Collavrate

University of Toronto at Mississauga, Spring 2015

Homepage:	http://collavrate.zohaibahmed.com
Github Repository:	https://github.com/ZohaibAhmed/collavrate

Zohaib Ahmed
email: zohaib@rocketfuse.com
Tanzeem Chowdhury
email: tanzeem.chowdhury@mail.utoronto.ca

Introduction

Collavrate is a Virtual Reality (VR) application built entirely on the Web using the WebVR API being developed by Mozilla. It allows users to collaborate and create 3D models while being in a virtual environment. By leveraging the Myo armband, Collavrate enables users to draw, model, and control movement using gestures.

Overall Architecture

Collavrate uses the following libraries:

- Three.js for graphics (https://github.com/mrdoob/three.js/)
- Myo.js for Myo Javascript Bindings (https://github.com/thalmiclabs/myo.js)
- csg.js for combining 3D solids (http://evanw.github.io/csg.js/)
- VRControls and VREffect provided by the MozVR project to enable WebVR (http://mozvr.com/)

The project also runs a server built on top of Node.js (https://nodejs.org/), using Express (http://expressjs.com/) and socket.io (http://socket.io/). It also uses a postgreSQL database.

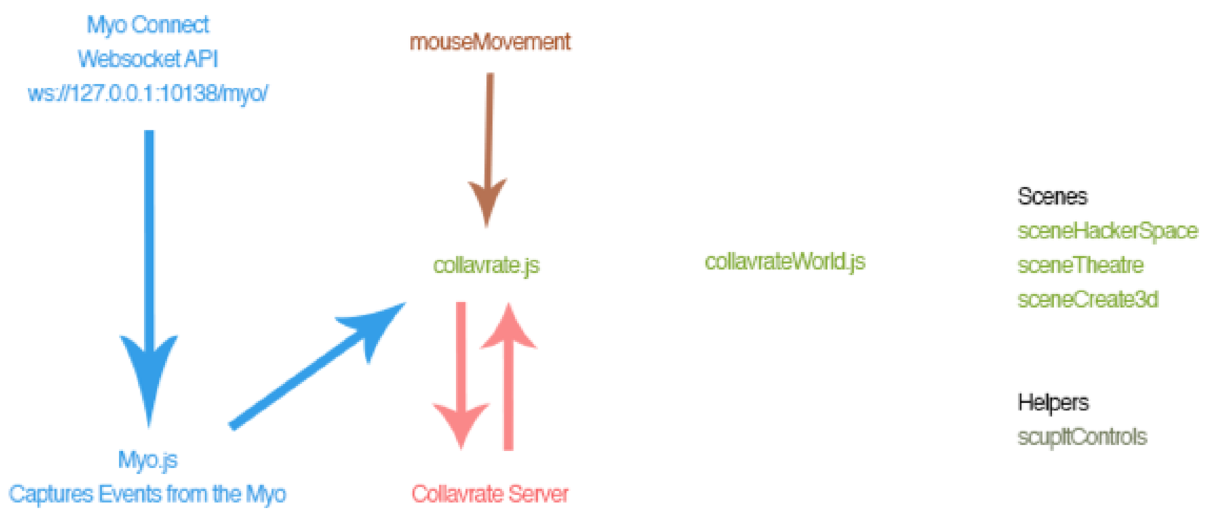


figure 1. Application flow

The overall structure is described in *figure 1*. First of all, *collavrate.js* is there to communicate between the server, myo or the mouse. It is the one that keeps all the data for each client and sends data to the collavrate server which is running socket.io. It also handles drawing lines and other actions.

Secondly, *collavrateWorld.js* is responsible for most of the drawing. It contains a sceneManager which allows us to modularly add more scenes into the application. It also

contains helper functions such as *addObjects*, which allows us to add objects into particular scenes. It also handles collisions, and contains the render loop.
The three scenes that are in the application right now have their own file. Note, that each scene has an index, named *thisIndex*, which allows the sceneManager to determine the order in which the scenes should be shown to the user. The scenes are toggled when the user approaches a marker, and presses the *o* key on their keyboard. Note that each scene has its own camera and its own scene, which are accessed by the following:

```
sceneManager[thisIndex].camera
sceneManager[thisIndex].scene
```

Lastly, in order to take advantage of multiple tools, we developed a toolbelt that the user can toggle through using the Myo or the keyboard. This allows the user to select a certain action before performing it. The toolbelt can be added to a scene using the following:

```
toolbelt.addTools(sceneIndex, x, y, z)
```

There are also methods built to help us rotate the toolbar the right amount. `toolbelt.startRotate(direction)`, where *direction* is the string "left" or "right", initializes a rotation. In order to rotate, the render loop just needs to contain `toolbelt.rotate(speed)`, where *speed* is the acceleration of the rotation. By default, the rotation stops when the next item in the toolbelt is in focus.

We can also listen for changes in this toolbelt, but registering for the 'change' event like so:

```
toolbelt.on('change', function(name) { … });
```

Using the WebVR API

The VRControls and VREffect use the WebVR API to get data from the Oculus Rift. In particular, VRControls maps the position of the Rift headset to the camera, and VREffect enables stereo rendering. In our application, all we need to do is the following:

```
// Apply VR headset positional data to camera.
var controls = new THREE.VRControls( camera );
// Apply VR stereo rendering to renderer
var effect = new THREE.VREffect( renderer );
effect.setSize( window.innerWidth, window.innerHeight );

// in the render loop
// Update VR headset position and apply to camera.
controls.update();
// Render the scene through the VREffect
effect.render( scene, camera );
```

Constructive Solid Geometry

Constructive Solid Geometry (CSG) utilizes boolean operations like intersection, union, and subtraction to combine 3D solids. csg.js by Evan Wallace uses BSP trees to implement the CSG operations on meshes [1].

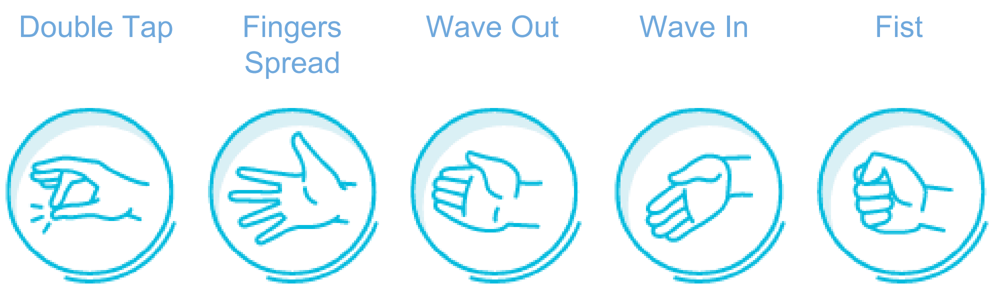
[1] - http://evanw.github.io/csg.js/

Using the Myo Armbands

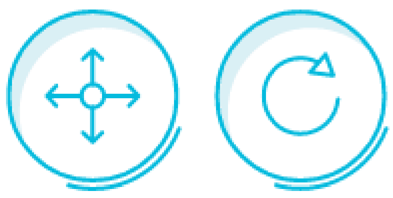
How It Works

In an effort to allow users to interact in virtual environments in an immersive way, Collavrate incorporates Myos, which are gesture control armbands designed by Thalmic Labs. These wearable devices afford the users to engage in collaborative virtual worlds more intuitively than generic input devices, such as the mouse, keyboard, or gamepad accessories.

Myo armbands obtain gesture input via eight medical grade stainless steel electromyography sensors. These sensors detect gestures being made by acquiring muscle signals from electrode to skin contact, amplifying and processing this signal, and finally returning a measure of muscle electrical activity. Out of the box, the Myo armbands support five distinct poses. Theses are:



In addition to the proprietary EMG sensors, Myo armbands incorporate a highly sensitive nine-axis Inertial Measurement Unit (IMU), which is comprised of a three-axis gyroscope, three-axis accelerometer, and three-axis magnetometer. The nine-axis IMU allows for a non-camera based and more seamless method of object tracking in Collavrate VR work spaces.



Connection

The Myo armband transmits gesture and positional data over a Bluetooth Smart connection to a clients machine to communicate.

To integrate the Myo, Collavrate incorporates the Myo WebSocket API to have access to the armbands. The Myo WebSocket server listens for incoming connections on a specific port at 127.0.0.1. A WebSocket URL is used to connect to the Myo WebSocket server in the following manner:

```
ws://127.0.0.1:$PORT/myo/$VERSION?appid=$APP_ID
```

where $PORT is the port the server is listening on, $VERSION is the API version, and $APP_ID is the application identifier for the client. As of April 2014, the server listens on port 10138, and the current API version is 3. Collavrate currently omits an app_id, and thus, the current websocket URL is as follows:

```
ws://127.0.0.1:10138/myo/3
```

Messages sent across the WebSocket connection are encoded as JSON. A paired event is sent whenever a Myo is paired with the server. The WebSocket connection handles communication with the Myo through a variety of events, some of which include, but are not limited to:

- Connection / Disconnection Events
- Arm Synced / Unsynced Events
- Orientation Event
- Pose Event
- EMG Events
- Commands (e.g. Vibrate, Set Locking Policy, Unlock...)

In Collavrate, a hand/myo manager prototype is used to initiate and store existing connected Myos through the socket. Collavrate adds Myos to the manager provided the myo object, it's ID, and an associated second Myo, for when the client wishes to use 2 myos one on each hand, and renders it to the world.

```
function handManager(socket) {
    this.hands = {};
    this.socket = socket;
}

handManager.prototype.addHand = function(myoId, myo, secondMyo) {
    this.hands[myoId] = {
       id: myoId, myo: myo, secondMyo: secondMyo,
       // ... < Further myo properties > ...
    };
    this.renderOnScene(myoId, true);
};
```

When the Myo syncs, Collavrate requires the Myo to remain permanently unlocked for the duration of use within the application. Using the commands previously mentioned, the myo is unlocked, the locking policy is set, and a flag is set in the manager.

```
this.hands[myoId].myo.on('arm_synced', function(){
    console.log("synced");
    var myo_manager = window.myoManager.hands[window.uuid];
    myo_manager.myo.unlock();
    myo_manager.myo.setLockingPolicy("none");
    myo_manager.unlocked = true;
});
```

To recognise gestures made by the user, event listeners are used to execute code on recognition of any of the poses we specify. For example, firing code when recognising a specific pose on the user's primary Myo armband:

```
this.hands[myoId].myo.on($HAND_GESTURE, function(edge) {
    window.myoManager.hands[myoId].myo.timer(edge, $HOLD_TIME, function(){
       // Execute code here
    });
});
```

where, `$HAND_GESTURE` is the pose we are listening for (e.g. 'wave_in', 'wave_out', 'fingers_spread', 'fist', 'rest' etc.), and `$HOLD_TIME` is the amount of time we require the user to hold the pose in milliseconds.

Please refer to the Myo WebSocket API documentation for further understanding of how these events are implemented.

Creating the Virtual World on the Web

Using Three.js

Three.js is a lightweight library that enables developers to create 3D content on the web using WebGL. This library includes numerous APIs such as scenes, cameras, lights, materials, shaders, objects, and other content. Created by Ricardo Cabello, it has been open source since its initiation in 2010 [1]. The Three.js library can be simply included on any web application by including the minified javascript file.

To begin, a simple Three.js scene needs three things: A scene, a camera, and a renderer. This can be as simple as the following:

```
var scene = new THREE.Scene();
var camera = new THREE.PrespectiveCamera(75, window.innerWidth/window.innerHeight,
0.1, 1000);

var renderer = new THREE.WebGLRenderer({ antialias: true });
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);
```

There are a few different cameras in Three.js. In this example, we use a *PerspectiveCamera*. The first parameter we pass in is the field of view. The second parameter is the aspect ratio. The third and fourth parameters represent the near and far clipping plane. This means that objects that are further away from the camera than the value of the far, and similarly, the objects that are closer than the near value will not be rendered.

The renderer is where Three.js does its magic. It is where Three.js renders the scene using WebGL. However, in order to actually render the scene, we must call the *render* method. This is typically done in the render loop:

```
function render() {
        requestAnimationFrame(render);
        renderer.render(scene, camera);
}
render();
```

The *requestAnimationFrame* method tells the browser that we wish to perform an animation and requests that the browser call the function that is passed in as a parameter before the next repaint (typically 60 frames per second) [2]. In Collavrate, we use this technique to update the scene and perform animations.

[1] - https://github.com/mrdoob/three.js/
[2] - https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame

Adding the objects in the scene

Adding objects to a scene is made trivial by Three.js. Primitive objects like cubes, cylinders, etc, can be added directly using the API. For example, in order to create a cube, we can use *BoxGeometry*:

```
var geometry = new THREE.BoxGeometry(5, 5, 5);
var material = new THREE.MeshBasicMaterial( {color: 0xff0000} );
var cube = new THREE.Mesh(geometry, material);
scene.add(cube);
```

The important thing to note in this snippet of code is there are three things required to construct an object. We need a *geometry*, which defines the vertices and faces, and we also need the *material*, which defines the color to be applied to the object. The last thing we need is the *Mesh*. A *Mesh* is simply an object that takes the geometry and applies the material. To add this new object to the scene, we can call the *scene.add()* method, and pass in the cube as a parameter.

In Collavrate, we make use of a built-in loader to load .obj files. In particular, we use the *OBJMTLLoader*, to load the obj file with its respective mtl file. This allows to create more complicated objects such as the whiteboard, tables, and lockers. The obj file defines the

3D geometry, and the mtl file defines the material. These models can be created by using 3D graphics and animation software such as Blender or Maya.

Adding Collaboration

Once we have the scene set up, the first aim we had was to enable users to draw on the whiteboard. To do this, we implemented the ability for the camera to move, so we could mimic the effect of walking. To do this, we leveraged FirstPersonControls as developed by the Three.js team [1]. Because we wanted the ability for the user to walk with a gesture using the Myo, the camera moved forward whenever the *fingers spread* gesture was held for 2 seconds:

```
this.hands[myoId].myo.on('fingers_spread', function(edge){
        window.myoManager.hands[myoId].myo.timer(edge, 2000, function(){
                camControls.autoForward = true;
        });
});
```

We then needed a way to figure out collisions so our camera would not go through objects and walls. To do this, we used ray casting. We first initialized multiple arrays that originated at the camera:

```
var rays = [
        new THREE.Vector3(pos.x, pos.y - 30, pos.z),
        new THREE.Vector3(pos.x, pos.y, pos.z),
        new THREE.Vector3(pos.x, pos.y + 20, pos.z),
        ...
];
```

We also define the direction in which the ray should point towards (in our case, away from the camera):

```
var plocal = new THREE.Vector3(0, 0, -1);
var pWorld = plocal.applyMatrix4( camera.matrixWorld );
var vec = pWorld.sub(camera.position).normalize();
```

Then we can cast the ray using the RayCaster built in Three.js.

```
var ray = new THREE.Raycaster(rays[r], vec);
var intersects = ray.intersectObjects( sceneManager[sceneIndex].sceneObjects, true
);
```

Note how we can grab the intersections to objects by using the intersectObjects method and passing in the objects we wish to check for intersection. The second parameter indicates whether we want to check recursively (i.e. all of the objects descendants).

Once we knew that we were at the whiteboard, we could enable the ability for the user to draw. Since each user is given a cursor in the form of the cube, we toggle the visibility of the cube, and place it in front of the whiteboard.

Then, as we get positional data from the Myo, we can move the cube accordingly. Once, the user holds a fist, we allow them to draw lines. To create dynamic lines using Three.js, we initialize a line with a large amount of vertices:

```
var geometry, line, lineMaterial,
     MAX_LINE_POINTS = 100000;

lineMaterial = new THREE.MeshBasicMaterial({
       color: 0x000
});

geometry = new THREE.Geometry();
for (i=0; i < MAX_LINE_POINTS; i++){
       geometry.vertices.push(new THREE.Vector3(0, 0, 0));
}

line = new THREE.Line(geometry, lineMaterial);
line.geometry.dynamic = true;
```

The important thing to note here is that we set the dynamic property on the line's geometry to true, thus allowing us to change the vertice everytime the user adds onto the line:

```
// shift the array
line.geometry.vertices.push(line.geometry.vertices.shift());
// add the point to the end of the array
line.geometry.vertices[100000-1] = new THREE.Vector3(newPoint);
line.geometry.verticesNeedUpdate = true;
```

Note, that we also implemented movement and drawing by using the mouse instead.

Lastly, in order to actually communicate to multiple clients, Collavrate leverages a websocket server built on Node.js and using socket.io. Once a client connects, a UUID is generated and sent back to the client as well as to all of the other clients. Then, as a line is drawn, the action is sent across to all of the other clients. This is done in two parts as mentioned previously.

To save the state of the whiteboard, all of the line information is stored in a postgreSQL database. When a new client connects, this data is sent to that client so that it could be drawn on the whiteboard.

[1] -
https://github.com/mrdoob/three.js/blob/master/examples/js/controls/FirstPersonControls.js

Adding Web Content

One of the ideas we explored in this project was the ability to add web content in virtual reality. A few ideas popped up, such as displaying pictures from websites in the form of an art gallery, or creating a twitter sphere, where users can see tweets all around them.

However, the idea we wanted to go with was displaying a video in the form of a theatre. We achieved this by mapping a video on a HTML5 canvas element to a Three.js PlaneGeometry.

Creating 3D models in Virtual Reality

The last scene in Collavrate involves 3D models. Using the same techniques as drawing as the whiteboard, the user is able to draw anywhere in the world. As soon as the user finishes drawing a line, a shape is constructed from the vertices.

After this, the user has the ability to extrude the shape so it becomes 3-dimensional. This is done by using the Myo by first selecting the extrude tool in the toolbelt, and then clenching a fist while pulling down. The distance the user pulls down directly correlates to the amount of extrusion. Once a 3D model has been constructed, the user can rotate the object by waving their hand in or out.

The toolbelt is also used to perform other operations on the 3D model. These include moving the object from one location to another, and performing basic mathematical operations such as intersection, subtraction, and union between two objects.

It is also possible to export the OBJ and STL files so that they could be open in other 3D graphic softwares like Blender or Maya. This can also be used to directly 3D print the created objects.

Alternatives to the Myo

Although the gestures that came with the Myo allowed the integration of interesting features such as the toolbelt and movement control, there were significant problems with positional tracking. It is evident that for the purposes of this project, devices that would allow more accurate positional tracking would improve the experience significantly.

There are a few alternatives out there that would allow for better positional tracking. One such alternative would be to use the Razer Hydra or the STEM system, both developed by Sixense. Both of these alternatives provide better positional tracking, however, they feel much more like game controllers that need to be held in the hand itself. This could deter away from the experience that the Myo excels in.

Another alternative, although much more complicated to integrate, would be the Microsoft Kinect. The Kinect would provide joint tracking which could be used not only for drawing purposes, but also for movement.

A simple alternative that is provided in Collavrate is the use of the mouse and keyboard as the input devices.

Future Development Plans

There are many steps that can be taken to improve the experience that Collavrate aims to deliver. First of all, a much more interactive experience could be provided if each client in the VR world had an avatar. This would allow others to see exactly where the other clients are positioned at all times.

Secondly, integrating other input devices that would allow further precision in terms of positional tracking would help the user draw more accurately. If the input device could replicate the control that a mouse has, a user could be able to replicate some basic functionalities of a primitive 3D graphics software.

Furthermore, the addition of more tools, such as primitive shapes, or the ability to remove or reverse a particular drawing would also help replicate basic functionalities of other 3D graphics software.

Lastly, there are a lot of optimizations that could be done in order to improve the experience on the Web. Since WebVR is very young, the MozVR team is constantly improving and modifying the WebVR API. It is essential to keep up with this API in order to deliver the best VR experience on the Web. Moreover, three.js is also a library that is constantly changing and maturing. This leads to many changes that break the existing codebase but make significant improvements in the performance of the entire application.