Apex Prototype

Documentation

by

Tommy Fang

7/06/15 - 8/21/15

Transformational Security

Internship

Table of Contents

1. Introduction

I began this internship with some knowledge of web development that I gained by having a hobby of developing applications with various technologies. I was unfamiliar with all of the frameworks that were used in this prototype, however by the end of this internship, I felt that I had mastered many of the frameworks I had just been introduced to about 2 months or so ago. The purpose of this document is to provide the information necessary to pick up where I had left of and to prevent the next programmer from making the same mistakes I made.

The prototype utilizes HTML5/CSS and Javascript to provide a sleek interface that processes large amounts of data.

First, a server is set up using Node.js/Express.js, this allows us to send data from the server to client in order to decrease the workload of the client. In this prototype, two CSV files are sent(a large T2 Signal data file and a small floor plan CSV file). The files are sent using an http put request from the server and received using an http get on the client side.

These files are utilized through the frameworks optimized by data analysis experts (Crossfilter.js and D3.js). The file gets parsed by D3.js and is converted into a CSV file object. This is an array of CSV objects and each index represents a line of the file and the first line of the file are used as the indices of these objects. The floor CSV file is loaded and rendered by three.js to the THREE.scene which holds all of the visible objects. We can view the scene using a perspective camera and move around using orbit controls. see OrbitControls.js for details.

Crossfilter takes this array and processes it so it can be usable in the data charts and Three.js visualizations. Once Crossfilter returns the filtered array that needs to be visualized, this array is sent to main.js, where the data pump resides. Currently, the data pump is triggered by the

“L” key and renders the respective signals as circles in the current filtered array. The circles are drawn using the signal data. TLW, Frequency, and Bandwidth can be toggled to define the color. The size of the circle is defined by the Amplitude. These values are scaled using D3.js and an appropriate value is output depending on the range of the min and max values of these variables. The filtered array is organized by timecode and the “L” key will cause pump to start visualizing the first time code until it reaches the last time code. It will start over once it reaches the end. Code needs to be added to allow play, pause, and reset functionalities. For now, the user has to reset the page to select a new selection.

---

2. Technologies and Frameworks

2.a. Languages

2.a.i. CSS/HTML5

index.html is the root of this prototype and it is heavily modified by charts.css. Much work still needs to be done to improve the overall design of the prototype, but the foundation of where everything should be has been laid out.

2.a.ii. Javascript

There are many javascript files used to create this prototype. I placed the frameworks I used in “public/vendors/” and all of the code I wrote has been placed in “public/resources/”.

Folder Structure:

Public/

- Index.html

Public/css

- Charts.css, Three.css

Public/Vendors

- Crossfilter.js, D3.js, Detector.js, jQuery.js, Tween.js, Three.js

- OrbitControls.js, RequestAnimationFrame.js, Stats.js,

Public/Resources

- Animations.js, Charts.js, Data.js, Controller.js, Main.js, Scene.js

I will explain how I used the most important files in Section 3: Source Code.

2.b. Frameworks

2.b.i. Three.js, Tween.js

Three.js is a powerful visualization Javascript 3D library created by Ricardo Cabello. It utilizes WebGL to render complex shapes. You can easily draw various geometric and custom shapes without any prior knowledge of WebGL through this simplified API. There are many examples readily available. The github has many utilities that were developed for open source usage, please utilize this repository if you run into trouble.

Tween.js is a tweening engine used to easily create tweens for three.js There are two APIs with the name “tween.js”, the one that I used can be found here.

2.b.ii. D3.js, Crossfilter.js

2.b.ii D3.js

D3.js allows us to select and bind data to DOM elements with very simple commands. Very elegant and efficient algorithms were developed by Mike Bostock to rapidly process thousands and thousands lines of data, which was necessary for this project.

2.b.ii Crossfilter.js

Crossfilter.js depends on D3.js and uses the data binding commands to combine data to create a filter that is able to take in multiple parameters. The current filter is able to filter signal data based on TLW, Amplitude, Bandwidth, Timecode, and Frequency. The charts render the data live as well. It is extremely efficient and easily processes the 11000 lines found in the test signal data. This API has so much potential and I was amazed at how powerful it was in conjunction with D3.js.

2.b.ii. Node.js, Express.js, jQuery.js

2.b.iii Node.js

Node.js is used to set up the backbone of this prototype. Sorry if the code looks a bit fishy or dicey. I had the most difficulties trying to learn about the server-client relationship. I'll try my best to explain my thought process behind setting up the server. This code was generated using the Intellij Idea 14.14 template. It provides routes, which are used when a page is loaded. It takes in a user request on load. This request can be a variety of actions such as file uploads/downloads and sending data back and forth. Views can be defined, they are an alternate form of page displays. In this prototype, I do not use views, I just serve a static html index, but it can easily be

modified to use views, so if multiple pages need to be set up, we can easily do so. In order to properly use a router and a view, the link must be set up within "Spiral/views/Template/header.ejs". The extension ".ejs" refers to "embedded javascript", it is a form of HTML, in which we can embed javascript within the tags without <script>.

Example of embedded Javascript:

```
<p>Welcome to <%= title %></p>
```

```
In this example, title is a string variable, it gets printed with the <p> tags
```

The key files are "Spiral/routes/config.js", where the csv file gets loaded in and "Spiral/bin/www", where the server gets created and setup on localhost:portnumber where portnumber is currently equal to 1337. The current code includes other routes and views such as users.js and about.js. These should be used as a template to show you how to correctly direct a page request to another page.

2.b.iii Express.js

Express.js is used as middleware for node.js in order to connect the components of this application. Usage of express can be found in "Spiral/app.js". In this file, express.js connects the application to the views, directories, cookie parser, body parser, and routes.

2.b.iii jQuery.js

jQuery.js is a feature rich library with a multitude of uses. In this project, it is used for getting AJAX requests to the server and receiving the data as a response.

---

3. Source Code

3.a. main.js, data.js

3.a. main.js

public/resources/main.js contains the data pump, which tells the page to retrieve signal data from charts.js and visualize it using three.js.

On document ready:

```
$(document).ready(function () {
    // initialization
    init(); see scene.js
    // animation loop
    animate();});
```

Animate: requestAnimationFrame was created by Paul Irish to optimize the animation loop. It prevents the page from animation while we are in a different tab or when we're not on the page.

```
function animate() {
    requestAnimationFrame(animate);
    render();
    update();
Render  is called 60 times a frame. It is the most memory consuming
function in this object. Look to prevent unnecessary render calls in
order to best optimize this program.
```

Variables:

```
currentTimeIndex = 0, This keeps track of the current time selection
Floors = [], dataset = [],These are used to hold the data we receive
Loading = true, Loading is used as a flag variable
SignalDictionary = {},
SignalDictionary holds all the objects are currently being visualized.
selected = []; The current filtered selection updates as a global,
So we constantly know what the filtered selection is.
```

3.a. data.js

public/resources/data.js contains the function that uses an AJAX put request to update the data that is sent from the server.

```
function LoadCSV(dataset, callback) {
 var result = [];
  $.ajax({
     type: 'PUT', define the request type
      contentType: 'text/csv',
      url: 'http://localhost:1337/config',
url: string containing the url to which  the request is sent
    success: function (CSV_ARRAY) {
We can only manipulate the data within this asynchronous success function
        if (dataset != null) {
```

```
            result[0] = CSV_ARRAY[0];
              result[1] = CSV_ARRAY[1];
Store the result using a callback function
Result is sent to charts.js (crossfilter/d3.js) and scene.js(three.js) upon
load
            }
            callback(result);
        }
    });
}
```

A TCode helper function was also written and is still within the code, but I found out that crossfilter handles the same functionality easily. I left it in data.js in case it needs to be used later on.

3.b. scene.js, controller.js, animations.js

3.b. scene.js

public/resources/scene.js handles all of the visualization that the crossfilter will feed it.

Variables: see code for more details on parameters.

```
container: refers to the div element that holds all of the rendering we will do
scene: the background /3D plane, it holds the three.js objects that we add.
camera: A perspective camera that allows us to freely view the scene.
renderer: A detector class is used to see if the user's browser has webGL
enabled, otherwise this becomes a THREE.CanvasRenderer.
controls: THREE.OrbitControls are provided by the Three.js team. It allows to
move around scene using the mouse.
stats: used for development purposes, shows framerate and lag time.
Animator: instance of AnimationHandler, see animations.js.
FloorData: holds the array of CSV objects representing each line of floor data.
CrossFilter: instance of chart class, used to filter signal data. see charts.js
RawSignalData; holds csv signal data that gets parsed by d3 server side.
```

Functions:

```
function LoadData() {
   //Grab the data from the ajax call started in data.js
   LoadCSV(dataset, function (result) {
       This asynchronous anonymous callback function stores the data into
results.
       FloorData = result[0];
       RawSignalData = result[1];
       if (Loading) {
           LoadFloors(FloorData, 1);
           Draw the floors using the floor CSV.
           second parameter is number of floors.
           CrossFilter = new FilterCharts(result[1]);
           Create an instance of the charts class,
           allowing us to access the filters/charts and render them.
       }
       Loading = false;
       We are finished loading data

Signal Data CSV Columns:
TCODE,TIMESTAMP,TXID,FREQ,BW,AMP,DEVCNT,X,Y,Z,TLW
```

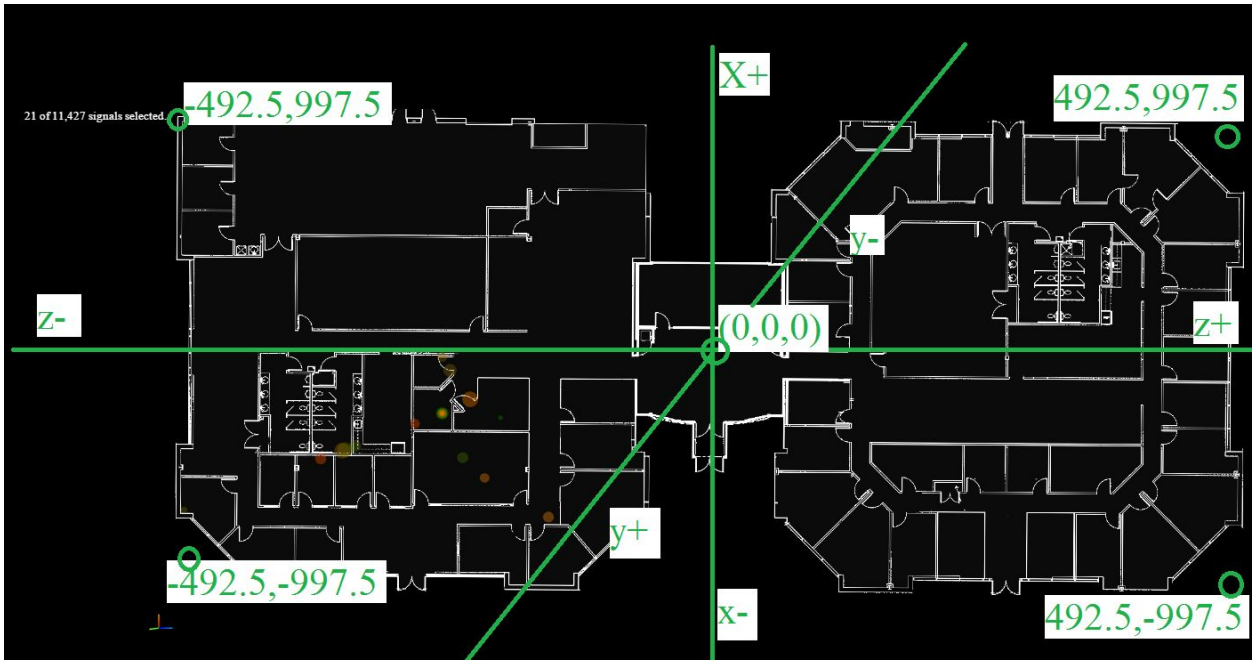
```
Floor Data CSV Columns:
floor_id,name,image,building_name,scale,origin_x,origin_y,altitude,building_off
set_x,building_offset_y,building_offset_z,last_updated,last_updated_by
```

I think the origins are unclear. For example, when looking at the map produced by the original Apex map in ASP.net, I thought the origin of 0,0 began in the top left. More careful work should be done when handling origin in this prototype. I will explain how the Three.js coordinate system works. I think the size of the image should also be included because there is not a built-in method or an easy way used to obtain the dimensions of an image. It is absolutely necessary to have these dimensions when scaling the objects using the origin.

I am unable to completely finish the dynamic origin because of the limited floor data test plans I have and I'm also unsure of how the future program will handle multiple floors, but it's too late for that now. Currently I check if the current floor that's being loaded is the base floor, then that origin as the base. The next floors, if they have different origins, then their positions are calculated using these formulas.

```
FloorMesh.position.x = BaseOrigin_X + Origin_X;
FloorMesh.position.z = BaseOrigin_Y + Origin_Y;
```

The new floor origins are added to the base origin, so that each floor will reference the same origin point, but their positions have moved.



HEIGHT: 1985, WIDTH: 995

In this figure, you can see that the X value actually refers to the 2 dimensional up/down values. Z is left and right. Y refers to altitude in 3D space. The scene sets up the plane in a 3D cartesian coordinate system. The quadrants above show how the points in space are laid out. The scene takes the image which was initially serialized into the CSV as a 64 byte array [using a base64 encoder] and easily deserialized using:

```
function CreateFloor(dataset, FloorNumber, FloorDimensions) {
                image = dataset[FloorNumber].image
                FloorPlan = "" + image,
```

When the floor is created, I also add point light at the center using three.js.

```
var light = new THREE.PointLight(0xffff00, 1, 0);
light.position.set(BaseOrigin_X + Origin_X, FloorMesh.position.y + 250,
BaseOrigin_Y + Origin_Y);
scene.add(light1);
```

This render function sets up the viewport for the screen, because we need to allocate space on the top for the charts and space on the left for the signal table. The render function also constantly updates the new tween animations.

```
function render() {
   TWEEN.update();
   var left   = Math.floor( window.innerWidth  * 0.248 );
```

```
    var bottom = 0;
    var width  = Math.floor( window.innerWidth );
    var height = Math.floor( window.innerHeight );
    renderer.setViewport( left, bottom, width, height );
    renderer.setScissor( left, bottom, width, height );
    renderer.setPixelRatio( window.devicePixelRatio );
    renderer.enableScissorTest ( true );
    //renderer.setClearColor( new THREE.Color().setRGB( 0.5, 0.5, 0.7 ) );
    renderer.render(scene, camera);
}
```

3b. controller.js

public/resources/controller.js contains the event handlers for the page.

```
function onDocumentMouseDown(event) {
    event.preventDefault();
    var offset = $('#ThreeJS').offset();
    var top = offset.top;
    var left =  Math.floor( window.innerWidth  * 0.248 );
    mouse.x = ((event.clientX - left) / renderer.domElement.width )
* 2 - 1;
    mouse.y = -((event.clientY - top) / renderer.domElement.height)
* 2 + 1;
    FindIntersects();
}
```

This is the most important function in this file. It takes in the renderer size and offsets of the inner window to calculate the position in space using mouse coordinates. FindIntersects() constantly uses this event handler on mouse click to determine if an object has collided with the mouse. Offset top refers to the spacing of the Three.js element. In this case, it is about 190px from the top when in fullscreen mode. Offset.left had to be calculated differently because I set the renderer's viewport from the left about `window.innerWidth  * 0.248` pixels. If the renderer's viewport size changes, then this equation needs to be adjusted.

```
function onWindowResize() {
    camera.aspect = window.innerWidth / window.innerHeight;
    camera.updateProjectionMatrix();
    renderer.setSize(window.innerWidth, window.innerHeight);
}
```

This listens to whenever the size of the window changes and adjusts the renderer. It needs to adjust for the signal table data and chart data.

```
function OnKeyDown(event) {
   switch (event.keyCode) {
       case 76: //'1'
           event.preventDefault();
           if (!Loading) {
                  //Slice is used to play a current selection
                   var slice = selected[0].values;
                   var sliceBegin = slice[0].TCODE;
                   var sliceEnd = slice[slice.length-1].TCODE;
                   currentTimeIndex = sliceBegin;
                   var start = window.setInterval(function () {
                       DataPump();
                       //
console.log(OrderedTimeSignals[currentTimeIndex]);
                       currentTimeIndex++;
                       if (currentTimeIndex > sliceEnd) {
                           currentTimeIndex = sliceBegin;// 0;
                       }
                   }, 3000);
           }
           break;
```

This function listens for any key presses. In this case, when the “L” key is pressed, we obtain the current filtered selection. Selected[0] is a selection of signal data, from the CSV, sorted by time code. The initial time code is obtained by using the first index of the selected array. The last timecode is obtained by using the very last index of this array.

The second parameter of the window.setInterval(function(), time(ms)) allows us to control how often this function gets called. Each time the function gets called, it increases the timecode by one and renders the signals of the same timecode until it reaches the end of the slice. Then, it will start over again afterwards.

3b. animations.js

```
function AnimationHandler()
```

This function is defined as a class that handles creating all of the tweens for the signal objects.

charts.js

d. app.js

Mistakes and Future Improvements

e. Obstacles

f. Optimization

Transitioning

g. Testing

h. Next steps

i. Resources