LAB 10 – GOING ONLINE!

In the last 2 labs we created a simple turn-based multiplayer game and we created our own server, even though we cannot use it for this lab, all the prerequisites are there to make our own online game, and this roadblock is actually very important to highlight how important it is to developers to be able to adapt fast to and incredibly fast changing development landscape.

Part 1 – a bit of analysis

Before starting this lab and jumping into the code, it's important to do a bit of software architecture analysys, let's have a look at how our program flow looks like.

There are two different scenarios:

- Host
- Client

The Host will be organizing the game, the Client will join a game created by the Host and follow its rules.

From the Host point of view the game will play like this:

- Click on start game
- A room will be created on the SSMMS server
- A message will be displayed on top of the game to notify the player that we are waiting for a Client to join
- As soon as the client joins, the Host will be notified that the game can start and are able to play

From the Client point of view the game will play like this:

- Click on start game
- A room will be joined from the SSMMS server
- A message will be displayed while the connection to the host is made to give feedback to the player that something is happening while they are waiting and can't play
- As soon as the Host is joined, a message will be sent to the host that the game can start
- The Client will now wait for a move to be made and they will after be able to play

Part 2 – Setup

As per usual, open up Wamp and set it up

Clone your repo from lab 8, the Tic Tac Toe game

If you don't have it, get it from the folder, but then create a repo and publish it before starting the lab.

- Begin by opening the index.html file and add a link to the (SSMMS) Super Simple Multiplayer Messaging System (from here: https://ssmms.herokuapp.com)

```
15      <!-- Load the engine -->
16      <script src="phaser.min.js"></script>
17
18      <!-- Load external libraries -->
19      <script src="https://ssmms.herokuapp.com/api"></script>
20
21      <!-- Load all our scenes here -->
22      <script src="scenes/Preloader.js"></script>
```

All our online functionality will be implemented in the GameScene class, but first let's load in the message to show both players in the PreloaderScene class:

- Open the PreloaderScene class and add this line:

```
29      this.load.image("img_o", "img/O.png");
30
31      //we use this to tell the player that the game will start soon
32      this.load.image("img_waiting", "img/Waiting.png");
33
34
```

Part 3 – Handling Server Messages

We are now ready to get into the meat of this lab, adding online functionality! Let's first do a bit of setup for the game to go multiplayer:

- Open the GameScene class
- Start by adding some variables to regulate the game flow at the top of the class:

```
5  var waitingImage = null;
6  var server = null;
7  var gameStarted = false;
```

- After that, let's present the player with a "Waiting" image, write this in the **Create** function:

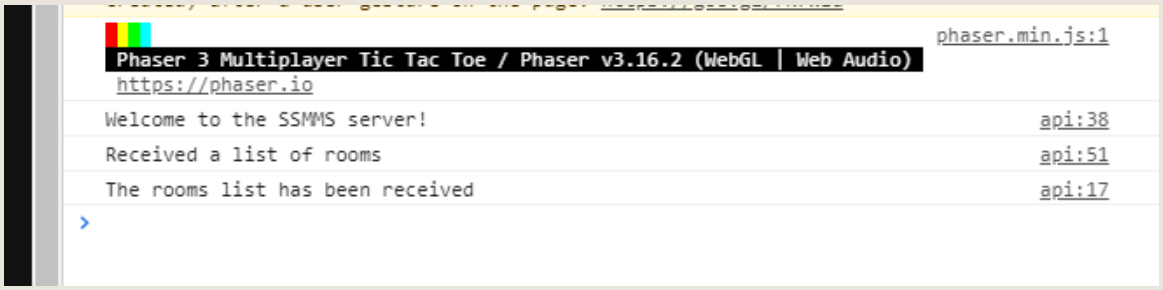
```
26 GameScene.create = function() {
27     //we add the "waiting for opponent" image on top of everything until somebody connects
28     waitingImage = this.add.image(centerX, centerY, 'img_waiting').setOrigin(0.2, 0.2).setDepth(10);
29
```

- Let's connect to the server and request a list of the rooms on it, write this just below the previous line

```
29
30      //init online functionality
31      server = new SSMMS(true);
32
33      server.Connect(server);
34      server.GetRooms();
```

If we now start the game from localhost and open up the console, we can see that a connection is happening, and some debug lines are thrown our way:

More zoomed in:



As you can see from the right, these messages are coming from the **SSMMS api**, this is happening because we don't have yet a way to work with these events, let's change that.

- Just before calling the **Connect** method and requesting the rooms list, add the following lines:

```
33      //connect our handlers
34      server.onMessage = this.MessageReceived;
35      server.onRoomsReceived = this.RoomsReceived;
36      server.onUserDisconnected = this.UserDisconnected;
37      server.onError = this.ErrorReceived;
38
```

Now that we have our handlers connected to some functions, we better start defining what those functions do!

Before writing the next lines, please note that "My Room" is the name that defines your game, please use something else!!!

- At the end of the class, add the following lines:

```
GameScene.RoomsReceived = function(rooms)
{
    console.log("rooms on the server:");
    console.log(rooms);

    var roomExists = false;

    rooms.forEach(function(room){
        if (room.name == "My Room") //somebody already created the room, so we join it
        {
            turn = false;
            server.JoinRoom("My Room");
            roomExists = true;
            //after joining, we notify the host that we have indeed joined, then start the game!
            server.SendMessage("player joined");
            gameStarted = true;
            waitingImage.destroy();
        }
    })

    if (!roomExists) //the room does not exist, we can host!
    {
        server.CreateRoom("My Room", 2); //we can only have 2 players at any given time
        turn = true;
    }
}
```

Take your time to understand what these lines mean. **Think about what we said at the beginning about the Host and the Client!**

- Write the following just under it, to handle what happens when a user disconnects from the playing session:

```
GameScene.UserDisconnected = function()
{
    //the other person left, might as well end the game
    alert("You won by default, the other player left...");
    GameScene.EndLevel();
}
```

- And this, to handle what happens when we receive an error:

```
GameScene.ErrorReceived = function(code, description)
{
    if (code == "CNC") //happens when we cannot create a room
    {
        alert("Unfortunately we could not create a room");
    }

    if (code == "CNJ") //happens when we cannot join a room
    {
        alert("Unfortunately we could not join a room");
    }
}
```

And as the last thing we do before getting into messaging, let's add a way to leave the room once we have finished playing:

- Add the following lines inside the **EndLevel** function

```
155  GameScene.EndLevel = function()
156  {
157      //reset variables
158      tiles = [];
159      tilesValues = [];
160      turn = true;
161
162      //let's leave the room since we have finished playing
163      server.LeaveRoom("My Room");
164
165      this.scene.start("Boot");
166  }
167
```

Wow, that was a lot of stuff, or was it?

We basically just added some functions that respond to certain events that happen on a remote server, this is called **asynchronous programming** or **event driven programming**.

If you are going to work on more online games, remember this structure as it is very common in online environments!

Part 4 – Messaging the other player

First, we are going to add some extra functionality to what happens when we click on a tile.

- Let's go back to the **Create** method and modify the **on pointerdown** event of the tiles when they are created like so:

```
65            if (gameStarted && turn) //we make a move only if the game is started and it is our turn!
66            {
67                GameScene.MakeMove(element.x, element.y);
68                //instead of just making a move, we also send it to our opponent
69                server.SendMessage("move", element);
70
71                //after sending the message, we check if it was the last move
72                GameScene.CheckForWin();
73
74                //finally swap turn
75                turn = !turn;
76            }
```

You may notice that the **CheckForWin** and the **turn** swapping is now handled here, so we need to remove this functionality from the **MakeMove** function.

- Go in the **MakeMove** function and remove those lines so that it looks like this:

```
187             tilesValues[locationX][locationY] = 2;
188         }
189
190         //we move the win check to after we sent the move to our opponent!
191     }
192
```

Finally, let's create the **MessageReceived** method somewhere in the class, best at the end of it.

- Write the following lines at the end of the file:

```
86  GameScene.MessageReceived = function(type, message){
87      console.log("received message type: " + type + ", with value: " + message);
88
89      if (type == "player joined")
90      {
91          gameStarted = true;
92          waitingImage.destroy();
93      }
94
95      if (type == "move")
96      {
97          //we get these from our opponent!
98          GameScene.MakeMove(message.x, message.y);
99
100         //after receiving the message, we check if it was the last move
101         GameScene.CheckForWin();
102
103         //finally swap turn
104         turn = !turn;
105     }
106 }
```

Conclusion

In this lab, we took the game created in **Lab 8** and made it an online multiplayer game by using an external library, hopefully at this point you understand how to write **asynchronous** and **event driven** code to make your own **turn-based multiplayer** games!