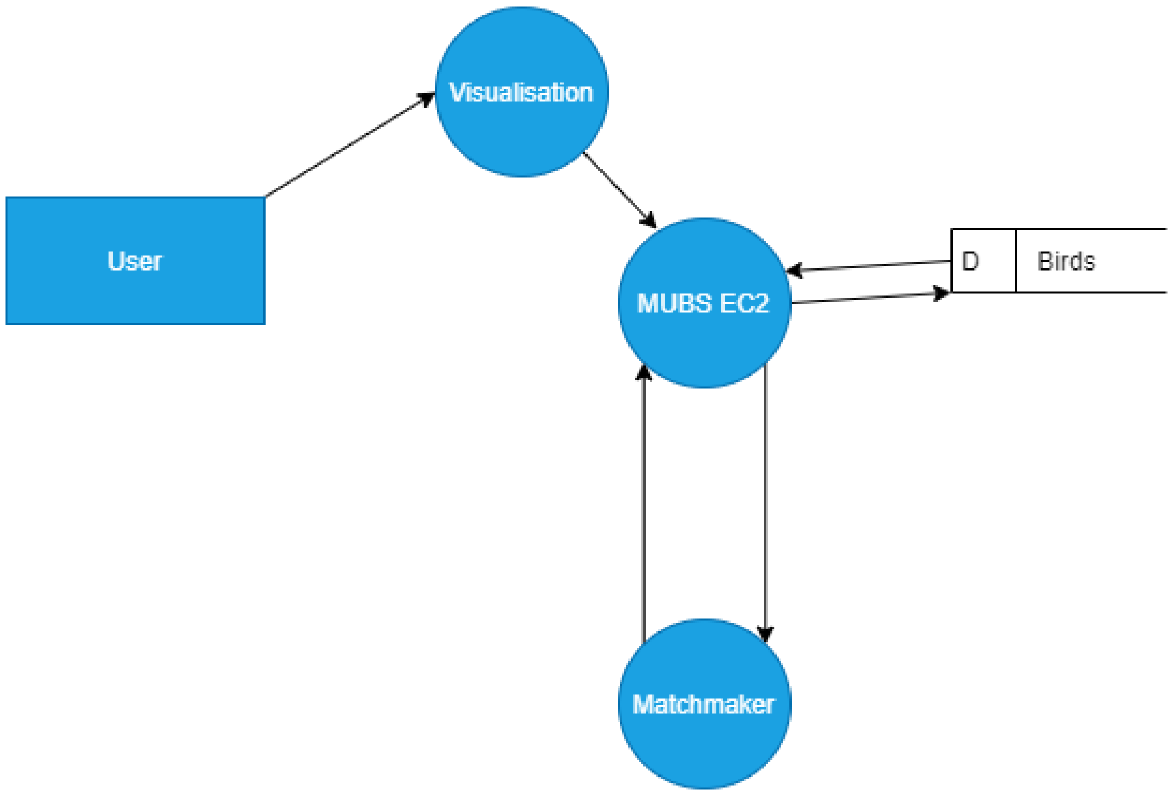
This is a class diagram depicting a high level design layout of our system which includes all of the variables and functions present within MUBS.



Our Logical Data Structure is as follows:



Our Data flow diagram is below:



Client-server-client communication.

All communication between the clients and server is handled by socket.io and express. Express is simply used for routing to serve new html pages to the clients.
Socket.io is used by the server for more nuanced client-server communication and client-client communication.
The server listens for messages when clients are looking to join rooms, get a list of current rooms, when boids transition from one player's environment to their opponent's and vice versa and finally chat messages. The server handles each message in the appropriate way and usually sends a follow up message to a client.

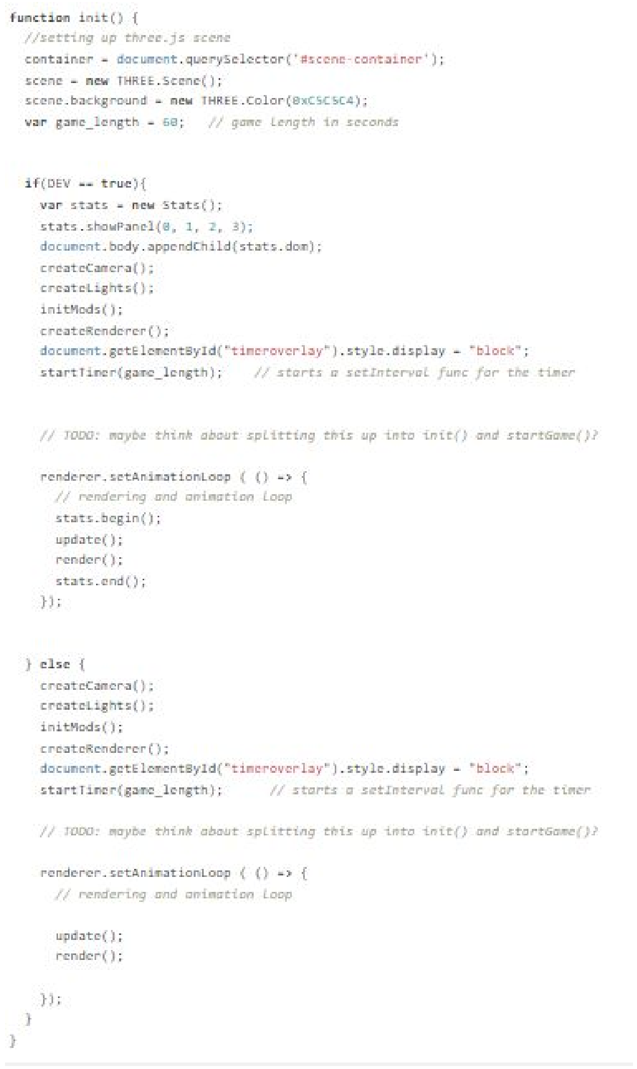
Testing

While there are too many JS testing suites to mention, our testing has been handled by the Mocha testing suite with the Chai assertion library and supplemented with various libraries such as Sinon and jQuery/JSDom for various functions. It is worth noting here that we attempted to use a library called "socket-tester" in an attempt to streamline the development with regards to testing socket functionality. Unfortunately, that library is unfinished and we discovered the hard way that it is unable to assert anything other than 0. Reading the issues on the github page only assert that it was a project the author got bored of or found no more need for. Due to this efforts were made to manually mock up the servers to test users joining a room and various messages in between and thankfully there was some success there.

Each JS file in the project has an accompanying .spec.js file in which we can add tests for various functionality we add. With the use of the --recursive flag Mocha automatically scans and runs all the tests in these files. A problem we encountered within testing was the fact that socket.io wouldn't close connections on the server during testing. By default socket.io should be cleaning up all of the unclosed connections on termination but this wasn't triggering. The aftereach calls to disconnect seemed to be ineffective too and this was causing the Mocha testing suite to hang indefinitely and not return the command line. This was detrimental to the gitlab runner as that stage would run indefinitely until it was either closed manually or the (extremely long) default timeout triggered. The added usage of the --exit flag forces Mocha to ignore the unclosed connections and exit upon the testing loop ending.

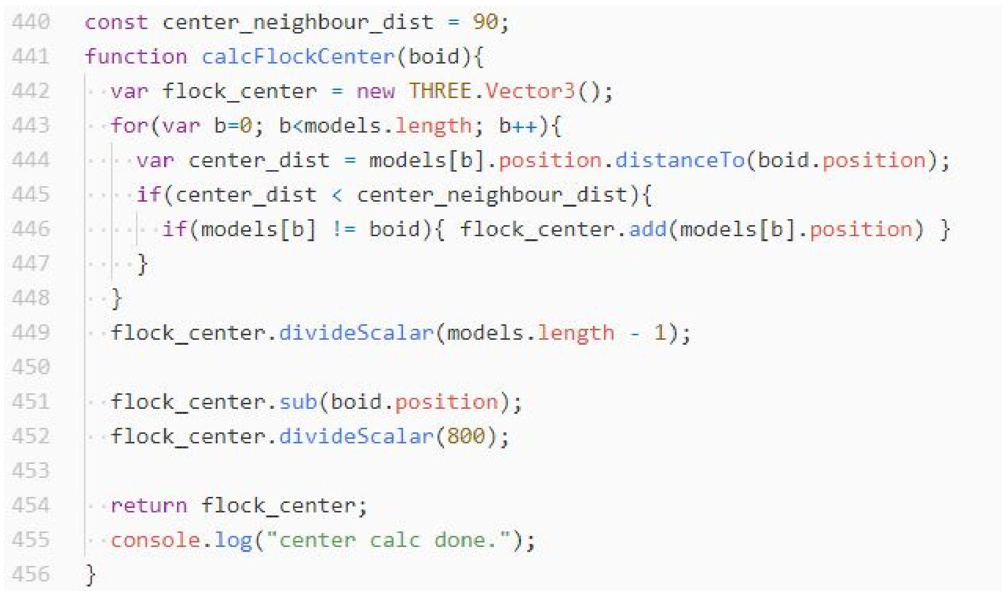
Sample Code

Below is a screenshot of a code snippet consisting of the main game loop inside init(), which is called once when the user receives a “startGame” message from the server. Inside init() is the main loop; setAnimationLoop(). setAnimationLoop() tries to repeat so that update() and render() are called 60 times a second.



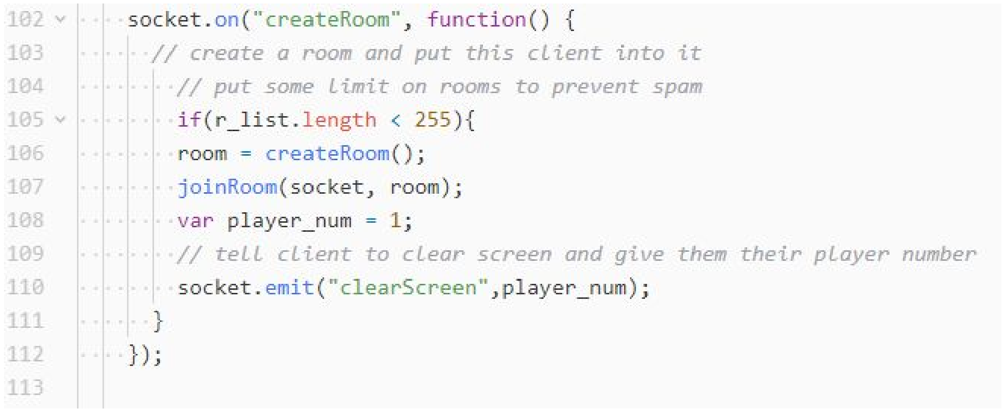
```
function init() {
  //setting up three.js scene
  container = document.querySelector('#scene-container');
  scene = new THREE.Scene();
  scene.background = new THREE.Color(0xC5C5C4);
  var game_length = 60;   // game length in seconds


  if(DEV == true){
    var stats = new Stats();
    stats.showPanel(0, 1, 2, 3);
    document.body.appendChild(stats.dom);
    createCamera();
    createLights();
    initMods();
    createRenderer();
    document.getElementById("timeroverlay").style.display = "block";
    startTimer(game_length);    // starts a setInterval func for the timer


    // TODO: maybe think about splitting this up into init() and startGame()?

    renderer.setAnimationLoop ( () => {
      // rendering and animation loop
      stats.begin();
      update();
      render();
      stats.end();
    });


  } else {
    createCamera();
    createLights();
    initMods();
    createRenderer();
    document.getElementById("timeroverlay").style.display = "block";
    startTimer(game_length);      // starts a setInterval func for the timer

    // TODO: maybe think about splitting this up into init() and startGame()?

    renderer.setAnimationLoop ( () => {
      // rendering and animation loop

      update();
      render();

    });
  }
}
```

Below is one of the boid algorithm rules.
It describes the “cohesion” part of the algorithm. It is called on each boid on every iteration of the main loop.
It starts by checking if the boids around it are within a certain distance from itself.
If they are it then calculates the center point in 3d space of these boids except for itself.
This point in 3d space is taken into account for the next position of the boid. Matrix addition is done after each rule then the resultant point is made the boid’s new position.



```
const center_neighbour_dist = 90;
function calcFlockCenter(boid){
  var flock_center = new THREE.Vector3();
  for(var b=0; b<models.length; b++){
    var center_dist = models[b].position.distanceTo(boid.position);
    if(center_dist < center_neighbour_dist){
      if(models[b] != boid){ flock_center.add(models[b].position) }
    }
  }
  flock_center.divideScalar(models.length - 1);

  flock_center.sub(boid.position);
  flock_center.divideScalar(800);

  return flock_center;
  console.log("center calc done.");
}
```

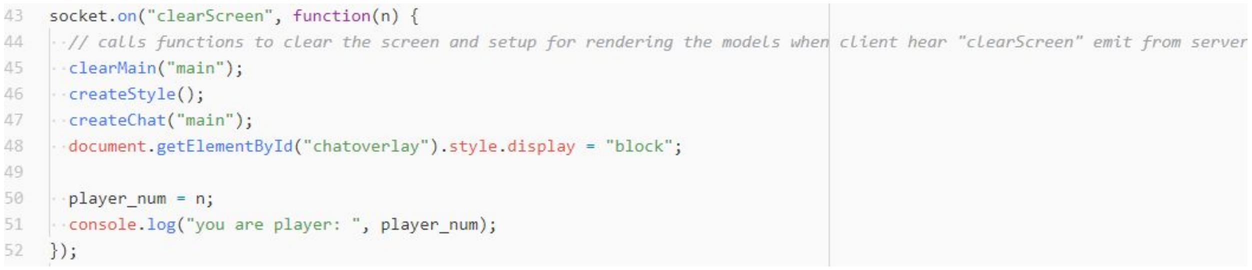
Below is a code snippet showing the emit that is called when a user creates a room.



```
      socket.on("createRoom", function() {
        // create a room and put this client into it
          // put some limit on rooms to prevent spam
          if(r_list.length < 255){
          room = createRoom();
          joinRoom(socket, room);
          var player_num = 1;
          // tell client to clear screen and give them their player number
          socket.emit("clearScreen",player_num);
        }
      });
```

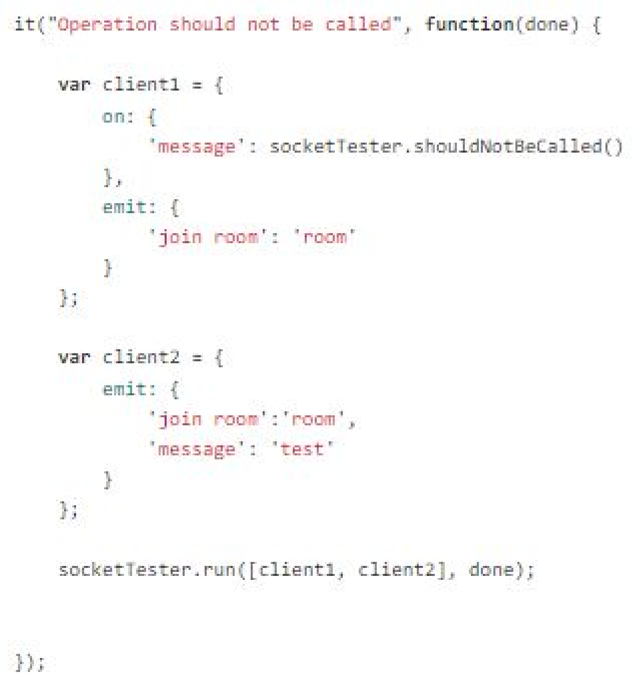
The server listens for a “createRoom” message being emitted by a client. When it hears one it checks that there aren’t too many rooms already then creates a room, places the client in it. After this the server emits a “clearScreen” message along with the client’s player number.

The below snippet shows that clients listen out for a “clearScreen” message, upon hearing one the client sets up, ready for the game to start.



```
socket.on("clearScreen", function(n) {
  // calls functions to clear the screen and setup for rendering the models when client hear "clearScreen" emit from server
  clearMain("main");
  createStyle();
  createChat("main");
  document.getElementById("chatoverlay").style.display = "block";

  player_num = n;
  console.log("you are player: ", player_num);
});
```

Below is a code snippet showing a test for a function that shouldn’t be called. This is the only type of assertion you can consistently do with “socket-tester”.



```
it("Operation should not be called", function(done) {

    var client1 = {
        on: {
            'message': socketTester.shouldNotBeCalled()
        },
        emit: {
            'join room': 'room'
        }
    };

    var client2 = {
        emit: {
            'join room':'room',
            'message': 'test'
        }
    };

    socketTester.run([client1, client2], done);


});
```

Problems Solved

While undertaking the project we ran into a few problems.

Socket.io difficulties.

When carrying out development using socket.io ran into some problems with spare documentation. When initially doing some research for the project socket.io seemed to be a perfect fit for us. When we got further into development we realised that official information on methods, ie what exactly was returned by them, and objects was lacking.
The ways in which functions are used and explained in official documentation was depreciated as well. To find out comprehensive information we had to do research online.

An example of this was in importing socket.io, doing it through the CDN sometimes didn't work and had known vulnerabilities yet was the recommended method of importing on the official socket.io website.

Testing socket.io was especially difficult. As a result of using the recommended method of importing via CDN (which didn't even work) in the HTML, it became impossible to unit test functions within the files that used said import. This was because the import happened at the HTML level whereas the testing was to be for the single files alone. We tried to remedy this by importing and using the socket.io client library and using checks to see if the socket.io library was imported, mocking the entire server and client side, mocking windows with jsdom and we even attempted to use asynchronous calls via jQuery to import the library before importing the functions to be tested. However, none of these fixes worked. Upon reading the files which used the CDN style import it threw errors stating the io() function reference was unknown. As far as the JavaScript compiler was concerned, it had the socket.io library imported so the checks always passed despite the library not being imported within the files. Jsdom and jQuery calls didn't remedy the issue as imports within the $.getScript call are not allowed, and removing them from that call just caused the same issue that led us to find these remedies.

We also noticed that socket failed to close connections properly upon completion and this was causing tests within Mocha to hang indefinitely. This was probably the easiest remedied issue as it is well documented and you simply call --exit on the testing suite which causes the tests to force close after a single iteration through them all.

Unfortunately these problems were realised too late in the project for us to find another method of client-server communication and do the rework to implement it, so we carried on using socket.io. However it did not stop us in implementing any features or providing any services that we wanted socket.io to provide, in fact it works quite well.

Communication Difficulties

Due to the COVID-19 pandemic our physical college term was cut short. This became a large difficulty as both of us live in different counties, with the distance between us being over 75km. This presented communication issues when it came to working on this project as we had planned to be side by side in the labs developing this throughout the year. In order to overcome this we used a mix of Zoom (until the security flaws were made public) and Discord (a persistent server based communication app designed to allow people to hop in and out of server chat rooms at their own will) in order to communicate our progress and design decisions being taken on this project. While we had solved the issue of communication as best we could, there is always that added difficulty of collaboration on a project between people who are miles apart.

Three.js difficulties

As there is a lot of content contained within three js and many modules compatible and made for use with it, there is a lot of documentation for it. Sometimes when trying to solve problems and errors with code, it was hard to find the most up to date documentation.

There are also different ways to tackle certain problems and tasks when rendering and moving objects in a 3D space. For example when applying rotation to an object. It can be rotated in radians or degrees, around a local axis or global axis, by using a combination or just one of quaternions, Euler angles or rotation matrices. There is no agreed “best way” to rotate an object for every use case. So in each case the method which provided the most consistent results, made the most sense to us and was easiest to implement; we used.
Thanks to how the boids’ movement being calculated as movement along the global axes and not rotations and translation along local axes we did not have to use quaternions to avoid gimbal locking.

Solving errors that arose during development were solved most of the time by looking through the three.js documentation[3] or by searching on stackoverflow[4] for a question that had encountered a similar error.

The main loop, setAnimationLoop() only plays when the tab is in focus. This built in to save computation resources so that rendering and updates are not done when the user is not looking at the screen.
When we tried to get around this by using a setInterval() function to call update() and render() 60 times a second three.js still paused animation when the tab was in the background.
We realised this behaviour is built into three.js functions. Unfortunately we weren’t able to find a workaround.

Deprecated libraries & modules

The uuid library became deprecated while we were mid development. This meant that we had to alter our import and usage of the functionality of it in order to maintain optimal functionality. This involved adding the package via node and changing the require call.

Future Work

More controls.(eg accelerometers, extended mobile controls, enhanced UI, accessibility)

Optimise rendering - Rendering could be altered/improved to ensure that MUBS would work effectively on any device which supports WebGL. Currently many platforms support this form of application however not all of these devices will be able to run it as efficiently as the next. Updating the rendering would allow this project to reach a wider audience by allowing MUBS to run smoother on less powerful devices.
Fixing the rendering pausing when the game tab is in the background would be part of this process. This could be achieved by having a check to see if the game tab is in the background. If it is then stop using setAnimationLoop() and use another loop to calculate the position of the boids without rendering them, when the game tab comes back into focus the latest positions would be applied to the boids.

User Interaction - Unfortunately the main “game” aspect of our project was not implemented; players being able to use their mouse to herd the boids into their opponent’s environment. The implementation of this would be done using raycasting a line from the player’s camera towards the mouse through the 3d environment. Boids would avoid an area around the line described by this ray using a modified version of repulseFromOthers(). The closer the boids are to the line, the greater the repulsion effect

Ensuring better consistency - When boids cross from one environment to the other, some disappear completely. This is a result of how coordinates are passed from one client to the other. The boid’s coordinates are flipped when they cross over from environment to environment, this is to ensure that they have space to travel into once they are rendered on the other side. Sometimes when the coordinates are flipped, they end up outside the camera’s viewpoint and are frustum culled immediately getting removed from both rendering stacks in the process. Given more time this could be fixed by having checks on the stack sizes to ensure they are not below the amount they’re supposed to be at.
Also upon crossing over a check could be done by onLoad() to ensure that boids are being loaded into a coordinate that is within the client’s viewpoint and facing towards “open space” i.e. in a direction where it will be continued to be rendered and not immediately culled again.

Chat feature - Currently text written into the chat bar is taken and written directly into the html for the page. This text is not checked to see if it contains code leaving it vulnerable to code injections. This would be easy to fix doing a simple regex check for html open or close tags and replacing them with whitespace or another string.

Optimise server-client communication - Choosing a different framework for client-server communication would increase the maintainability of the code as well as easing the addition of any new features which may need to be added in the future. More than likely this would also increase efficiency and incur lower bandwidth costs which is always favourable.
There is also a bug with our current server-client communication model. When tabs running a game in progress are closed when those tabs are not currently focused on, the server fails to properly end the game and dispose of the room resulting in a stale room being stored on the server. This results in the next person who clicks automatch to be joined into a room and the game being started despite not being shown any rooms initially and no other players in the room.
The server does recognise the client disconnecting though.
This bug could be fixed by the server doing checks on the number of players in a room. The check would be triggered when the number of players in a room changes. If it changes from 1 to 2 then the game should be started. If the number changes from 2 to 1 then the game should be ended, with the remaining player and the room closed and disposed of properly.
A check could be done whenever a client disconnects either. If the disconnecting client was in a room, stop the game in that room, make the remaining player the winner and destroy the room.

In both cases the remaining player would be redirected back to the matchmaking page.

Polishing looks both in game and UI/UX - Better looking applications are just easier on the eye but more importantly good UX keeps users interested as well as showing them the correct paths to choose and easing their use of MUBS. We have chosen functionality over style here but improving on UI/UX is always a good idea.

Would work more on game mechanics - Currently there is not much real point to the game, apart from simulating the BOIDS algorithm and displaying an interactive multi-user environment. This is mostly down to the fact that even though we are both *interested* in games, neither of us are game *design* oriented. Given the chance we would spend more time researching game theories and implement more rewarding features for the user to keep them more both entertained and interested. (For any feature suggestions please get in contact via git.)

References

1. http://www.red3d.com/cwr/boids/ - Craig Reynolds' website. First accessed October 2019
2. http://www.kfish.org/boids/pseudocode.html - Conrad Parker's website. First accessed October 2019
3. http://threejs.org - Official website for the three.js library
4. http://nodejs.org - Official website for Node.js
5. https://stackoverflow.com/ The best forum
6. https://mochajs.org/ Mocha testing suite
7. https://www.chaijs.com/ Chai assertion library