Chapter 12

Multiplayer Gameplay

In the previous chapter, we saw how the WebSocket API could be used with a Node.js server to implement a simple client–server networking architecture. We used this to build a simple game lobby, so two players can now join a game room on a server and start a multiplayer game against each other.

In this chapter, we will continue where we left off at the end of Chapter 11 and implement a framework for the actual multiplayer gameplay using the lock-step networking model. We will look at ways to handle typical game networking problems such as latency and game synchronization. We will then use the `sendCommand()` architecture that we designed in earlier chapters to ensure that the players' commands are executed on both the browsers so that the games stay in sync. We will then implement winning and losing in the game by using triggers like we did in Chapter 10. Finally, we will implement a chat system for our game.

Let's get started.

The Lock-Step Networking Model

So far, we used the Node.js server to communicate simple messages such as the game lobby status and joining or leaving a room. These messages were independent of each other, and one player's messages did not affect another player. However, when it comes to the gameplay, this communication is going to get a little more complex.

One of the most important challenges in building a multiplayer game is to ensure that all the players are in sync. This means every time a change occurs in any of the games (for example, a player issues a move or attack command), the change is communicated to the other players so that they too can make the same change.

What is even more important is that the action or change occurs at the same moment in both the players' machines. If there is a delay in executing these changes, subtle differences in unit positions will eventually build up, resulting in noticeable divergences between the game states.

For example, a unit that is just half a second late in arriving at an enemy location might avoid an enemy attack and survive in one browser, while the same unit may have been destroyed on the other player's browser. The moment something like this happens, the two players are now playing two completely different games instead of the same one.

To ensure that both players are completely in sync, we will implement an architecture known as the *lock-step* networking model. Both players will start with the same game state. When the player gives a unit a command, we will send the command to the server instead of executing it immediately. The server will then send the same command to the connected players with instructions on when to execute the command. Once the players receive the command, they will execute it at the same time, ensuring that the games stay synchronized.

The server will achieve this behavior by running its own game timer, at 10 clock ticks per second. When a player sends the server a command, the server will record the clock tick when it received the command. The server will then send the command to the players, while specifying the game tick to execute the command. The players in turn will keep track of the current game tick and execute the command at the right tick.

One thing to remember is that since the server needs to execute the commands for all the players at the same time, it will need to wait for the commands from all the players to arrive before stepping ahead to the next game tick, which is why it's called *lock-step*.

This process is further complicated by the fact that network latency can cause communication delays, with messages sometimes taking several hundred milliseconds to travel between client and server. Our networking model will need to measure and take this latency into account to ensure smooth gameplay.

We will start by modifying our game code to measure the network latency for each player when the player first connects to the server.

Measuring Network Latency

For our purposes, we will define the latency as the time taken by a message to travel from the server to the client. We will measure this latency by sending several messages back and forth between the server and the client and then taking an average of the time used for each trip.

We will start by defining two new methods for measuring the latency inside `server.js`, as shown in Listing 12-1.

Listing 12-1. Methods for Measuring Network Latency (server.js)

```
function measureLatency(player){
    var connection = player.connection;
    var measurement = {start:Date.now()};
    player.latencyTrips.push(measurement);
    var clientMessage = {type:"latency_ping"};
    connection.send(JSON.stringify(clientMessage));
}
function finishMeasuringLatency(player,clientMessage){
    var measurement = player.latencyTrips[player.latencyTrips.length-1];
    measurement.end = Date.now();
    measurement.roundTrip = measurement.end - measurement.start;
    player.averageLatency = 0;
    for (var i=0; i < player.latencyTrips.length; i++) {
        player.averageLatency += measurement.roundTrip/2;
    };
    player.averageLatency = player.averageLatency/player.latencyTrips.length;
    player.tickLag = Math.round(player.averageLatency * 2/100)+1;
    console.log("Measuring Latency for player. Attempt", player.latencyTrips.length, "-
Average Latency:",player.averageLatency, "Tick Lag:", player.tickLag);
}
```

In the `measureLatency()` method, we first create a new `measurement` object with a `start` property set to the current time and add the object to the `player.latencyTrips` array. We then send a message of type `latency_ping` to the player. The player will respond to this message by sending back a message of type `latency_pong`.

In the `finishMeasuringLatency()` method, we take the last measurement from the `player.latencyTrips` array and set its `end` property to the current time and its `roundTrip` property to the difference between the end and start times.

We then calculate the average latency for the player by adding up all the `roundTrip` values and then dividing the sum by the number of trips.

Finally, we use `averageLatency` to calculate a `tickLag` property for the player. This is the number of ticks after sending a command that the player can be safely expected to have received the command. The heuristic uses a value that is 200 percent of the typical latency, with a minimum value of one game tick.

You can play around with this heuristic and fine-tune it for accuracy if you like; however, for the purposes of smooth gameplay, it is safer to have a high value. It has been found that players are able to get used to network lag and automatically adjust for it as long as the delay is consistent. Any time the lag varies too much, players tend to get frustrated by it.

Next we will modify the multiplayer object's handleWebSocketMessage() method to respond to the server's latency_ping message, as shown in Listing 12-2.

Listing 12-2. Responding to latency_ping with a latency_pong (multiplayer.js)

```
handleWebSocketMessage:function(message){
    var messageObject = JSON.parse(message.data);
    switch (messageObject.type){
        case "room_list":
            multiplayer.updateRoomStatus(messageObject.status);
            break;
        case "joined_room":
            multiplayer.roomId = messageObject.roomId;
            multiplayer.color = messageObject.color;
            break;
        case "init_level":
            multiplayer.initMultiplayerLevel(messageObject);
            break;
        case "start_game":
            multiplayer.startGame();
            break;
        case "latency_ping":
            multiplayer.sendWebSocketMessage({type:"latency_pong"});
            break;
    }
},
```

When the browser receives a latency_ping message from the server, it immediately sends back a latency_pong message to the server.

Finally, we will modify the request event handler for the websocket object on the server to start measuring latency when a player connects and to finish measuring latency when the player sends back a latency_pong response, as shown in Listing 12-3.

Listing 12-3. Starting and Finishing Latency Measurement (server.js)

```
wsServer.on('request',function(request){
    if(!connectionIsAllowed(request)){
        request.reject();
        console.log('Connection from ' + request.remoteAddress + ' rejected.');
        return;
    }

    var connection = request.accept();
    console.log('Connection from ' + request.remoteAddress + ' accepted.');

    // Add the player to the players array
    var player = {
        connection:connection,
        latencyTrips:[]
    }
    players.push(player);
```

```
    // Send a fresh game room status list the first time player connects
    sendRoomList(connection);

    // Measure latency for player
    measureLatency(player);

    // On Message event handler for a connection
    connection.on('message', function(message) {
        if (message.type === 'utf8') {
            var clientMessage = JSON.parse(message.utf8Data);
            switch (clientMessage.type){
                case "join_room":
                    var room = joinRoom(player,clientMessage.roomId);
                    sendRoomListToEveryone();
                    if(room.players.length == 2){
                        initGame(room);
                    }
                    break;
                case "leave_room":
                    leaveRoom(player,clientMessage.roomId);
                    sendRoomListToEveryone();
                    break;
                case "initialized_level":
                    player.room.playersReady++;
                    if (player.room.playersReady==2){
                        startGame(player.room);
                    }
                    break;
                case "latency_pong":
                    finishMeasuringLatency(player,clientMessage);
                    // Measure latency at least thrice
                    if(player.latencyTrips.length<3){
                        measureLatency(player);
                    }
                    break;
            }
        }
    });

    connection.on('close', function(reasonCode, description) {
        console.log('Connection from ' + request.remoteAddress + ' disconnected.');
        for (var i = players.length - 1; i >= 0; i--){
            if (players[i]==player){
                players.splice(i,1);
            }
        };

        // If the player is in a room, remove him from room and notify everyone
        if(player.room){
            var status = player.room.status;
            var roomId = player.room.roomId;
```

```
            leaveRoom(player,roomId);

            sendRoomListToEveryone();
        }
    });
});
```

We start by adding a `latencyTrips` array to the `player` object and calling `measureLatency()` once the player has connected.

We then modify the message handler to handle messages of type `latency_pong`. When the player responds to a `latency_ping` message with a `latency_pong` message, we call the `finishMeasuringLatency()` method that we defined earlier. We then check whether we have at least three latency measurements and, if not, call the `measureLatency()` method again.

Now, if you start the server and run the game, the server will make three attempts to measure the latency. You can see the Websocket communication using the browser's developer console, as shown in Figure 12-1.

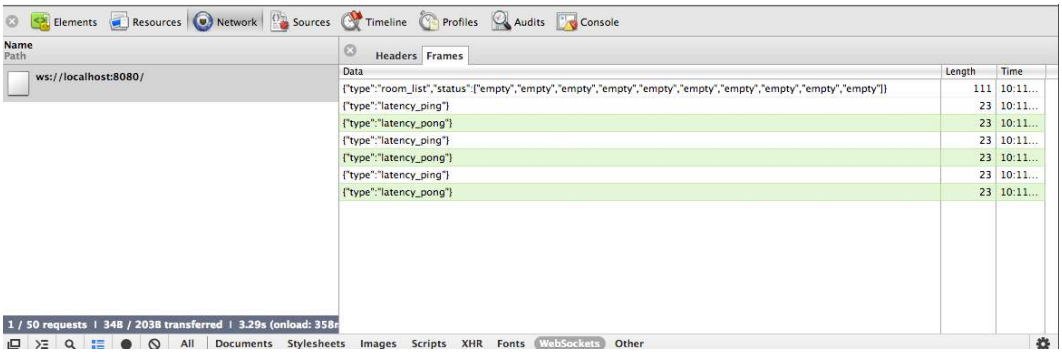

Figure 12-1. *Observing the websocket communication in the developer console*

Now that we have measured latency for the players, it's time to implement sending commands.

Sending Commands

Once the game starts, we will maintain a game clock with a game tick number on both the server and the clients. When a player sends a command to the server, we will send the command back to the clients with instructions to execute the command at a later tick calculated by using `tickLag`.

We will start by modifying the `multiplayer` object's `handleWebSocketMessage()` method to receive commands within a `game_tick` message, as shown in Listing 12-4.

Listing 12-4. Receiving Commands in game_tick Message (multiplayer.js)

```
handleWebSocketMessage:function(message){
    var messageObject = JSON.parse(message.data);
    switch (messageObject.type){
        case "room_list":
            multiplayer.updateRoomStatus(messageObject.status);
            break;
```

```
        case "joined_room":
            multiplayer.roomId = messageObject.roomId;
            multiplayer.color = messageObject.color;
            break;
        case "init_level":
            multiplayer.initMultiplayerLevel(messageObject);
            break;
        case "start_game":
            multiplayer.startGame();
            break;
        case "latency_ping":
            multiplayer.sendWebSocketMessage({type:"latency_pong"});
            break;
        case "game_tick":
            multiplayer.lastReceivedTick = messageObject.tick;
            multiplayer.commands[messageObject.tick] = messageObject.commands;
            break;
    }
},
```

When we receive a `game_tick` message from the server containing a list of commands and the tick number on which the commands need to be executed, we save the commands in the `multiplayer.commands` array and then update the `lastReceivedTick` variable.

Next we will implement the game loop and handle sending commands, as shown in Listing 12-5.

Listing 12-5. Sending Commands from the Client (multiplayer.js)

```
startGame:function(){
    fog.initLevel();
    game.animationLoop();
    multiplayer.animationInterval = setInterval(multiplayer.tickLoop, game.animationTimeout);
    game.start();
},
sendCommand:function(uids,details){
    multiplayer.sentCommandForTick = true;
    multiplayer.sendWebSocketMessage({type:"command",uids:uids,
details:details,currentTick:multiplayer.currentTick});
},
tickLoop:function(){
    // if the commands for that tick have been received
    // execute the commands and move on to the next tick
    // otherwise wait for server to catch up
    if(multiplayer.currentTick <= multiplayer.lastReceivedTick){
        var commands = multiplayer.commands[multiplayer.currentTick];
        if(commands){
            for (var i=0; i < commands.length; i++) {
                game.processCommand(commands[i].uids,commands[i].details);
            };
        }
```

```
        game.animationLoop();

        // In case no command was sent for this current tick, send an empty command to the server
        // So that the server knows that everything is working smoothly
        if (!multiplayer.sentCommandForTick){
            multiplayer.sendCommand();
        }
        multiplayer.currentTick++;
        multiplayer.sentCommandForTick = false;
    }
},
```

First, in the `startGame()` method, we set an interval to call the `tickLoop()` method every 100 milliseconds when the game starts.

Next, in the `sendCommand()` method, we send a message of type `command` to the server with the details of the command as well as the UIDs for the command.

The command message also contains the current game tick. This way, the command message acts as a heartbeat to let the server know what game tick the client is currently on. We also set the `sendCommandForTick` flag to true.

In the `tickLoop()` method, we check to see whether we have received commands for the current tick. In case we have not, we will wait for the commands to arrive from the server.

If we have received the commands for the tick, we process all the received commands using the `game.processCommand()` method. We then call the `game.animationLoop()` method.

In case we have not sent out any commands so far, we also send an empty command to the server.

Finally, we increment the game tick number and clear the `sentCommandForTick` flag.

Now that the client has been modified to send and receive commands, we will modify the server to handle these commands as well.

We will start by modifying the message event handler on the server to handle messages of type `command`, as shown in Listing 12-6.

Listing 12-6. Handing Messages of Type command (server.js)

```
// On Message event handler for a connection
connection.on('message', function(message) {
    if (message.type === 'utf8') {
        var clientMessage = JSON.parse(message.utf8Data);
        switch (clientMessage.type){
            case "join_room":
                var room = joinRoom(player,clientMessage.roomId);
                sendRoomListToEveryone();
                if(room.players.length == 2){
                    initGame(room);
                }
                break;
            case "leave_room":
                leaveRoom(player,clientMessage.roomId);
                sendRoomListToEveryone();
                break;
            case "initialized_level":
                player.room.playersReady++;
                if (player.room.playersReady==2){
                    startGame(player.room);
                }
```

```
                break;
            case "latency_pong":
                finishMeasuringLatency(player,clientMessage);
                // Measure latency at least thrice
                if(player.latencyTrips.length<3){
                    measureLatency(player);
                }
                break;
            case "command":
                if (player.room && player.room.status=="running"){
                    if(clientMessage.uids){
                        player.room.commands.push({uids:clientMessage.uids,
details:clientMessage.details});
                    }
                    player.room.lastTickConfirmed[player.color] = clientMessage.currentTick +
player.tickLag;
                }
                break;
        }
    }
});
```

When the server receives a message of type `command`, we check whether the message has UIDs. If so, we store the commands in the room's `commands` array. If not, the message is just a heartbeat message with no command that needs saving. We then update the `lastTickConfirmed` property for the player.

Next, we will modify the `startGame()` method in `server.js`, as shown in Listing 12-7.

Listing 12-7. Modifying the startGame() Method (server.js)

```
function startGame(room){
    console.log("Both players are ready. Starting game in room",room.roomId);
    room.status = "running";
    sendRoomListToEveryone();
    // Notify players to start the game
    sendRoomWebSocketMessage(room,{type:"start_game"});

    room.commands = [];
    room.lastTickConfirmed = {"blue":0,"green":0};
    room.currentTick = 0;

    // Calculate tick lag for room as the max of both player's tick lags
    var roomTickLag = Math.max(room.players[0].tickLag,room.players[1].tickLag);

    room.interval = setInterval(function(){
        // Confirm that both players have send in commands for up to present tick
        if(room.lastTickConfirmed["blue"] >= room.currentTick &&
room.lastTickConfirmed["green"] >= room.currentTick){
            // Commands should be executed after the tick lag
            sendRoomWebSocketMessage(room,{type:"game_tick",
tick:room.currentTick+roomTickLag, commands:room.commands});
            room.currentTick++;
            room.commands = [];
```

```
        } else {
            // One of the players is causing the game to lag. Handle appropriately
            if(room.lastTickConfirmed["blue"] < room.currentTick){
                console.log ("Room",room.roomId,"Blue is lagging on
Tick:",room.currentTick,"by", room.currentTick-room.lastTickConfirmed["blue"]);
            }
            if(room.lastTickConfirmed["green"] < room.currentTick){
                console.log ("Room",room.roomId,"Green is lagging on Tick:",
room.currentTick, "by", room.currentTick-room.lastTickConfirmed["green"]);
            }
        }
    },100);
}
```

When the game starts, we initialize the `commands` array, `currentTick`, and the `lastTickConfirmed` object for the room. We then calculate the tick lag for the room as the maximum of the tick lag for the two players and save it in the `roomTickLag` variable.

We then start the timer loop for the game using `setInterval()`. Within this loop, we first check that both players have caught up with the server by sending commands for the present game tick.

If so, we send out a `game_tick` message to the message to the players with a list of commands and ask them to execute the commands `roomTickLag` ticks after the current tick. This way, both the players will execute the command at the same time, even if the message takes a little time to reach the players.

We then clear the `commands` array on the server and increase the `currentTick` variable for the room.

If the server hasn't received confirmation for the current tick from both the clients, we log a message to the console and do not increment the tick. You can modify this code to check whether the server has been waiting for a long time and, if so, send the players a notification that the server is experiencing lag.

If you start the server and run the game on two different browsers, you should be able to command the units and have your first multiplayer battle, as shown in Figure 12-2.

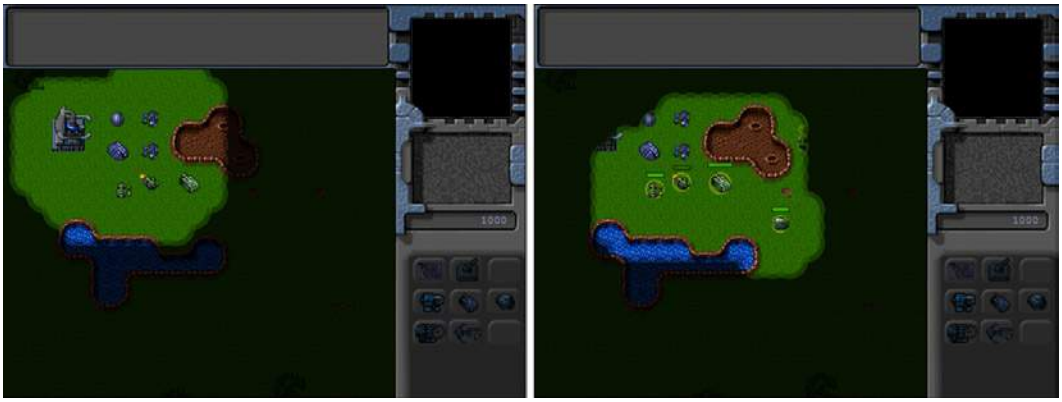

Figure 12-2. *Commanding units in a multiplayer battle*

The multiplayer portion of our game is now working. Right now both the browsers are on the same machine. You can move the server code onto a separate Node.js machine and modify the `multiplayer` object to point to this new server instead of localhost. If you want to move to a public server, you can find several hosting providers that provide Node.js support such as Nodester (`http://nodester.com`) and Nodejitsu (`http://nodejitsu.com/`).

Now that we have implemented sending commands, we will implement ending the multiplayer game.

Ending the Multiplayer Game

The multiplayer game can be ended in two ways. The first is if one of the players defeats the other by satisfying the requirements for the level. The other is if a player either closes the browser or gets disconnected from the server.

Ending the Game When a Player Is Defeated

We will implement ending the game using triggered events just like we did in Chapter 10. This gives us the flexibility to design different types of multiplayer levels such as capture the flag or death match. We are limited only by our imagination.

For now, we will make the level end when one side is completely destroyed. We will start by creating a simple triggered event in the multiplayer map, as shown in Listing 12-8.

Listing 12-8. Trigger for Ending the Multiplayer Level (maps.js)

```
/* Conditional and Timed Trigger Events */
"triggers":[
    /* Lose if not even one item is left */
    {"type":"conditional",
        "condition":function(){
            for (var i=0; i < game.items.length; i++) {
                if(game.items[i].team == game.team){
                    return false;
                }
            };
            return true;
        },
        "action":function(){
            multiplayer.loseGame();
        }
    },
]
```

In the conditional trigger, we check whether the `game.items` array contains at least one item belonging to the player. If the player has no items left, we call the `loseGame()` method.

Next we will add the `loseGame()` and `endGame()` methods to the `multiplayer` object, as shown in Listing 12-9.

Listing 12-9. Adding loseGame() and endgame() Methods (multiplayer.js)

```
// Tell the server that the player has lost
loseGame:function(){
    multiplayer.sendWebSocketMessage({type:"lose_game"});
},
endGame:function(reason){
    game.running = false
    clearInterval(multiplayer.animationInterval);
    // Show reason for game ending, and on OK, exit multiplayer screen
    game.showMessageBox(reason,multiplayer.closeAndExit);
}
```

In the `loseGame()` method, we send a message of type `lose_game` to the server to let it know that the player has lost the game.

In the `endGame()` method, we clear the `game.running` flag and the `multiplayer.animationInterval` interval. We then show a message box with the reason for ending the game and finally call the `multiplayer.closeAndExit()` method once the OK button on the message box is clicked.

Next, we will define a new `endGame()` method in `server.js`, as shown in Listing 12-10.

Listing 12-10. The Server endGame() Method (server.js)

```
function endGame(room,reason){
    clearInterval(room.interval);
    room.status = "empty";
    sendRoomWebSocketMessage(room,{type:"end_game",reason:reason})
    for (var i = room.players.length - 1; i >= 0; i--){
        leaveRoom(room.players[i],room.roomId);
    };
    sendRoomListToEveryone();
}
```

We start by clearing the interval for the game loop. We then send the `end_game` message to all the players in the room with the reason provided as a parameter. We then set the room to `empty` and remove all the players from the room using the `leaveRoom()` method. Finally, we send the updated room list to all connected players.

Next, we will modify the message event handler on the server to handle messages of type `lose_game`, as shown in Listing 12-11.

Listing 12-11. Handing Messages of Type lose_game (server.js)

```
// On Message event handler for a connection
connection.on('message', function(message) {
    if (message.type === 'utf8') {
        var clientMessage = JSON.parse(message.utf8Data);
        switch (clientMessage.type){
            case "join_room":
                var room = joinRoom(player,clientMessage.roomId);
                sendRoomListToEveryone();
                if(room.players.length == 2){
                    initGame(room);
                }
                break;
            case "leave_room":
                leaveRoom(player,clientMessage.roomId);
                sendRoomListToEveryone();
                break;
            case "initialized_level":
                player.room.playersReady++;
                if (player.room.playersReady==2){
                    startGame(player.room);
                }
                break;
            case "latency_pong":
                finishMeasuringLatency(player,clientMessage);
```

```
                // Measure latency at least thrice
                if(player.latencyTrips.length<3){
                    measureLatency(player);
                }
                break;
            case "command":
                if (player.room && player.room.status=="running"){
                    if(clientMessage.uids){
                        player.room.commands.push({uids:clientMessage.uids,
details:clientMessage.details});
                    }
                    player.room.lastTickConfirmed[player.color] = clientMessage.currentTick +
player.tickLag;
                }
                break;
            case "lose_game":
                endGame(player.room, "The "+ player.color +" team has been defeated.");
                break;
        }
    }
});
```

When we receive a `lose_game` message from one of the players, we call the `endGame()` method with the reason for ending the game.

Finally, we will modify the `multiplayer` object's `handleWebSocketMessage()` method to receive messages of type `end_game`, as shown in Listing 12-12.

Listing 12-12. Receiving Messages of Type end_game (multiplayer.js)

```
handleWebSocketMessage:function(message){
    var messageObject = JSON.parse(message.data);
    switch (messageObject.type){
        case "room_list":
            multiplayer.updateRoomStatus(messageObject.status);
            break;
        case "joined_room":
            multiplayer.roomId = messageObject.roomId;
            multiplayer.color = messageObject.color;
            break;
        case "init_level":
            multiplayer.initMultiplayerLevel(messageObject);
            break;
        case "start_game":
            multiplayer.startGame();
            break;
        case "latency_ping":
            multiplayer.sendWebSocketMessage({type:"latency_pong"});
            break;
        case "game_tick":
            multiplayer.lastReceivedTick = messageObject.tick;
            multiplayer.commands[messageObject.tick] = messageObject.commands;
            break;
```

```
        case "end_game":
            multiplayer.endGame(messageObject.reason);
            break;
    }
},
```

When the client receives an `end_game` message, we call `multiplayer.endGame()` with the reason provided in the message.

If you start the server and run the game, you should see a message box when one of the players destroys all of the other players units and buildings, as shown in Figure 12-3.

Figure 12-3. *Game ends when one player defeats the other*

If you click the Okay button, you should be taken back to the main game menu. You will notice that when a game ends, the lobby automatically shows the room as empty so that the next set of players can join the room.

We will also end the game when a player closes the browser or is disconnected from the server.

Ending the Game When a Player Is Disconnected

Whenever a player disconnects from the server while playing a game, it will trigger a websocket `close` event on the server. We will handle this disconnect by modifying the close event handler on the server, as shown in Listing 12-13.

Listing 12-13. Handling Player Disconnects (server.js)

```
connection.on('close', function(reasonCode, description) {
    console.log('Connection from ' + request.remoteAddress + ' disconnected.');
```

```
    for (var i = players.length - 1; i >= 0; i--){
        if (players[i]==player){
            players.splice(i,1);
        }
    };

    // If the player is in a room, remove him from room and notify everyone
    if(player.room){
        var status = player.room.status;
        var roomId = player.room.roomId;
        // If the game was running, end the game as well
        if(status=="running"){
            endGame(player.room, "The "+ player.color +" player has disconnected.");
        } else {
            leaveRoom(player,roomId);
        }
        sendRoomListToEveryone();
    }
});
```

If the player is in a room, we remove the player from the room and send the updated room list to everyone. If the game was running, we also call the `endgame()` method with the reason that the player has disconnected.

If you start the server and begin a multiplayer game, you should see a disconnect message when either of the players gets disconnected, as shown in Figure 12-4.

Figure 12-4. *Message shown when a player gets disconnected*

Clicking the Okay button will take you back to the main menu screen. Again, the lobby automatically shows the room as empty so that the next set of players can join the room.

The last thing we will handle is ending the game if a connection error occurs and the connection is lost.

Ending the Game When a Connection Is Lost

Whenever the client gets disconnected from the server or an error occurs, it will trigger either an `error` or a `close` event on the client. We will handle this by implementing these event handlers within the `start()` method of the `multiplayer` object, as shown in Listing 12-14.

Listing 12-14. Handling Connection Errors (multiplayer.js)

```
start:function(){
    game.type = "multiplayer";
    var WebSocketObject = window.WebSocket || window.MozWebSocket;
    if (!WebSocketObject){
        game.showMessageBox("Your browser does not support WebSocket. Multiplayer will not work.");
        return;
    }
    this.websocket = new WebSocketObject(this.websocket_url);
    this.websocket.onmessage = multiplayer.handleWebSocketMessage;
    // Display multiplayer lobby screen after connecting
    this.websocket.onopen = function(){
        // Hide the starting menu layer
        $('.gamelayer').hide();
        $('#multiplayerlobbyscreen').show();
    }

    this.websocket.onclose = function(){
        multiplayer.endGame("Error connecting to server.");
    }

    this.websocket.onerror = function(){
        multiplayer.endGame("Error connecting to server.");
    }
},
```

For both the events, we call the `endGame()` method with an error message. If you run the game now and shut down the server to re-create a server disconnect, you should see an error message, as shown in Figure 12-5.

Figure 12-5. *Message shown in case of a connection error*

If there is a problem with the connection while the player is either in the lobby or playing a game, the browser will now display this error message and then return to the main game screen.

A more robust implementation would include trying to reconnect to the server within a timeout period and then resuming the game. We can achieve this by passing a reconnect message with a unique player ID to the server and handling the message appropriately on the server side. However, we will stick with this simpler implementation for our game.

Before we wrap up the multiplayer portion of our game, we will implement one last feature in our game: player chat.

Implementing Player Chat

We will start by defining an input box for chat messages inside the `gameinterfacescreen` layer in `index.html`, as shown in Listing 12-15.

Listing 12-15. Adding Input Box for Chat Message (index.html)

```
<div id="gameinterfacescreen" class="gamelayer">
    <div id="gamemessages"></div>
    <div id="callerpicture"></div>
    <div id="cash"></div>
    <div id="sidebarbuttons">
        <input type="button" id="starportbutton" title = "Starport">
        <input type="button" id="turretbutton" title = "Turret">
        <input type="button" id="placeholder1" disabled>
```

```
        <input type="button" id="scouttankbutton" title = "Scout Tank">
        <input type="button" id="heavytankbutton" title = "Heavy Tank">
        <input type="button" id="harvesterbutton" title = "Harvester">

        <input type="button" id="chopperbutton" title = "Copter">
        <input type="button" id="wraithbutton" title = "Wraith">
        <input type="button" id="placeholder2" disabled>

    </div>
    <canvas id="gamebackgroundcanvas" height="400" width="480"></canvas>
    <canvas id="gameforegroundcanvas" height="400" width="480"></canvas>
    <input type="text" id="chatmessage"></input>
</div>
```

Next we will add some extra styles for the chat message input to `styles.css`, as shown in Listing 12-16.

Listing 12-16. Styles for Chat Message Input Box (styles.css)

```
#chatmessage{
    position:absolute;
    top:460px;
    width:479px;
    background:rgba(0,255,0,0.1);
    color:green;
    border:1px solid green;
    display:none;
}
#chatmessage:focus {
    outline:none;
}
```

Next we will add an event handler for keydown events inside `multiplayer.js`, as shown in Listing 12-17.

Listing 12-17. Handing Keydown Events to Handle the Chat Message Input (multiplayer.js)

```
$(window).keydown(function(e){
    // Chatting only allowed in multiplayer when game is running
    if(game.type != "multiplayer" || !game.running){
        return;
    }

    var keyPressed = e.which;
    if (e.which == 13){ // Enter key pressed
        var isVisible = $('#chatmessage').is(':visible');
        if (isVisible){
            // if chat box is visible, pressing enter sends the message and hides the chat box
            if ($('#chatmessage').val()!= ''){

                multiplayer.sendWebSocketMessage({type:"chat",message:$('#chatmessage').val()});
                $('#chatmessage').val('');
            }
            $('#chatmessage').hide();
        } else {
```

```
            // if chat box is not visible, pressing enter shows the chat box
            $('#chatmessage').show();
            $('#chatmessage').focus();
        }
        e.preventDefault();
    } else if (e.which==27){ // Escape key pressed
        // Pressing escape hides the chat box
        $('#chatmessage').hide();
        $('#chatmessage').val('');
        e.preventDefault();
    }
});
```

Whenever a key is pressed, we first confirm that the game is a multiplayer game and it is running and exit if it is not. If the key pressed is the Enter key (key code 13), we first check whether the `chatmessage` input box is visible. If it is visible, we send the contents of the message box to the server inside a message of type `chat`. We then clear the contents of the input box and hide it. If the input box is not already visible, we display the chat input box and set the focus to it. If the key pressed is Escape (key code 27), we clear the contents of the input box and hide it. Next, we will modify the message event handler on the server to handle messages of type `chat`, as shown in Listing 12-18.

Listing 12-18. Handing Messages of Type chat (server.js)

```
// On Message event handler for a connection
connection.on('message', function(message) {
    if (message.type === 'utf8') {
        var clientMessage = JSON.parse(message.utf8Data);
        switch (clientMessage.type){
            case "join_room":
                var room = joinRoom(player,clientMessage.roomId);
                sendRoomListToEveryone();
                if(room.players.length == 2){
                    initGame(room);
                }
                break;
            case "leave_room":
                leaveRoom(player,clientMessage.roomId);
                sendRoomListToEveryone();
                break;
            case "initialized_level":
                player.room.playersReady++;
                if (player.room.playersReady==2){
                    startGame(player.room);
                }
                break;
            case "latency_pong":
                finishMeasuringLatency(player,clientMessage);
                // Measure latency at least thrice
                if(player.latencyTrips.length<3){
                    measureLatency(player);
                }
                break;
```

```
            case "command":
                if (player.room && player.room.status=="running"){
                    if(clientMessage.uids){
                        player.room.commands.push({uids:clientMessage.uids,
details:clientMessage.details});
                    }
                    player.room.lastTickConfirmed[player.color] = clientMessage.currentTick +
player.tickLag;
                }
                break;
            case "lose_game":
                endGame(player.room, "The "+ player.color +" team has been defeated.");
                break;
            case "chat":
                if (player.room && player.room.status=="running"){
                    var cleanedMessage = clientMessage.message.replace(/[<>]/g,"");
                    sendRoomWebSocketMessage(player.room,{type:"chat", from:player.color,
message:cleanedMessage});
                }
                break;
        }
    }
});
```

When we receive a message of type `chat` from a player, we send back a message of type `chat` to all the players in the room, with a `from` property set to the player's color and a `message` property set to the message we just received.

Ideally, you should validate the chat message so that a player cannot send malicious HTML and script tags inside chat messages. For now, we use a simple regular expression to strip out any HTML tags from the message before sending it back.

Finally, we will modify the `multiplayer` object's `handleWebSocketMessage()` method to receive messages of type `chat`, as shown in Listing 12-19.

Listing 12-19. Receiving Messages of Type chat (multiplayer.js)

```
handleWebSocketMessage:function(message){
    var messageObject = JSON.parse(message.data);
    switch (messageObject.type){
        case "room_list":
            multiplayer.updateRoomStatus(messageObject.status);
            break;
        case "joined_room":
            multiplayer.roomId = messageObject.roomId;
            multiplayer.color = messageObject.color;
            break;
        case "init_level":
            multiplayer.initMultiplayerLevel(messageObject);
            break;
        case "start_game":
            multiplayer.startGame();
            break;
```

```
        case "latency_ping":
            multiplayer.sendWebSocketMessage({type:"latency_pong"});
            break;
        case "game_tick":
            multiplayer.lastReceivedTick = messageObject.tick;
            multiplayer.commands[messageObject.tick] = messageObject.commands;
            break;
        case "end_game":
            multiplayer.endGame(messageObject.reason);
            break;
        case "chat":
            game.showMessage(messageObject.from,messageObject.message);
            break;
    }
},
```

If you start the server and play a multiplayer game now, you should be able to send chat messages from one player to the other, as shown in Figure 12-6.

Figure 12-6. *Chat between players during multiplayer*

We now have a working player chat for multiplayer. With this last change, we can consider our multiplayer game wrapped up.

Summary

We have come a long way over the course of this book. We started by looking at the basic elements of HTML5 needed to build games, such as drawing and animating on the canvas, playing audio, and using sprite sheets.

We then used these basics to build a Box2D physics engine–based game called *Froot Wars*. In the process, we looked at creating splash screens, asset loaders, and customizable levels. We then examined the building blocks of the Box2D engine and integrated Box2D with the game to create realistic-looking physics. We then added sound effects and background music to create a very polished game.

After that, we built a complete real-time strategy game called *Lost Colony*. Building upon ideas from the previous chapters, we first created a single-player game world with large pannable levels and different types of entities. We added intelligent movement using pathfinding and steering, combat using states and triggers, and even a game economy. We then saw how this framework could be used to tell a compelling story over several levels of a single-player campaign.

Finally, over the last two chapters, we used Node.js and WebSockets to add multiplayer support to our game. We started by looking at the basics of WebSocket communication and using it to create a multiplayer game lobby.

We then implemented a framework for multiplayer gameplay using the lock-step networking model, which also compensated for network latency while maintaining game synchronization. We handled connection errors as well as game completion using triggered events. Finally, we built a chat system to send messages between the players.

If you have been following along, you should now have the knowledge, resources, and confidence to build your own amazing games in HTML5.

My goal in writing this book was to demystify the process of building complex games in HTML5 and provide you with everything that you would need to build such games on your own.

If you have questions or feedback, you can reach me using the dedicated page for this book on my website at `www.adityaravishankar.com/pro-html5-games/`. I would love to hear about how you used this book as a starting point for your own projects.

I wish you all the best in your game-programming journey.