Pictures with Friends

Draw and Guess game with more interesting implementation of Socket.io and Node.js
(because we love them)

Authors:

- Christian Chiang
- Faizul Jasmi

Table of Contents

Introduction

The main feature of the game that is needed to be implemented is the canvas for player to draw. The mission is to implement a real-time canvas that more than one user can see at the same time. This is the most important and fundamental mechanic of the game. So, how did we do it? We use node.js server which when a client draws a line on the canvas in the browser, it sends them to other client who is connected so that they can see the same thing.

To do this, we need:

1) **Node.js** - Installed using the npm package manager.
2) **Express.js** - Act as a webserver so that we can open the web app in the browser. Even though Node.js comes with a built-in http package, we need more than that to serve files. It comes with a lot of features, which most of them we don't really need but it's really easy to use a convenient.
3) **Socket.io** - It allows us to use websockets in a very convenient manner. Has various fallbacks if the websockets aren't supported.

There is also this concept about sockets separation that Socket.io allows which is called Rooms. Rooms are ways to separate various connections into different places where their actions will only be displayed to the sockets in that room. What this means for our project is that we are able to have multiple game rooms where players can play with their friend without getting interrupted by other players. An instance of this, it that the word generated in room #1 will be different from room #2 or room #3, rooms give an independability to every socket. Without the need of redirecting the user to another page and have much duplicate code.

How We Made It

Package.json, HTML & CSS:

Just like every Node.js based project, we need the package.json file to install all of the dependencies that we need in this project. After we have installed the list of dependencies, it is time to create the HTML and CSS files. This serves as the front-end of the web app. For a more systematic directory, we put the files except for the server and index.html file, in a different root file called public.

To implement something that the user can draw on on the page, we need an element called <canvas>. It is also important for this canvas to be placed on a proper place for the mouse pointer to draw perfectly on the screen. To do that, we implemented the CSS file.

The Server:

In index.js, we import the modules that we have installed to our dependencies earlier. Other than that, we also imported the built-in http module to create a web server.

```
var express = require('express'),
app = express(),
http = require('http').Server(app),
io = require('socket.io')(http);
```

Then, the webserver is started and socket.io is initialized. Next, we connect the server files with the HTML and CSS file using express by asking it to look in the public directory.

```
http.listen(4000, function() {
   console.log('listening on *:4000');
});
```

At this point, the server is able to be run by ‘node index.js’ in the terminal.

Socket Handling Receiving and Sending:

First, we register a handler for new incoming connections. When a new client is connected, the function is called and the new socket for the client is passed as the argument.

```
io.on('connection', function (socket) {
```

Inside this handler, we send the lines drawn to the new client. Then, we add a handler for our own message-type draw_line to the new client. Each time a line is received, it is sent to all connected clients in a specific room so the canvas can be updated only in that room.

```
socket.on('draw_line', function (data) {
  io.to(socket.room).emit('draw_line', { line: data.line });
});
```

The Client:

On the client side, we first need to select the room where we want to play, Selecting a room allows our web app to isolate your responses to only to the people in that room. Once in the room we are able to take on the Drawer status and be able to draw on the canvas while the other players are guessing. All of these, the pseudo random generate word, the players response, the lines drawn on the canvas are sent to our server so other players can see the response in real time.

Challenge: Specifically for the drawing part each client may have different screen resolutions. If we send an definite or absolute mouse position of client A to client B who has different screen size, client B would have a problem viewing the drawing of Client A. To solve that, we normalize the mouse coordinates to range from 0 to 1. We do this by dividing the mouse coordinates by the width and height of the canvas.

The client side then receive the coordinates and multiply it by their own browser width and height and get the absolute position for their respective screen.

The Base:

To make sure that the browser is ready, we use the DOMContentLoaded event and also we wait for HTML files to load, some of the time it's not necessary since by the time a room is picked the page will load accordingly. The client script is run when the event is initiated. To keep track of the mouse, we declare an object called mouse with properties of click, move, pos, and pos_prev.

Next, we implement the canvas element and create a 2d drawing context, we set the width and height of it and then we tell Socket.io to connect to the server.

```
var canvas  = document.getElementById('drawing');
var context = canvas.getContext('2d');
var width   = 500;
var height  = 500;
```

Mouse Events:

For the browser, we set a few variables for the mouse so that it could do certain events when something happened with the mouse. The variables are `mousedown`, `mouseup` and `mousemove`. mouse.click will be true when the mouse is clicked.

Everytime the mouse is moved, we keep track the x and y position and assign it to `mouse.pos`, respectively to its x and y. After we get the value, then we can do the position normalization that we mentioned before.

Server Listening for Messages:

Since we store the lines drawn on the server side, we want to in a way transmit it to the client side so that it can be shown on the canvas. So, we use the draw_line message that we have in the server file. For this use and size of canvas, where it would not have many and complex combinations of lines, it is okay to draw every single line individually. `beginPath` begins a new path. `moveTo` takes the first point to the second

point which is `lineTo,` which then draws the line. To actually draw the line on the canvas, we call `context.stroke()`.

```
socket.on('draw_line', function (data) {
    var line = data.line;
    context.beginPath();
    context.lineWidth = 2;
    context.moveTo(line[0].x * width, line[0].y * height);
    context.lineTo(line[1].x * width, line[1].y * height);
    context.stroke();
  });
```

The Most Important Part, The Loop:

In this loop, we check every 25ms whether the mouse is clicked, has moved and has a previous position (since we need two points for a line). We send a `draw_line` message to the server and reset `mouse.move` to false only when all three conditions are true. Then, we do the trivial process of setting the current mouse position to the previous one.

```
function mainLoop() {
    if (mouse.click && mouse.move && mouse.pos_prev) {
      socket.emit('draw_line', { line: [ mouse.pos, mouse.pos_prev ] });
      mouse.move = false;
    }
    mouse.pos_prev = {x: mouse.pos.x, y: mouse.pos.y};
    setTimeout(mainLoop, 25);
  }
  mainLoop();
});
```

Rooms

As mentioned briefly in the beginning, having a Rooms in our web app makes the experience a bit more pleasant. It isolates sockets to the desired room so other cannot interfere with other rooms. Socket.io handles this with extreme ease, to join a room you just use:

```
socket.join(room);
```

And to leave you just call .leave(room) on the same socket. The problem arises when you join another room without previously leaving yours, this will cause multiple bugs and many interference in both joined rooms, meaning that before joining a room you should first leave the current room.

Generally using Socket.io to send information from the server to the client you just call `socket.emit('name of event', function(data){ });` , the problem with doing this, is that everything done in the event will be down in every socket that is connected. To solve this and just perform events in a specific room we just do `io.to(room).emit('name of event', function(data){ });` , been the variable room the desired room to send the information. An important note here is that everything of room isolation aside from the room selection is done in the server, this means that if you are emitting and event from the client side, you should not worry about using io.to(room) .

Bloom’s Taxonomy

Creation

When we first started the project, we started on doing the canvas, we started doing it strictly with only JavaScript and having it on the client side. We noticed that the mouse position was not precisely on the drawing pen of the canvas so we calibrated using fixed values in a standard canvas. We added features of colors and erasers which worked perfectly on the client side. Finally for the canvas we added sounds effects for the timer countdown.

After the canvas was working perfectly as a pure javascript application we decided to tweak itnto using Node.js and Socket.io for that real time response and so multiple connections could see the what was happening. This was mostly difficult due to the nature of our application been just tested in the client, as a result we had to remove some features like changing the color of the pen as the color would only be seen in the drawers page.

Learning from that mistake we decided to perform the word generation side of the project having Socket.io already in mind. The way we did this is we thought of functions as a socket.emit method were instead of just performing everything on the client side we will alert the server that a certain event will be going on so it would display the different changes to everyone in the server. Therefore when generating a word or inputting the answer we will send the data to the server and then re-emit it to the other sockets.

After merging both apps together we implemented the rooms, so we could separate different sockets into different spaces. It was challenging at the beginning as the documentation about it it’s really vague but, it was actually pretty simple as the only change to our code was the need to join for a room and instead of emitting the event to everyone you would just isolate the event to the certain room for that socket.

Analysis

At first we thought this is a familiar project since we worked with Socket.io on portfolio two and we wanted to keep expanding our knowledge in this. But it turned out it is a whole lot more of a new thing. In the previous portfolio we used namespaces and cookies to keep track of the sessions that were connected in our server, this was easier to do as we just generated a link where the other player could just join. This time having rooms presented a bigger challenge as information kept getting leaked to other rooms causing a bad user experience.

Although Node.js and Socket.io facilitate the creation of server and client communication, designing and making a responsive web app is not very easy, specially if you are new to the subject. A work around we did was just hiding and showing different divisions of the HTML source code to control what the user could see and use. We though of using Angular JS for the project to clean up the front end, but as most of our code was handled by Socket.io in the server side we saw it was a bit of an overkill.

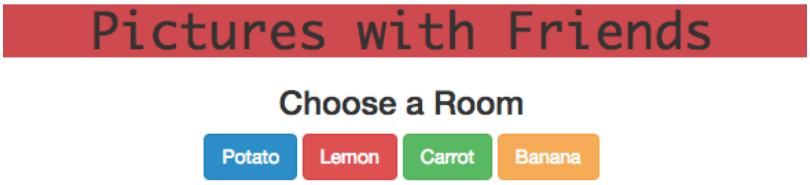
One small issue we encountered among the creation of the canvas was keeping track of the resolution of the screen and the position of the mouse with respect of it. This caused many problems as the mouse itself it's not centered in the canvas, we created a fixed value that will work perfect if the screen it's maximized in a 13.3" screen, anything different than that won't be as accurate.

Evaluation

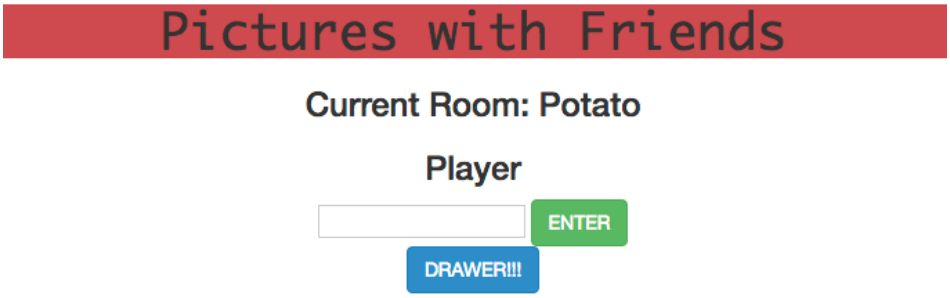
We began this portfolio as an extension for the previous one, we wanted to expand our knowledge on Node.js and Socket.io. In Picture with Friends we kept the same way of thinking of having a single page app which will not need any refreshing and would be real time. Making a merge with CyChess in the future is a possibility on where PIctures with Friends might go in order to have a multiplayer social gaming website where users can spend their time in a quick match. There are a few features that are planned for a near future, this include but are not limited to: A database to store users information, a messaging chat to communicate during the game or out in the

lobby, huge popup when the user wins, cleaner navigation and such more. Although this features are not included in the current version, Pictures with friends is a complete application which can be played and have a blast.

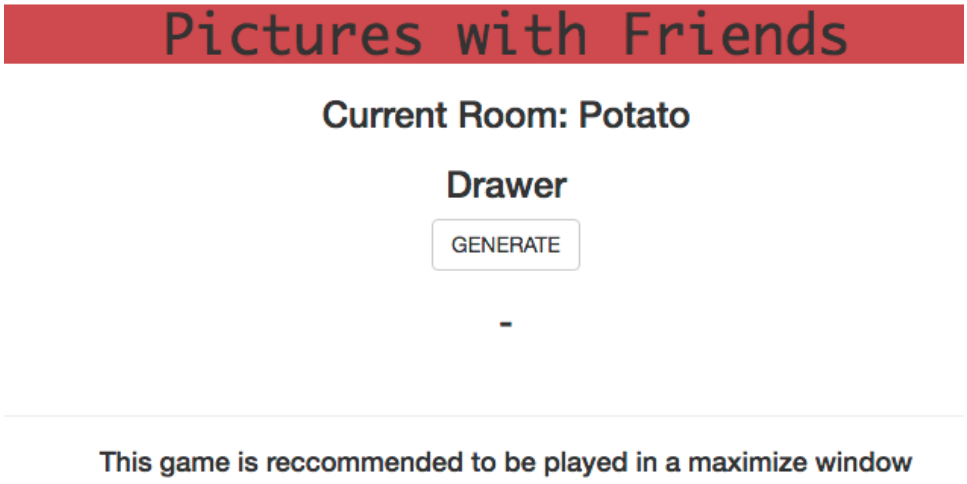
Screenshots



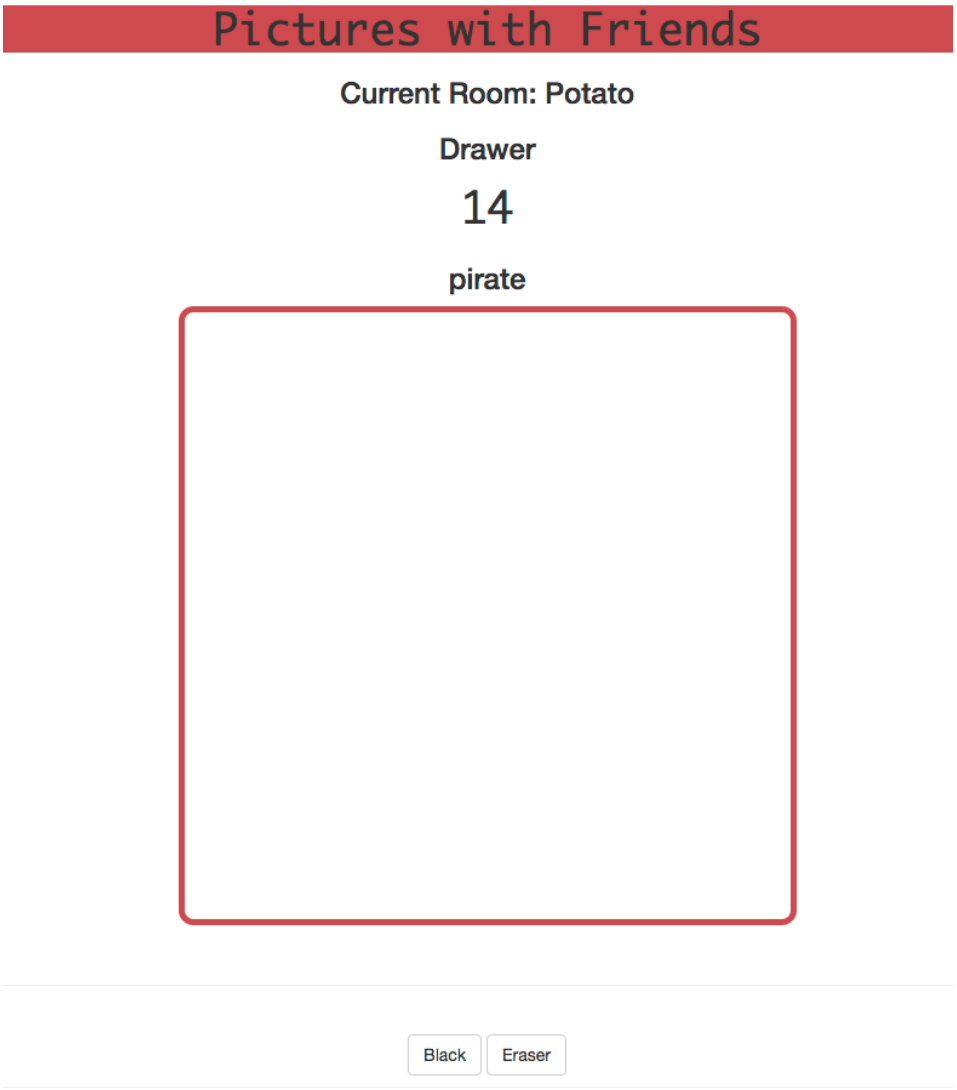
Main Page



Player view before game



Drawer view before game



On going game