DoodleChat

Briar Sullivan
Undergraduate Computer Science Capstone
Capstone Advisor: Dr. Michael J. Reale
SUNY Polytechnic Institute
Fall 2025

Introduction

In an ever-connected world where a large portion of software users are reliant on mobile devices for their daily computing needs and therefore do not directly install software onto their operating systems as historically expected, having knowledge of both front end and back end development of progressive web applications is vital to reaching and meeting the needs of these end users. In addition to this, knowing how to handle real-time connections is incredibly important to consider when developing any type of software that aims to create a collaborative platform for work or play. With "DoodleChat," I aimed to gain experience with the design, development, and knowledge of common pitfalls and design patterns of applications that utilize a backend server and communications between users and the server through websockets. Choosing to work with HTML5 Canvas elements and image processing was an additional challenge, as figuring out how to send only the necessary data to complete the user action to all clients connected in a format and size that is within the limits of the websocket architecture was something I had not initially considered.

Related Work

DoodleChat's inspiration lies entirely with other highly accessible, web-based software I have utilized in the past as both a young teen and an adult artist. Some of the earliest online art applications I encountered were the "oekaki" boards of the early 2000s. "Oekaki" is a diminutive Japanese word for "drawing"[1] and refers to art-based messaging boards that consisted of sharing and posting images created in previously Java-based, now HTML5 in-browser drawing applets such as PaintBBS or ChickenPaint[2]. Users would draw directly in an application that was rendered in their browser with no installation needed, and upload the finished drawing to the site through the application as a post that could receive comments and be shared with others. Several of these programs also included support for viewing a stroke-by-stroke replay of the user's actions taken when creating their artwork. The format of application I was seeking to emulate has colloquially been referred to as "PaintChat"[3] since the early 2000s and is a spinoff of the aforementioned oekaki applications, but geared towards a more social experience. PaintChats were built with the capability for multiple users to connect to and draw on a singular canvas at the same time and participate in a text chat that was available on the same page. Other examples of early multi-user chat-and-draw art applications include the flash-based iScribble, which was opened in 2006 and has since been replaced by a successor called HelloPaint after the death of Flash in 2016 [4], and the JavaScript-based Magma, formally known as "Aggie.io." All of these

applications have the same underlying core concepts: a shared canvas in your browser where you and others collaborate to create drawings in real time with various tools and a chat box where you can talk to each other as you draw. They also tend to support many separate canvas instances so multiple groups of people can draw together in different canvases and labels indicating who is drawing what and where they are on the canvas, but not all of them include a social posting aspect of their predecessor oekaki boards.

Approaches to creating a real-time PaintChat application have varied greatly over the last twenty or so years, and the source code of the most robust and well-known applications available today is closed-source. Magma started off as Aggie.io, an initially open-source project created in 2016 by lone developer Radek Eichler [5] and written in Typescript, with an Angular front end and Node.js server for user synchronization [6]. After shifting to a closed-source model with their Magma re-brand and expansion, Magma's exact development stack is unclear, but the developers are still utilizing an HTML5 Canvas alongside a backend server to handle synchronization across users. Their proprietary solution that allows up to 50 artists to collaborate on a single canvas takes an “MMORPG-like” approach, wherein the client processes each action locally and the server attempts to keep synchronization of the canvas consistent for all connected users [5]. From personally examining iScribble’s HelloPaint, their frontend appears to be written with React/Next.js and messages visible in the console indicate that each action is sent to a server by the client to then be forwarded to other users connected to the board. Both Magma and HelloPaint have support for user accounts and persistent canvas sessions, wherein a user saves the session for later and the server stores the associated information and actions within a database to be retrieved when the board is accessed again in the future. Older applications such as the original PaintChat based on PaintBBS and the oekaki board software did not have user accounts or persistent canvases, but as the world trends towards cloud computing this feature tends to show up more often in the modern solutions for paint-and-chat programs.

Method

The approach I chose for this project was to utilize only as many external libraries as necessary to get the project running, as my focus was not to learn writing code for a specific front-end library, but how to handle managing synchronization of an application through websockets and a web server. For the backend, I chose Node.js as it is well-supported and Express.js makes it incredibly easy to set up a minimalist webserver. Express also has ways to route and serve dynamically rendered HTML from the

server using templates that do not require me to use a separate front-end framework such as React or Angular through its Express Layouts extension. I ultimately decided to write the clientside application in vanilla Javascript, heavily utilizing the open-source graphics processing library P5.js as it greatly extends the capabilities of the HTML5 canvas element and makes working with it much easier. The main reasons for choosing P5.js were the ease of integration with Node.js, alongside the mentioned extensions of the HTML5 canvas element. Features that drew me to P5.js was the support for offscreen graphics buffers, a built-in drawing loop that was easy to access and incorporate other functionality into, easy access and manipulation of pixel information of canvases, and other common functions utilized in graphics processing applications such as image masks. I chose Socket.io for my websocket connections as it is straightforward and a well-known library that supports event-based two-way communication between the clients and server.

Rendering and the Main Loop

Due to the single-threaded nature of Javascript and P5.js' use of a draw function as the main rendering loop in combination with data being sent to and received from the server through asynchronous events, I chose to also utilize the draw loop as the main program loop. Through trial and error, choosing to implement a command pattern design solution ensured the timing of action execution occurs without interfering with the draw loop. By pushing every event asynchronously received or locally completed to a queue of actions, I ensure that those actions are performed at the beginning of a new frame and will not overlap with the redrawing process. Timing was an issue because the functions to redraw layers rely on sorting and accessing each layers' stroke history array, which incoming strokes were attempted to be pushed into. Initially I was modifying layers with commands directly in the functions that were called when data was received by the server, but P5.js' attempts to reach 60fps and the draw function's frequency of access, sorting, and changes to layer stroke history meant data was being lost with the asynchronous arrivals of strokes from the server. Implementing the command pattern, where functions instead queued up actions for the action queue to then invoke execution of at a better time, solved this situation and the associated data loss, and made it much easier to implement per-user undoing and redoing on both a local and remote level. The following figure, Figure 1, demonstrates the program loop and the order in which performing tasks and drawing different elements of the UI works.

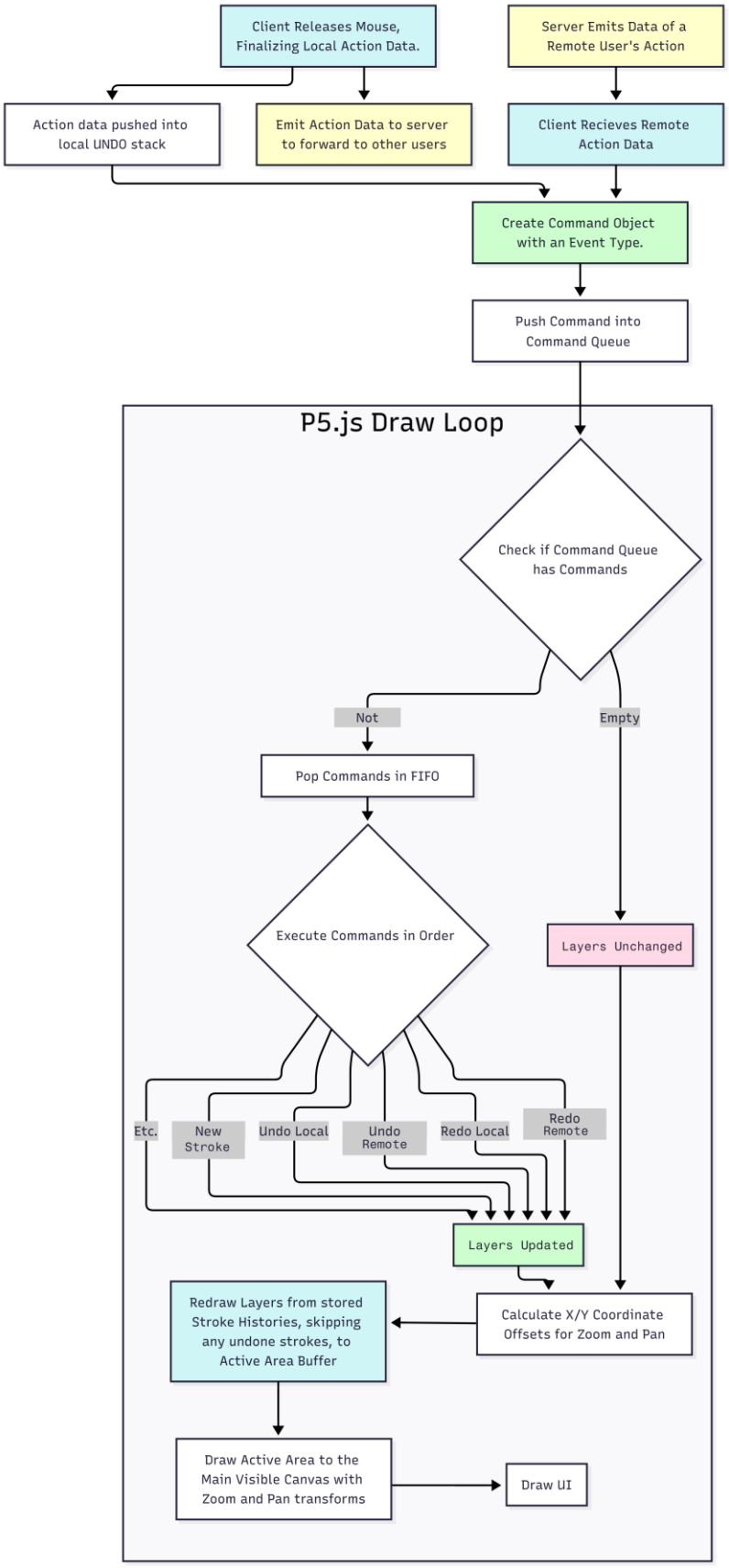


Figure 1: Flowchart of Draw Loop

The function that redraws layers relies on P5.js' expanded canvas functionality and each layer keeping track of its entire stroke history. Strokes are recorded on the client side by drawing on both live layers for visibility and a hidden graphics buffer called the 'action' layer, which consists only of the active stroke and nothing else. While the mouse is held and dragged across the screen, a bounding box for the stroke is calculated, expanding as the stroke grows in width and height so that only a small portion of the canvas is captured to minimize the amount of data needed to be transferred over the socket. This results in more efficiency in redrawing the canvas from sprites rather than having to recreate the exact action movements through other means. Every single action that is not an undo or redo is saved in the layer history, allowing future users who join to retrieve the canvas state from the server and synchronize the canvas state with other users. Regular strokes are applied to a single layer, while eraser strokes affect the layer mask instead, which is applied to the layer using the P5.js mask function to apply the expected erasing functionality.

Drawing Tools

When it came to the actual drawing tools, I utilized a combination of algorithms and manipulated the canvas on a per-pixel level, and some of P5.js' built-in tools. The pen tool is simply a combination of P5.js' functions. By combining the stroke, strokeWeight, and line functions with P5.js' current and previous mouse coordinate variables, it's very easy to draw a line of a certain color and weight between the client's mouse's previous x and y coordinates and the current x and y coordinates. Utilizing the default settings for the line function results in drawing a line that has aliasing and a circular shape, which was exactly what I was looking for in a default pen tool. For the pencil tool, I wanted to have no aliasing, which was not something that appeared to be possible with P5's built-in stroke function. Instead, I looked to Jack Elton Bresenham's line algorithm, developed in 1962 [7], for a fast and efficient way to draw a line between two points that would get the pixelated look I was looking for and be easy for clients to process live. My implementation of his algorithm generally follows the following pseudocode (Figure 2), with an addition for the user's stroke weight. I implemented it as a simple modifier that expands the pixels affected around the x and y points by half the stroke weight in the left, right, up, and down directions to achieve the correct stroke size centered around the algorithm-calculated slope of the line. The eraser also uses the same algorithm, but instead it affects the layer's mask. By erasing pixels in the layer's mask, it erases in a non-destructive manner that is conductive to how the undo and redo functions are handled.

```
plotLine(x0, y0, x1, y1)
    dx = abs(x1 - x0)
    sx = x0 < x1 ? 1 : -1
    dy = -abs(y1 - y0)
    sy = y0 < y1 ? 1 : -1
    error = dx + dy

    while true
        plot(x0, y0)
        e2 = 2 * error
        if e2 >= dy
            if x0 == x1 break
            error = error + dy
            x0 = x0 + sx
        end if
        if e2 <= dx
            if y0 == y1 break
            error = error + dx
            y0 = y0 + sy
        end if
    end while
```

Figure 2: Bresenham's line algorithm Pseudocode from Wikipedia [7]

For the flood fill functionality, I also ended up working with an established algorithm rather than built-in P5 functions because of the nature of how the layers are built dynamically from sprites. When the user clicks to use the fill tool, a snapshot of the layer is built from the stroke history and a span fill algorithm is used to scan adjacent pixels for color matching, as seen in Figure 3. I then fill in the appropriate pixels of the action layer as the algorithm continues to search, allowing the program to capture the filled areas as a finalized “sprite” that can be sent to other clients.

```
fn fill(x, y):
    if not Inside(x, y) then return
    let s = new empty stack or queue
    Add (x, y) to s
    while s is not empty:
        Remove an (x, y) from s
        let lx = x
        while Inside(lx - 1, y):
            Set(lx - 1, y)
            lx = lx - 1
        while Inside(x, y):
            Set(x, y)
            x = x + 1
      scan(lx, x - 1, y + 1, s)
      scan(lx, x - 1, y - 1, s)

fn scan(lx, rx, y, s):
    let span_added = false
```

```
    for x in lx .. rx:
        if not Inside(x, y):
            span_added = false
        else if not span_added:
            Add (x, y) to s
            span_added = true
```

Figure 3: Span fill pseudocode from Wikipedia [8]

The final tools are the zoom and pan tool and the eyedropper tool. The eyedropper also builds a snapshot of the layer, and takes the color data from the pixel the user clicked on and sets the user's color to match. Zooming and panning are modifiers that are applied when calculating the relative x and y coordinates of the user and when drawing the final active area buffer to the visible canvas. Zoom can scale the canvas from 0.5x to 5x size, and the end user can pan as far left and right as they wish. Hitting space resets the zoom and pan modifiers to recenter the canvas in the screen.

Layers and Canvas Attributes

Layers are made up of layer objects, which each have their own graphics buffer in a live layer, a graphics buffer for the mask, a baked image, a stroke history array to store action objects, a history start array that allows us to push and pop new starting indexes from clearing, and variables for whether or not the layer is removed. Removing a layer consists of removing its associated P5 elements and its arrays, while adding a new layer involves creating a new object and new P5 buffers. Currently deletion and addition of layers cannot be undone.

Users are also able to change the size of the canvas and clear layers. Clearing the layer changes the index that layers start drawing from the stroke history by pushing the latest, which theoretically allows this to be undone, but it is not currently implemented. When the canvas is resized, all layers have their graphics buffers re-created and data from the old buffers is copied onto the new buffer. This also resizes the active area buffer, the action layer where strokes are drawn to for capturing, and the checkerboard for the background. Layers are drawn to the active area in order, with the layer with the highest index on top. Unfortunately, I did not implement rearranging layer order or locally hiding the visibility of layers.

Sockets

Transporting Images and Text

Setting up sockets was incredibly accessible with Socket.io. By installing the Socket.io node module and including a link to the Socket.io client file in the header of the drawing page, I was able to

immediately begin seeing client connections and to be able to send things to and from the server. The main idea behind Socket.io is sending and receiving data via "events" triggered by both the client and the server [9]. By attaching these send events to actions taken in the client, it was very simple to set up the server to broadcast the messages to other connected users. I chose to send the entire pixels array over the socket rather than converting each sprite image into another form, and had to expand Socket.io's buffer limit from the default 1e6 (1mb) [11] to 1e8 (100mb), as the client's socket was crashing and disconnecting due to attempting to send too large of a buffer to the server. Users do not have accounts, but rather user information is set on the settings page and stored in local browser storage, and the name, color, and socket id are saved by the server for host and admin verification.

Rooms

Socket.io also has support for rooms, which are arbitrary channels that sockets can join and leave that is for broadcasting actions to a subset of connected clients [10]. When a socket connects to the page of a particular room, I'm able to assign them to a socket.io room that has a matching unique id to the room's unique URL. At this point, actions from the server, such as when a client sends their x and y positions when drawing or sends a chat message, I can broadcast the message to the specific room only. This also allows for support of keeping track of each individual room's canvas progress in the server as well, so that the server can synchronize the state of the canvas whenever a new user joins the session.

Moderation and Advanced Permissions

Sockets also allowed me to enact and enforce moderation and advanced permissions with server verification. By keeping track of who is the host on room creation and connection in the server, I can verify that the host is performing an action that requires advanced permissions. The host has full control of all advanced permission actions: clearing a layer, deleting a layer, adding a layer, resizing the canvas, kicking users, and promoting non-host users to admins to give them the ability to also have more advanced control over the canvas and to kick regular users. By having clients request to perform an elevated permission action, I'm able to have the server perform verification and respond with either confirmation by broadcasting the action to all clients or sending a kick event to the requesting client for requesting an administrator action when they are only a normal user.

Results

The results of this project are a functional PaintChat program, although it is not as advanced as the current well-known closed-source options. The program successfully includes the core concepts of PaintChat: a functional art program, functional chat, and functional canvas sessions that allow various groups of users to chat and draw at once. The art program has several different standard tools expected for a basic art program: a drawing brush, an eraser, fill, undo, redo, zoom, pan, and an eyedropper. It also features more advanced features like layers and transparency, which is standard in the more modern paint chat applications.

Users are able to open their own room with a custom title and unique URL path, and can choose whether or not to have it be a public room that will be listed on the rooms page of the website or a private room that will only be accessible by those who know the direct URL. Creating a room and the public rooms list is illustrated below in figures 4, 5, and 6.

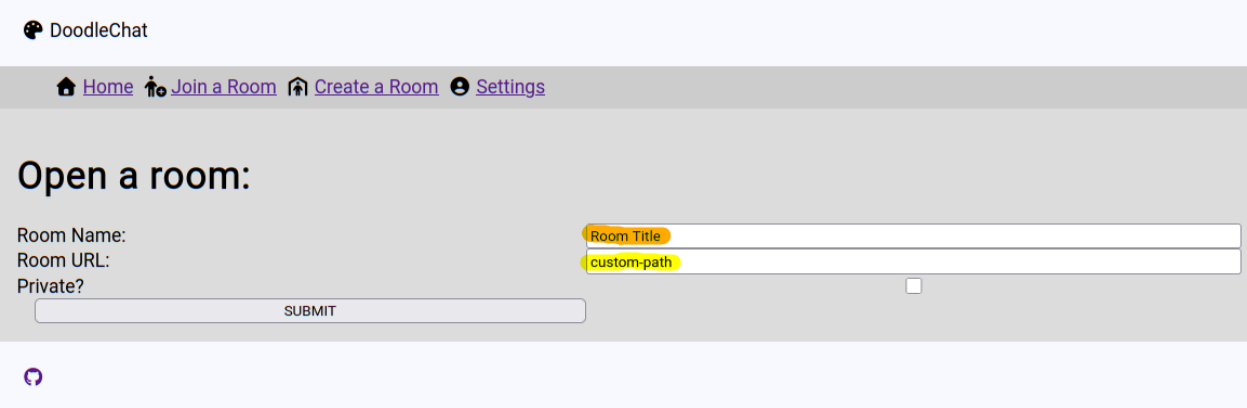


Figure 4: the Create a Room page. Only the URL must be unique.

Figure 5: Showcases how the Room Name is the page title and the Room URL is the custom path for the room.

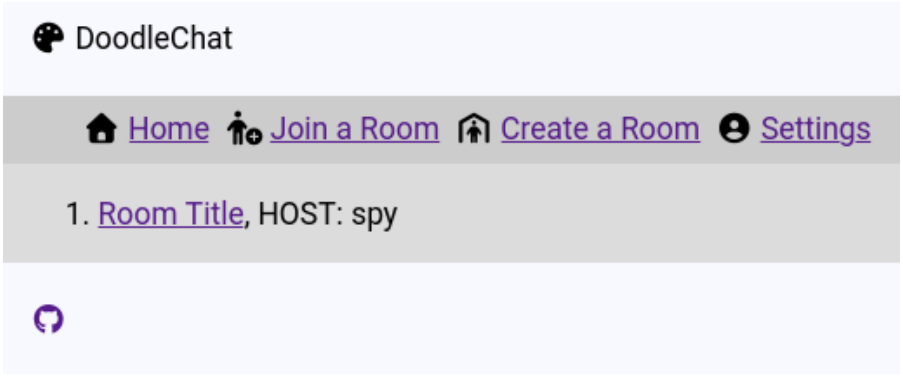


Figure 6: the Public Room page with the public room list, which shows the room created in the last two figures. It also lists who the host is.

Upon creation of the room, the user is met with a canvas with several tools and can begin drawing and chatting immediately. A screenshot of the interface is pictured below in Figure 7, labeled with boxes surrounding different sections of the interface, and has examples of what the tools look like when used drawn on the canvas. The green box indicates the various tools you can select from and the undo/redo buttons, which also have keyboard shortcuts. The current active tool is highlighted with a yellow background. The blue box is surrounding the color picker and alpha slider, which lets you change the color and opacity of your stroke. The red box surrounds the stroke width slider, which changes the size of the brush. The black box is the dropdowns with various functions. The file action allows you to export either the current layer or the entire canvas as a flat image. The edit dropdown contains the action for clearing the layer. The layers dropdown allows you to change between active layers and add or delete layers if you're an admin. The exit button returns you to the main page. The yellow box is text that displays your current layer, tool, color, and the total number of actions you have taken since joining. The chat and current user list is surrounded by the magenta box; the room's host has a crown icon, while any admins have a knight icon next to their name. The chat box contains all user-sent text, as well as any server-confirmed actions such as resizing the canvas or users joining and disconnecting.

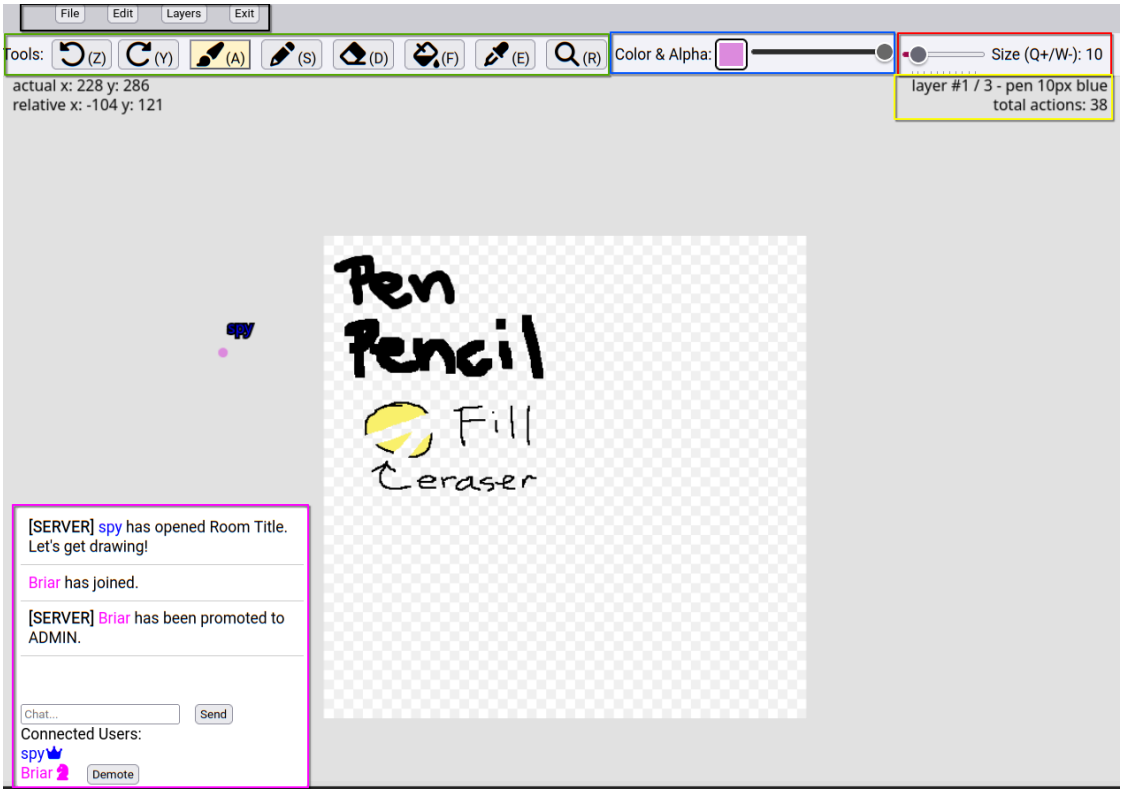


Figure 7: A screenshot of the drawing room interface that is marked up with various colored boxes, annotated in the above text.

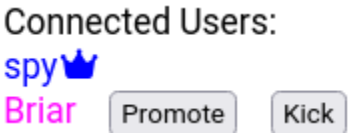
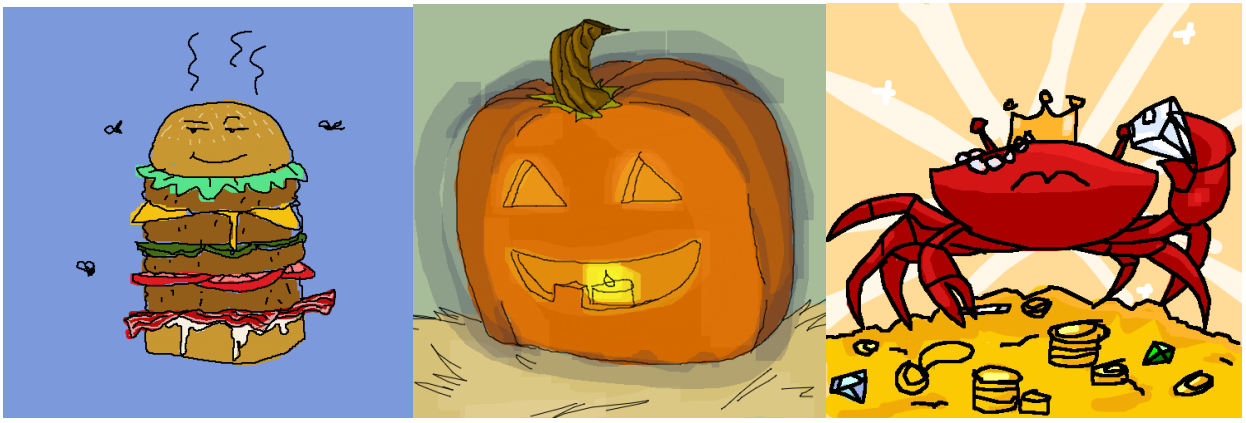


Figure 8: Showcases the user actions shown only to the host. The Host can promote or demote other users to Admin status, and kick users. Admins can also kick normal users.

While not a program designed for creating the most advanced artwork, it can still be used to make fun doodles and chat with friends. Some artwork created while testing is below, Figures 9-12.



Figures 9-11: Artworks created in DoodleChat by three people drawing together.

Figure 12: Spyro drawing to test the limits of the tools and canvas performance. 805 actions on 3 layers.

Discussion

Through taking on this project, I learnt how to utilize web sockets through Socket.io, experimented with communication between multiple clients and a backend Node.js web server, and learnt a lot about working with HTML5 canvases to render interactive graphics in the browser. It was quite a challenging project as I had never worked with web sockets or a server in depth before, and my experience utilizing HTML5 canvas elements in my work beforehand was minimal.

Adding functionality to the canvas for making artwork and designing a way to store the canvas state in a manner that would be easy to synchronize across clients and send over the webserver was the biggest hurdle of the project. P5.js is a powerful library for displaying graphics on the web through its extension of HTML5 canvas elements, but the documentation and its provided examples are very simple and straightforward representation of what the extension is capable of. Searching online for tutorials specifically relating to creating a drawing program results in examples that were not particularly useful for the specific issues I was encountering, as in many instances they were addressing creating a program that was only a singular layer on a solid white background. P5.js makes it very easy to create simple lines and other 2D primitives, but I ended up deciding to work directly with the pixels array in several

circumstances because of some quirks of the library in combination with the more real-time friendly but complex design choice of storing the strokes as sprites instead of sending the entire canvas with every action.

For example, P5.js has an erase function that, when called before a line or shape function, will instead erase pixels on the canvas rather than adding colored pixels to the canvas. However, erasing the canvas in this manner cannot be done when drawing an image object to the canvas with the image function, which was an issue when attempting to figure out how to re-create the erase action with a sprite when an undo or redo was called and the layer would need to be redrawn. It also would become an issue in the future if I ever chose to utilize brush shape images as the base for custom brushes. The P5.js stroke and fill functions' application of color is also additive, meaning that there was no other way aside from the erase function to remove alpha from the canvas through directly using its 2D primitives, as setting the color's alpha value to 0 does not affect the canvas in any manner. There was also the issue of how to affect the background layers with the eraser in a manner that could be undone later, which was not something that had to be considered with the pen tool when using P5.js functions to draw solid lines. Implementing an eraser that could also function with an undo and redo required a complete overhaul of the layer objects to approach erasing in a different manner. With Professor Reale's guidance, I was able to design an eraser that would instead work on a layer mask that utilized the P5.js image object's mask function to remove pixels rather than using P5.js's erase function. This made undoing utilizing images possible, but also meant that when redrawing the layers it would have to be recreated as a P5 image object in order to utilize the P5.js' mask function to delete sections of the mask. Other quirks of the library required experimentation to discover. Examples include needing to specifically use the copy function to copy pixels from one image object to another image object instead of the image function that is used to place images on the canvas, having to explicitly set the pixel density of the canvas and other created P5.js objects to 1 to avoid encountering issues when redrawing sprites to specific coordinates in the canvas and manipulating the pixels array across higher definition displays, and the need to call P5.js' noSmooth function when redrawing images to the canvas to prevent blurring when redrawing sprites.

As soon as I was able to overcome the hurdles caused by learning the quirks of P5.js and the issues with getting the library to work within the design intentions of the program, the project became much smoother to finish. Working directly with the pixels array in P5.js is incredibly simple, fast, and efficient for updating the canvas and implementing algorithms commonly found in computer graphics.

The P5.js pixels array is a flat array of all the RGBA values of each pixel in the image and can be looped through with a nesting loop for x and y values. Indexes of individual pixels calculated by adding the current x value and y values multiplied by the canvas width, and then multiplying that resulting value by 4 and its RGBA values can be directly set with either assigning values to the associated array indexes directly or calling P5.js' set function. My implementations of the scan flood fill and Bresenham's line algorithms was very easy to incorporate with P5.js functions with the direct access to the pixels array that P5.js supports. Socket.io was incredibly straightforward to implement and worked perfectly with the P5.js library, as P5.js has event-driven mouse functions that tied in nicely with the event-drive socket emitting. For example, I was able to send data to draw on layers live using the canvas action function from remote users due to the mouse dragged function, and was able to then send a finalized stroke to all connections by emitting the object with a socket event through the mouse released function of P5.js. Sending the finalized sprites was as simple as sending the pixels array and converting it back to a P5 Image object on arrival.

Knowing what I know now, I think I would mainly change small things about the project. I would spend a little more time researching what might be the most performant canvas library, as P5.js' performance does begin to take a hit when the number of strokes begins to creep towards being in the thousands, and there are a lot more canvas libraries out there than I realized going into this project. I also think that some of the ways I had to approach working with the different types of P5 objects to get the program to do what I wanted it to do might not have resulted in the most efficient approaches. I also think that I would spend more time on the logic of redrawing layers to account for "baking" older strokes to reduce the amount of strokes needed to be redrawn by the client and help with performance, as this current version unfortunately does not have baking implemented in it yet and I'm unsure if implementing baking will be an issue with the current design.

Conclusion and Future Work

I accomplished what I set out to do by creating a working PaintChat application that utilized web sockets and a server, even if I made some missteps in the design process that created some difficulty getting off the ground and had a hard time with getting my chosen canvas library to work with the design of the program as a concurrent, multi-user experience. If I were to continue working on this project, I would work towards making the end-user experience better. I would want to improve the efficiency regarding how stroke data is sent between the server and clients over the web sockets,

implement user accounts with a backend database so that moderation could be more robust and it would be possible to support persistent rooms on the server, and I would attempt to implement more advanced brushes as the tools I have are very minimal with some glitches with opacity.

References

[1] “oekaki Etymology,” *Wiktionary*, 2022. https://en.wiktionary.org/wiki/oekaki (accessed Dec. 10, 2025).

[2] satopian, “GitHub - satopian/poti-kaini-EN: POTI-board EVO English ver, The OekakiBBS for PaintBBS NEO, tegaki.js,AXNOS Paint,ChickenPaint and Klecks. (PHP7.4 - PHP8.5) https://paintbbs.sakura.ne.jp/poti/,” *GitHub*, 2025. https://github.com/satopian/poti-kaini-EN (accessed Dec. 10, 2025).

[3] “Urban Dictionary,” *Urban Dictionary*, 2025. https://www.urbandictionary.com/define.php?term=paintchat (accessed Dec. 10, 2025).

[4] “HelloPaint by iScribble: Drawing Together Made Easy!,” *Kickstarter*, Jul. 02, 2022. https://www.kickstarter.com/projects/iscribble/iscribble (accessed Dec. 10, 2025).

[5] The Magma Team and A. Burton, “The Story and Development of Magma, a Collaborative Art Platform,” *80.lv*, Apr. 13, 2023. https://80.lv/articles/the-story-and-development-of-magma-a-collaborative-art-platform (accessed Dec. 10, 2025).

[6] Agamnentzar, “Aggie.io,” *Aggie.io*, Mar. 28, 2019. https://web.archive.org/web/20190328171346/https://aggie.io/ (accessed Dec. 10, 2025).

[7] Wikipedia Contributors, “Bresenham’s line algorithm,” *Wikipedia*, Jun. 02, 2021. https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

[8] Wikipedia Contributors, “Flood fill,” *Wikipedia*, Sep. 08, 2025. https://en.wikipedia.org/wiki/Flood_fill#Span_filling

[9] “Tutorial step #4 - Emitting events | Socket.IO,” *Socket.io*, Oct. 28, 2025. https://socket.io/docs/v4/tutorial/step-4 (accessed Dec. 13, 2025).

[10] D. Arrachequesne, “Rooms,” *Socket.IO*, Apr. 30, 2021. https://socket.io/docs/v3/rooms/

[11] “Server Initialization | Socket.IO,” *Socket.io*, Oct. 28, 2025. https://socket.io/docs/v3/server-initialization/#maxhttpbuffersize (accessed Dec. 13, 2025).

Appendix

GitHub Repository: https://github.com/paw/doodlechat

To Install:

- Download the code. You must have Node.js/NPM installed on your computer to run the program.
- All dependencies can be automatically downloaded by running “npm install” in your terminal before proceeding to the next step.
- Start the webserver locally in your terminal by running “node server.js,” and it will open a server that can be accessed at localhost:3000
- To test the functionality with other users across the internet, the easiest and safest solution is to install Tailscale on your device and use the funnel option.
- To host a version for an actual webserver, follow DigitalOcean’s tutorial for Deploying Node Applications on a VPS.
- **Please be aware if hosting it live:** there is no database support currently available. This means anyone who sets a name and color and saves it to their local storage can create a canvas and draw, and that kicking unfortunately does not prevent a user from re-joining the room due to being assigned a new socket id every time they connect.