

Collab.io - Real Time Collaborative Sketching

Max Mayr, Max Zauner

June 25, 2018

Abstract

During this semester we had to design implement and present a project. We decided that we wanted to try something new using web technologies. Therefore we choose to design a project around real time collaboration using Websockets and WebGL. Our concrete idea is a sketching platform that allows users to open a new workspace or join a existing one. Within these workspaces the users can sketch ideas on an infinite canvas using different colors. Thereby the strokes should be visible while they are drawn on other users screens. This actually gave us a bit of a headache as this report will show.

**websockets, angular, webgl, real-time collaboration,
docker**

1 Introduction

1.1 Motivation and Goal

The motivation for this semester's project was to write an application that would challenge us to use advanced web technologies and be useful at the same time. After some confusion and some dead ends we modified our project to the following vision:

We wanted to build a collaboration drawing platform where the user can sketch any project desired that can be drawn and share the drawing process in real time to other clients that are connected to the same instance as well. It can be imagined

as a drawing board like OneNote but in the browser. In addition it is now possible to join other drawing sessions and see in real time what other people are drawing.

We knew that that would be challenging, as there there is no system that can be used to easily add drawing to a web project that also allows the programmer to easily connect the drawing engine to some form of socket communication. So the goal of this project was to create a rendering engine based on web technologies that allows the integration of real time network solutions like websockets to be integrated to enable real time collaboration.

2 Technology

This chapter explains the core concepts of our project and describes the core technologies we intended to use to implement the software. For simplicity's sake we will divide the whole technology into two major parts, the graphical render technology and the connection technology.

As stated before, the whole system consists of two major parts that must be considered at first before being able to implement the software. One aspect is the concept on how to graphically render anything drawn on a surface including all kinds of inputs like pens, fingers, mouse pointers etc. and the second aspect is the concept on how to connect clients together and how to organize all connections to being able to propagate any change in real time to the correct clients.

2.1 Graphic Programming

To understand graphical programming we first have to take a look into triangles. Triangles are how faces are represented in a 3d model. Each face consists out of three vertices that are basically just points within the 3d world space of the scene. The world space is the space the objects are placed within. To map what we currently see in the world space to the screen space (out actual display) we need a camera. The camera is basically an eye and the world space is an an object. We can walk around the object by moving the eye around or we can move the object and change its position in the world space.

If we connect two triangles we would currently double two points because they are used by both triangles. To save memory and gain performance we can create so called indices that tell our gpu - the graphical processing unit - what points belong to what triangle. Thereby we can either go clockwise or counter-clockwise, depending on the graphic framework this tells the gpu what is the front side and

what is the back side. This is important because the graphic card will by default only render the front side of a triangle to improve performance.

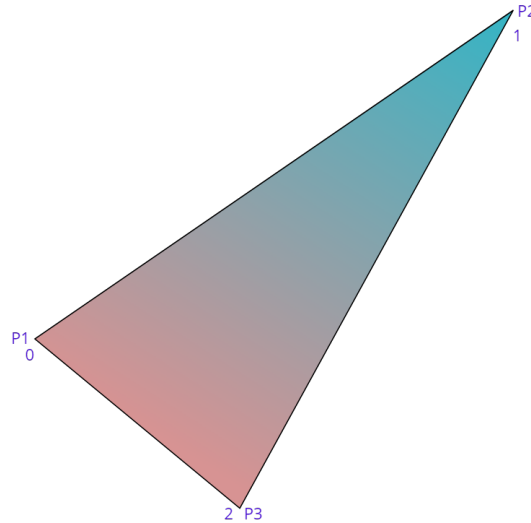


Figure 1: Vertices, Indices, Triangles

We already talked about cameras and that they help us map the world space to our screen space. What we didn't talk about is that there are different projections we can use to do our mapping.

2.1.1 Perspective Projection

The first projection is the perspective projection it has a field of view that defines the extend of the scene that is currently visible. This projection can be compared with our human eye. If we get closer to an object it will be bigger so our view frustum (the pyramid of where we look) starts small and extends more the farer we are away from the camera.

The perspective projection is is the one that is used the most at its the one that can display "normal 3d" scenes.

2.1.2 Orthographic Projection

The orthographic camera uses the orthographic projection to display the scene. This projection maps everything to the screen as if it where just 2d. This means we can move around but there is no perspective involved. Everything is always

the same distance away from the camera and the depth is just used to know what is in front of other objects in the scene.

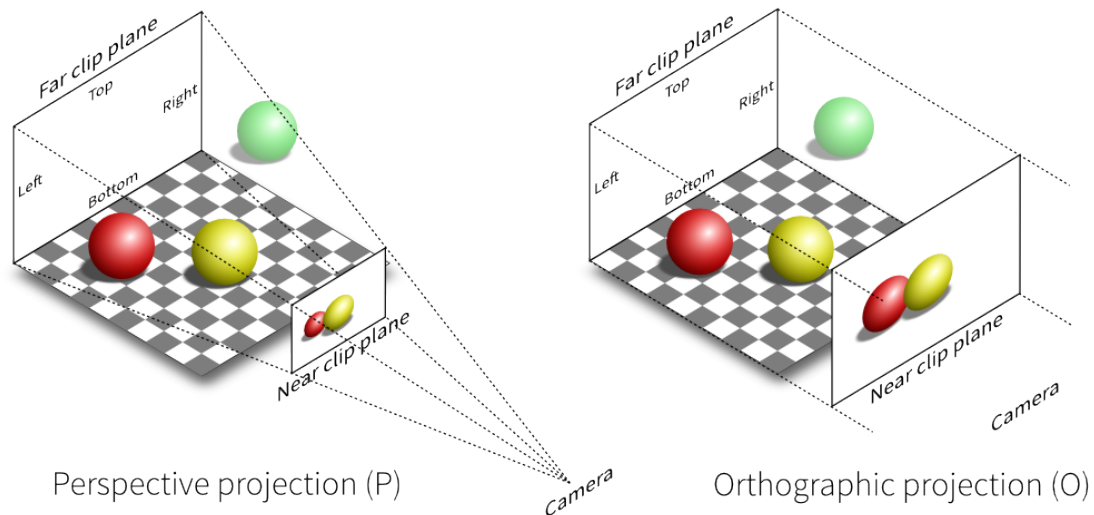


Figure 2: Perspective vs. Orthographic projection

2.1.3 three.js

three.js¹ is a open source graphic library that internally uses WebGL. It runs in modern browsers and provides an API that makes it more easy to get started with WebGL. Additionally three.js provides a lot of helper functions that allow quick prototyping.

To display a 3d scene using three.js one needs a scene that contains what we can see, a camera that defines how we see and a renderer that displays the result.

2.2 Realtime Data Transmission Concept

In order to being able to properly propagate any drawing to all clients that are connected to the same sketching board it is important to not only develop a good strategy on how to connect the correct clients together but also to think about what to transfer over the socket so all clients are able to render the same sketching board and also render the drawing of other clients in real time as well.

¹threejs.org

2.2.1 Room concept

The most important part of connecting clients is the possibility to divide connections into rooms. The idea was to have user entities that can enter or leave rooms to decide on what project they want to collaborate on. Once a user enters a room he is connected to all other people in the room as well to not only send his input to all clients in the room but to also receive all the other drawing inputs from other clients. As there might exist multiple clients possible the whole system it is evident to make use of the room concept that exists in context with websockets.

By creating rooms the the separation of multiple websocket clients into different rooms is easily possible. Any emitted message on the socket will therefore only reach clients that are in the same room. The rooms must be persisted on the server and every websocket message can be emitted to any websocket room.

After logging in the user is prompted with a screen to select or create a room to collaborate. The socket rooms are be handled internally without any lists on the frontend to select from. The system should be as easy to use as possible. In order to being able to join a room only the name must be known as the room's name must be unique anyways. Once the user joins a room this action is persisted internally so the user can reload or reconnect to the same room without experiencing any problems or without having to rejoin the room on the room selection screen.

2.2.2 Websocket Transmission

The websocket transmission concept is also of utter importance as this is the essential part of the system that is responsible for enabling the real time rendering of multiple client's drawing input on all the other connected clients.

Thankfully the concept of the websocket transmission is not that complicated, as the more difficult part is the separation of the clients. Once every client only sends its data to the other clients in the same room the data that is sent is pretty obvious. As the rendering engine depends on mouse events and the position of the mouse, it is only logical to sent exactly that data over the socket as well.

Every time the browser records a mouse event, that data will be sent over the websocket as well. To being able to separately render each user on the client it is also important to send the user's id over the socket as well to being able to divide the different input steams of events on each client.

Last but not least there must also be a miscellaneous part of the transmission including color and thickness of the stroke. That part is also important as every

client probably uses its own input method like pens, fingers or mouse. This is also the part of the transmission that is extensible in the future and can contain various additional information to being able to properly propagate all information to the clients.

3 Backend Implementation

This chapter is all about the concrete implementation. In previous chapters the technology on how we intend to achieve our project's goals were elaborated and this chapter aims to give insight into how the backend is implemented and what technologies are used.

3.1 Typestack

The backend is implemented in serverside JavaScript called Node.js. For this project we decided to use a very remarkable project that consists of many libraries all aiming to improve the JavaScript backend development experience. The project is generally known as typestack. The used libraries will be explained in the following sections.

3.1.1 Dependency Injection: typedi

As the educated reader might have already guessed, the whole typestack project only consists of libraries completely written in typescript. This makes the development of the backend a smoother experience and eliminates a lot of the type related problems and errors. Also the IDEs can more easily provide code completion features.

One of the most essential libraries of the backend implementation is the typedi library though. It offers dependency injection with typescript support. With typedi it is possible to use interfaces instead of concrete implementations which makes certain parts of the code very reusable or interchangeable. The concrete implementations can be injected via dependency injection.

The code quality of the backend is quite high, as dependency injection cleans up a lot of the code that is required to separate the different services and structure the whole project.

Furthermore, the typedi project enables annotation based injection which removes even more boilerplate code which in return even more improves the overall code quality of the project.

3.1.2 Restful Controllers: routing-controllers

The routing controllers project aims to make it really easy to create restful servers with Node.js and typescript. By using this library rest controllers can be created by only having to annotate classes with certain keywords. Even the underlying technology is interchangeable. The developer can choose between koa and express for example which are both Node.js libraries to create http-servers.

The library includes multiple helping functions and annotations for creating routes and handling requests. Http routes can be created for example by annotating a function with `@Get('/foo')` or `@Post('/bar')` for example.

3.1.3 Database: typeorm

The typestack also provides an extraordinarily sophisticated ORM implementation for typescript that can be combined with the typedi dependency injection. Typeorm is also independent from the underlying database technology and provides a software layer to manipulate the database.

By injecting the different entity repositories it is easy to implement services that have access to the database. Additionally, the library improves the code quality as it is type safe as well due to the usage of typescript.

3.2 Websockets: socket.io

For the websocket technology we chose to use socket.io as this library provides the best features for a seamless websocket experience. The library makes it easy to connect clients and even provides implementations for reconnecting clients and having fallback implementations.

The challenge with this library was to integrate it into the typestack project, as it uses different services throughout the whole application. We tried to also provide a websocket interface that the socket.io service implements to make the technology interchangeable as well and to comply to the typestack philosophy.

3.3 General implementation details

As described before, the backend is responsible for managing users and rooms. Therefore the backend must manage the creation, deletion, and changing of the room and user entities. One room can contain many users but a user can only be in one room. This functionality is achieved by restful routes.

Also, calls to the rest api have to be authorized. We use the JWT (Json Web Token) technology on the server to sign authorization tokens that can be used by the clients to send authorized requests to the backend.

Once users are created and users have authorized themselves on the backend they can now create rooms which is also a restful route. The server creates the room and sends back the room details which have to be used by the client to then establish a connection to the websocket part of the backend.

While the websocket connection is active almost all data exchange happens over the websocket. The client has to send the room's id over the socket in order to being connected to the room. The server then adds the user to the room on the database to persist the fact that the user joined the room to being able to restore the users connection if the user reloads.

If the user reloads he can query his room status over the rest api as well. There are routes to check if the current user is inside a room. If a room is found it is sent to the client which can then simply react by reconnecting itself to the room the user is currently in.

4 Frontend Implementation

Because we wanted to create something easily sharable we decided to deploy the Frontend of our application as a single page application. This single page application can be accessed by navigating to a url in the browser. This way no user needs to install anything to collaborate with others.

Because one of us is strongly familiar with Angular we decided to use this framework to implement our application. To also provide a learning experience we decided to play around with some special features that are rarely used in normal websites. During this process we found out that we could use some of those features to create a easily maintainable and extendable solution to manage WebGL objects.

Before implementing and designing the application we first focused on the technical

part of the application. The most important part thereby was the rendering of our input strokes.

4.1 Prototypes

We wanted to make sure that we provide the most beautiful ink representation as possible with the highest performance. To archive this we first tested drawing using an normal canvas object with the provided methods to draw lines. This was really fast but it was not possible at all to render strokes with a fluctuating thickness. A fluctuating thickness naturally occurs when we write something on a real piece of paper, therefore getting rid of it makes the inking feel artificial and wrong.

After researching we noticed that we could also use SVG's to represent ink strokes. The SVG way of representing strokes looked really promising because we could zoom, pan, and move around the page using native browser controls, but after some performance testing we quickly discovered that its unrealistic to try to archive the performance we want with SVG's.

The next prototyping iteration brought us closer to the hardware as we where utilizing three.js to render our strokes to the screen. Our first and unoptimized was already a lot faster than our previous prototypes. Therefore we decided to stick to three.js - that is using WebGL internally - as our render engine.

To draw our strokes using three.js we have to calculate all points that represent the shape of our stroke therefore we connect all input points with a line and extend the corner points of each segment outwards by the thickness or pressure of the stroke. This already provides really good looking strokes but by saving the points from the last segment and reusing them for the next segment we not only save cpu power but also smooth the stroke as now no hard corners will be visible any more within the stroke.

To render the stroke now we take all the points and construct two triangles per segment. These triangles inserted into an three.js geometry object with also contains a material that is used to render the shape. After inserting the geometry gets added to the screen and will be displayed in the next render cycle.

The next problem we had to solve was to smooth the stroke somehow as the direct input was often a bit noisy. First we reduced the number of points using the Ramer–Douglas–Peucker² algorithm, whereby we placed around with the numbers

²de.wikipedia.org/wiki/Douglas-Peucker-Algorithmus

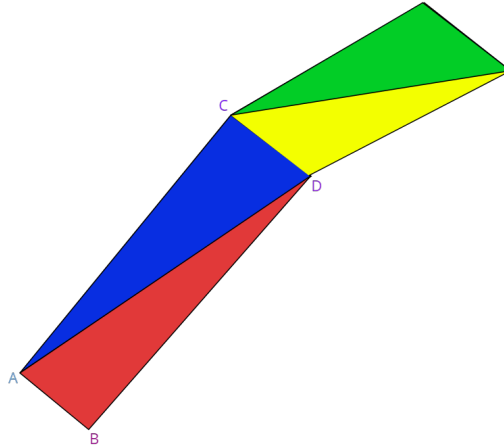


Figure 3: Anatomy of a stroke

to ensure we don't smooth the stroke to hard as this would make it hard to draw precise sketches.

Because we still weren't satisfied with the results we also decided to calculate additional points that would make the stroke more natural using Bézier curves³. To archive this we had to implement a curve fitting algorithm that approximates the best possible Bézier curve for our given input.

4.2 Live Rendering

The biggest problem with our three.js prototype was to enable live rendering again. Live rendering means that you can see the ink flowing out of the pen tip instead of rendering only the finished stroke once the mouse stops drawing. Because we were working with data on the GPU we had to figure out a way to dynamically update our data and tell the GPU to render the data again.

After a bit of researching we found out that three.js objects have a flag that indicates that they need to update. With this information we implemented our first prototype that featured live stroke rendering. Because we didn't have the full stroke at render time we had to comment out our smooth render logic and point reduction as these algorithms were constructed for full point sets.

To limit the number of points we took a simpler approach and now just check if we are at least not on the same point and that the next point is at least 0.5px

³en.wikipedia.org/wiki/B%C3%A9zier_curve

away from the previous point. This already helped a lot in reducing the number of points we get.

For curve fitting we first thought we need to come up with another approach, but after going through the algorithm step by step we noticed that we could just delay rendering of the stroke by one point to apply the algorithm just for two segments at a time. This way we don't need to calculate as much as normally and still get beautiful results.

When we were testing this solution everything seemed fine but on fast moves and at the start and end of a stroke we noticed some significant performance drops. After some researching we first enabled rendering only when the camera changes or when we actually draw something on screen. This saved us a lot of cpu/gpu power in idle mode but while drawing we still had performance drops. We found out that we send twice the amount of data that we actually need to the gpu. We did this to not have to set all indices of the stroke manually and to test out a lot of stuff using the simpler but also slower Geometry object instead of the faster one called BufferGeometry.

After figuring out all the mathematics behind calculating the correct indices and uploading the data we finished everything we need for our render logic. Our renderer could render thousands of lines without any problems. Therefore we decided to stop performance optimization for now focus on building a working product that can be used by everyone.

4.3 A-Frame Architecture

After several prototypes we noticed that we could use some advanced features of Angular to represent our scene as a DOM structure. In order to do this we separated all three.js components and implemented each of them as its own angular component. To grab all nested components of a specific type we query our view and inject all entries in the list into our scene.

The idea for this kind of representation came from the WebVR framework A-Frame⁴ that gets developed by Mozilla.

Our solution is easily extendable as we can just switch one component out with another without having to change anything anywhere else. This allows us to try out new stuff fast without having to worry what it might break somewhere else.

⁴aframe.io

```

</three-orthographic-camera>/three-orthographic-camera>
controls>
▼<three-scene>
  <three-orthographic-camera></three-orthographic-camera>
  <!-->
  <three-stroke></three-stroke>
  <three-stroke></three-stroke>
  <three-stroke></three-stroke>
  <three-stroke></three-stroke>
  <three-stroke></three-stroke>
  <three-stroke></three-stroke>
</three-scene>
<canvas width="581" height="772" style="width: 581px; height: 772px;">
</three-renderer>
</three>
div>

```



Figure 4: three scene strokes

5 Frontend Workflow Walkthrough

In this chapter the system will be explained in a more practical manner. By including screenshots it should be easier how the system is intended to use and it is easier to get a grasp of the working frontend.

5.1 Login

First the user is prompted with a login as you can see in Figure 5. As discussed before, having users is very important in order to being able to distinguish different users and to being able to arrange them to the corresponding rooms they're currently in.

5.2 Room creation/selection

Not all users are in a room though, thats why the user is prompted with a room joining/creation screen. As it can be seen in Figure 6 the user can either create or join a room by name.

The image shows a login interface. At the top, the word "Login" is centered. Below it, there are two input fields. The first is labeled "Username" and contains the text "test3". The second is labeled "Password" and contains four dots. To the right of each input field is a small icon for toggling password visibility. Below the password field is a green button with the word "SUBMIT" in white capital letters.

Figure 5: Login screen

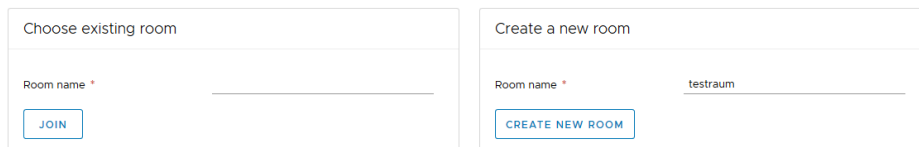
The image shows two side-by-side forms. The left form is titled "Choose existing room". It has a label "Room name *" followed by an empty input field. Below the input field is a blue button with the word "JOIN" in white capital letters. The right form is titled "Create a new room". It has a label "Room name *" followed by an input field containing the text "testraum". Below the input field is a blue button with the words "CREATE NEW ROOM" in white capital letters.

Figure 6: Room screen

5.3 Drawing

Finally the user is able to draw as the drawing screen can only be reached if the user is logged in, connected to the socket, and connected to a certain room. In the screen 7 the drawing screen can be seen. Also, if the user decides to reload the page, he directly rejoins the same room and can draw again right after the reload.

6 Challenges

One of the biggest challenges in this project was to find an efficient way to render out input strokes. After several prototypes we found a solution that seemed like a perfect fit as the performance with WebGL was just astonishing. One we had the basic rendering solution the next challenge was the live rendering. Because we work with objects on the graphics card we have to dynamically shovel data from the CPU to the GPU in order to render it. After we had set up our dynamic buffer structures and managed data cleanup to get rid of memory leaks the solution worked perfect for a single simultaneous stroke, however it was not possible to render multiple strokes that are being rendered at the same time.

To solve this problem was the hardest challenge we had to face as we had to refactor the project from ground up in order to make the live rendering independent from

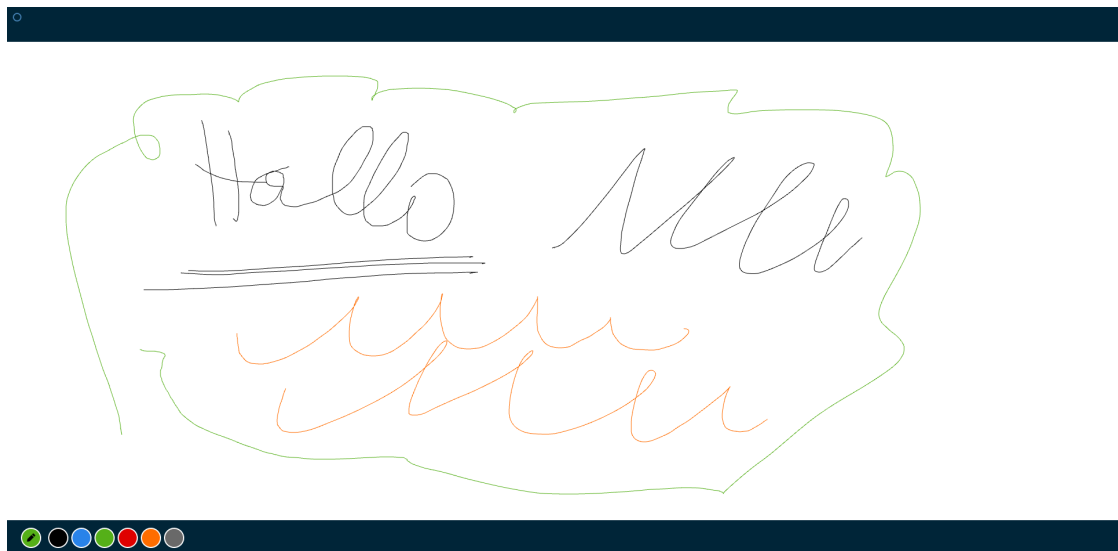


Figure 7: Drawing screen

every other component of the application. Therefore we structured and isolated every functionality we could into several components that are exchangeable by default.

The next challenge was to normalize all input methods we had into an uniform structure to allow touch, pen and mouse input. For this we utilized RxJS which is a library that focuses on streams. Basically we create a stream for each event type and create a new stream while our mouse is down this stream then get grouped per input identifier and emits a new stream that contains all movements of a single stroke event.

7 Future Work

In the future we want to extend this project with private and public rooms. This would allow rooms that strangers can join and collaborate in.

Additionally we want to provide a live preview where other users have their input device on the screen and an indicator who is currently in the room.

Of course also the tools could be extended to allow fixed widths with a selector. Other tools could be an eraser, an selection tool to move strokes around or a tool that allows embedding of images and other files within the workspace.

Also we could adder an automatic saving mechanism that saves finished strokes to

the server so reloading clients can see the current status quo of the drawing that had already happened before they joined.

8 Conclusions

The project took us way longer than we thought because of all the prototypes and the streaming logic. Therefore we couldn't implement all the features we had initially planned for the project. Nonetheless we have now created a project that is pretty unique and shows what is possible in todays browsers.

Also it was a great learning experience for us two as we had never done anything in WebGL before and with this project we also had to learn a lot of the internals to improve performance.