**UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI**



GROUP PROJECT

Multiplayer Draw and Guess Game

Submitted by Group Light up

|  |  |
| --- | --- |
| Pham Gia Phuc | BI10 - 138 |
| Do Hoang Quan | BI10 - 147 |
| Dang Gia Linh | BI10 - 101 |
| Pham Duc Thang  Tran Dac Minh | BI10 - 159  BI10 - 113 |
|  |  |

Under the guidance of: Dr. TRAN Giang Son

Hanoi, January 2022

Table of contents

Contents

**No table of contents entries found.**

# Acknowledgements.

We would like to express our special thanks of gratitude to our teacher Dr. Tran Giang Son who gave us the opportunity to do this wonderful project on the topic Multiplayer Draw and Guess game, which also helped us in enriching experience.

# Introduction.

**1.1 Context.**

With our learning experience, we will build a multiplayer game using WebSockets. WebSockets will enable us to create event-based server-client architecture. The messages are passed between all connected browsers instantly. We will combine the Canvas drawing, JSON data packing, and several techniques learned in the university to build a draw and guess game.

**1.2 Objective.**

Our objective for this project is to develop and preliminary design a game with a purpose that with potential to build a dataset for drawing based image retrieval field. To achieve this, we:

* Develop a user-friendly interface to allow users to easily color sketches.
* Propose an entertaining game, which employs a drawing based image retrieval paradigm in the gameplay.

**1.3 Thesis Structure.**

The rest of the thesis will be structured as following sections:

* Section 1: Introduction about project’s objective.
* Section 2: Methodology.
* Section 3: Summarize the results and talk about future improvement.

# Materials and methods.

**2.1. Functional requirement.**

* User can join a room to play the game
* User can choose a word then sribble to describe it
* User can guess what another user is drawing

**2.2. System analysis and design.**

**2.2.1 Use case Diagram.**

(use case diagram)

**2.2.2 Class Diagram.**

A class diagram models the static structure of a system. It shows relationships between classes, objects, attributes, and operations.

(class diagram)

**2.2.3 System Architecture.**

**2.3. Implementation.**

**2.3.1 Installing Node.js Websocket server.**

We build server logic on top of this environment. The WebSocket server does not necessarily run on Node.js. There are different server-side implementations of the WebSockets protocol. We chose Node.js because it uses JavaScript, and we are familiar with it.

**2.3.2 Creating a WebSocket server.**

We create a simple server logic that initialized the WebSockets library and listened to the connection event. In Node.JS, different functions are packed into modules. When we need a functionality in,a specific module, we use require to load it. We load the WebSockets module and then initialize the server using the following code in the server logic:

<code>

Since the ws module is managed by npm, it's installed inside a folder called node\_modules. When we require a library with only the name, the Node.js runtime looks for that module in the node\_modules folder. We used 8000 as the server's port number, with which a client connects to this server. We may choose a different port number, but we have to ensure that the chosen port number is not overlapped by other common server services.

*server.on('connection', function(socket) {*

*console.log("A connection established");*

*});*

The connection event comes with a socket argument. We will need to store this socketlater because we use this object to interact with the connecting client.

**2.3.3 Creating a client that connects to WebSocket server and getting the total connections count.**

We build a client that established a WebSockets connection to the server that we built in the last section. The client will print any messages that are received from the server to the console panel in the Inspector of Developer Tools.

* *Establishing a WebSocket connection*

*var socket = new WebSocket(url);*

The url argument is a string with the WebSockets URL. In our example, we run our server locally. Therefore, the URL we used is ws://127.0.0.1:8000, where 8000 represents the port number of the server to which we are connecting. It is 8000 because the server was listening to port 8000 when we built the server-side logic.

* *About WebSocket server client events*, the following table lists the events we will use to deal with WebSockets:

|  |  |
| --- | --- |
| **Event name** | **Description** |
| Onopen | This is fired when a connection to the server is established |
| Onmessage | This is fired when any message from the server is received |
| Onclose | This is fired when the server closes the connection |
| Onerror | This is fired when there is any error in the connection |

* *Sending message to all connected browsers*

Once the server gets a new connection event, we send the updated count of the connection to all clients. We just need to call the sendAll function in the server instance with a string argument as the message. The following code snippet sends a server message to all connected browsers:

*var message = "a message from server";*

*server.sendAll(message);*

We defined two classes, User and Room, in a game.js file, which we use to manage all the connected sockets.

* *Defining class and instant instance methods*

In JavaScript, object-oriented programming is done by using functions and prototypes. When we create a room instance by calling new Room(), the browser clones all properties and methods in Room.prototype to the instance.

* *Handling a newly connected user*

For each connected user, we need to interact with them via an events handler. We add the user object into an array for easy management. We need to handle the onclose event when a user disconnects. To do this, we remove that user from the array.

* *Exporting modules*

After defining our classes in the game.js file, we exported them. By exporting them to the module, we can import them in the other file by using the require method, as follows:

*var User = require('./game').User;*

*var Room = require('./game').Room;*

* *Sending messages to the client*

WebSockets have the ability to send messages from the server to a user. Normally, the client requests the server and then the server responds. In a socket server, all users are connected, so messages can be triggered and sent in both directions.

*Room.prototype.send = function(message) {*

*for (var i=0, len=this.users.length; i<len; i++) {*

*this.users[i].socket.send(message);*

*}*

*};*

Then listen the message on the client:

*// on message event*

*websocketGame.socket.onmessage = function(e) {*

*console.log(e.data);*

*};*

**2.3.4 Building a chatting application with WebSockets**

We want to build a chat room where users can type a message in their respective browsers and send the message to all the connected users instantly.

* *Sending a message to the server*

We add an input text field for the users to type some text there and send it out, the user input a message and then the text is sent as a message to the WebSocket server. The server will then forward the message to all connected browsers. Once a browser receives the message, it displays it in the chat area. In this case, the users are connected to the instant chat room once they load the web page. The server will then print the received message in the terminal.

(add code)

* *Sending a message from the client to the server*

In order to send a message from the client to the server, we call the following send method in the WebSocket instance:

*websocketGame.socket.send(message);*

In the following code snippet from our example, we get the message from the input text field and send it to the server:

*var message = $("#chat-input").val();*

*websocketGame.socket.send(message);*

* *Receiving a message on the server side*

On the server side, we need to handle the message we just sent from the client. We have an event named message in the connection instance in the WebSocket node.js library. We can listen to the connection message event to receive a message from each client connection.

* *Sending every received message on the server side to create a chat room*

The server could receive messages sent from browsers. However, the server does nothing except print the received messages in the terminal. We learned how a server sends the connection count to all the connected clients. We also learned how the client sends a message to the server. Therefore, we add some logic to the server to send the messages out by combining these two techniques to let the server send the received messages to all the connected users.

(add code)

**2.3.5 Marking a shared drawing whiteboard with Canvas and WebSockets**

We want a shared sketchpad. Anyone can draw something on the sketchpad and all others can view it. We learned how messages are communicated between clients and servers. We will go further and send drawing data.

* *Building a local drawing sketchpad*

Before we work with data sending and server handling, we focus on making a drawing whiteboard. We use the Canvas to build a local drawing sketchpad. We created a local drawing pad. This is like a whiteboard where the player can draw in the Canvas by dragging the mouse. However, the drawing data is not sent to the server yet all drawings are only displayed locally.

(add code)

* *Drawing in the Canvas*

When we draw something on the computer, it often means that we click on the Canvas and drag the mouse (or pen). The line is drawn until the mouse button is up. Then, the user clicks on another place and drags again to draw lines. In our example, we have a Boolean flag named isDrawing to indicate whether the user is drawing. The isDrawing flag is false by default. When the mouse button is at a point, we turn the flag to true. When the mouse is moving, we draw a line between the moved point and the last point when the mouse button was. Then, we set the isDrawing flag to false when the mouse button is up. This is how the drawing logic works. We also add five buttons with 5 different colors. The player can choose the color when drawing.

* *Sending the drawing to all the connected browsers*

We build a multiuser sketchpad. This is similar to the sketchpad we did at the beginning. We build a chat room by sending a complex data object as a message.

(add code)

* *Defining a data object to communicate between the client and the server*

In order to communicate correctly between the server and clients when there is a lot of data packed into one message, we have to define a data object that both the client and server understand.

There are several properties in the data object. The following table lists the properties and why we need them:

|  |  |
| --- | --- |
| Property name | Why we need it |
| dataType | This is an important property that helps us to understand the entire data. The data is either a chat message or drawing line segment data |
| sender | If the data is a chat message, the client needs to know who sent the message. |
| message | If the data is a chat message, the client needs to know who sent the message. |
| startX | When the data type is a drawing line segment, we include the xy coordinates of the starting point of the line. |
| startY |
| endX | When the data type is a drawing line segment, we include the x/y coordinates of the ending point of the line. |
| endY |

* *Packing the drawing lines data into JSON for sending*

(add code)

* *Recreating the drawing lines after receiving them from other clients*

(add code)

**2.3.6 Building a multiplayer draw-and-guess game.**

We built an instant chat room earlier. We built a multiuser sketchpad. Now we combine these two techniques and build a draw-and-guess game A draw-and-guess game is a game in which one player is given a word to draw. All other players do not know the word and guess the word according to the drawing. The one who draws and who correctly guesses the word earn points.

We need a few more constants to determine different states during the game play.

(add code constants hoac picture k co thi thoi)

When the client receives a message from the server, it parses it and checks whether it is a chat message or a line drawing. We have another type of message named GAME\_LOGIC for handling the game logic. The game logic message contains different data for different game states.

(add code)

There is one last step in the client-side logic. We want to restart the game by sending a restart signal to the server. At the same time, we clear the drawing and chat history.

(add code restart)

The server side was just in charge of sending any incoming message to all connected browsers. This is not enough for a multiplayer game. The server will act as the game master that controls the game flow and determination of the winning condition. We extend the Room class with GameRoom that can handle the game flow.

This is the constructor function of a new class called GameRoom, which initializes game logic:

(add gameroom function)

The following function creates a new game inside the room by picking a player as a drawer and the others as guessers, then, it randomly picks a word for the drawer to draw:

(add function start game hay gi day tuong tu)

After saving all the files and relaunching the server, we launch the index.html file in two browser instances. One browser will get a message from the server informing the player to draw something. The other browser, on the other hand, will inform the player to guess what the other player is drawing within one minute.

The player who is told to draw something can draw in the Canvas. The drawings are sent to all the other connected players. The players who are told to guess cannot draw anything in the Canvas. Instead, players type what they guess in the text field and send it to the server. If the guess is correct, then the game ends. Otherwise, the game continues until the one-minute countdown finishes.

* *Controlling the game flow*

There are several states in the game flow. Before the game starts, the connected players are waiting for the game to start. Once there are enough connections for the multiplayer game, the server sends a game logic message to all the players to inform them of the start of the game.

When the game is over, the server sends a game over state to all the players. Then, the game finishes and the game logic halts until any player clicks on the restart button. Once the restart button is clicked, the client sends a game restart state to the server instructing the server to prepare a new game. Then, the game starts again. We declare the four game states as the following constants in both the client and server so that they understand them:

(add may cai constant for game logic state)

The following code on the server side holds an index to indicate which player's turn is:

(add dong` player turn)

The data which is sent to the player (whose turn it is) is different from the data that is sent to other players. The other players receive the following data with only a game start signal:

(add code cua guesser)

On the other hand, the player (whose turn is to draw) receives the following data with the word information:

(add code cua drawer)

# Conclusion and future development

* 1. **Conclusion**
  2. **Development**

# References