Senior Capstone

GlobeHub: An Interactive News Site

Project Located at: https://github.com/TheWizardOfTime , (@TheWizardOfTime)

Submitted to Adviser: Prof. Steven Janke

By Gianluca Nicolas Paterson

For Senior Thesis Spring 2016 April 8th

Table of Contents:

1. **Introduction**
2. **Specifications, Problems and Planning**
3. **Application Development**
4. **Future of the Project**
5. **Conclusions**

Introduction

The end goal of my senior capstone project was to build a web application that would deliver users with up-to-date, i.e. relevant news about a particular country. The finished product was to be comprehensive in its design and comprehensive for users, by providing a visually stimulating and appropriate context through which news consumption could be associated i.e. you know where your news is coming from. The idea for this capstone project stemmed from a personal inability to fully grasp the association between news and its source. JavaScript, HTML, and CSS languages were used for frontend development and the Node.js Framework, coupled with Express.js and mongodb, for the backend development.

Given my non-existent experience with any of languages or tools mentioned above, I was concerned throughout development on how I would effectively utilize all of these different languages. HTML and CSS seemed relatively straightforward, and JavaScript was described, "in a nutshell, [as] a dynamic scripting language supporting prototype based object construction, and the basic syntax is

intentionally similar to both Java and C++ to reduce the number of new concepts required to learn the language." I formed my understanding gradually about each language, and this made figuring out each individual problem all the more straightforward throughout the development process as I would learn how to deal with JavaScript, application design, and web development.

In the proceeding pages I will provide a detailed and concise explanation of the design, implementation, development of my project, by addressing the main problems I had to solve to achieve the current state of the application. In addition, I will attempt to explain the resources and tools I have used to develop this application work. That being said, I would like to provide a disclaimer of sorts for anyone who is interested. My experience insofar has only touched the tip of the web-development iceberg, so to speak. During development, I was incredibly ignorant of the available methods of producing my application, and therefore I would like to emphasize that the application is more of a proof-of-concept that an application to be used in production. The process through which I discovered the tools I would end up using in the development of my application happened gradually, and even though I have been able to produce something that works for my purposes, I do not claim to have followed any industry standards or practices by any means. It is my hope that this paper will provide a comprehensive reference (for all those interested in pursuing a project such as mine) about what developing an application entails.

Specifications and Problems

In the initial proposal for this project, I had conceived an application where news could be collected and displayed upon specifying a geographic location, i.e. a pair of latitude and longitude coordinates. The interactive portion of the application would involve a user clicking anywhere on a 3D rendered globe to form a query, which would then be used to gather and display links to different articles. Once the

information was rendered, a user would be able to choose the desired reading material. In order to change the query, a user would simply need to click somewhere else on the globe. Additionally, a user would also be allowed to "watch" over particular locations of interest, and quickly get news for a desired location without the need to click on the globe. Additionally, to supplement the news delivery feature, the application would display geographically relevant information and display it as a "location profile", in order to provide users with a context while gathering news. Furthermore, users would also have the option to "pocket" any article links to read for later. So far, all of these specifications have been met; however, there are more features that I would have liked to implement for application.

The exact means of developing the application were unclear during the initial planning phase, and it was not immediately decided that the application would be web based. Nevertheless, a breakdown of the specifications for the application are as follows:

1. 3D Rendered Globe – An interactive UI display, which is a clickable and rotatable sphere, with a comprehensive map, and a function of forming user queries.
2. News Feed – A UI display for the news that is delivered, based on user queries
3. Watch List – A UI display and feature to keep tabs on a particular country
4. Pocket – A UI display and feature to save articles for later consumption
5. Country Profile – A UI feature to provide users with a context for a country in question

Based on the application specifications, there were a few immediate problems I needed to address in order to begin development. These problems were as follows:

1. A way to gather news links and information

2. A way to render a 3D globe on a computer
3. A way to display the globe and the news links to the user
4. A way to implement the functionality of the application

I feel it is important to mention that I did not address each problem one or in any particular order in the beginning at a time, and that I did not have a concrete idea as to how I would put this application together until I was well into production. Nevertheless, these were the general problems I had to address throughout the development of this application. From this point onward, I will refer to these problems only by their numbering.

In regards to (1), it seemed to me that the best way to collect news and information would be by using a web scraping tool. I decided on utilizing a web-scraper because it would save me the trouble of having to register for a ton of API keys for various news websites, and it gave me a more freedom in regards to the type of news I would be able to gather. To begin my web-scrapping experimentation, I used bbc.co.uk (BBC News) as a guinea pig, and scraped the site for news based on country names utilizing the Java HTML parsing library, jsoup. The library provides a very simplistic API for extracting and manipulating data from HTML, using DOM, CSS, and jQuery-like methods and is a great training tool for understanding how HTML is generally structured and utilized on the web.

Once acquainted with the functionality of an HTML parsing tool, it dawned on me how cumbersome it would be to develop and maintain multiple web-scrapers for multiple news sites. Moreover, I had considered developing my own “intelligent” web scraper/crawler of sorts, but I imagined that doing so would be a substantial project of its own and for my purposes I really only needed one way of getting news from a variety of sources without having to spend a lot of time to do so.

Therefore, I turned my focus away from any particular news site, and instead started searching for highly regarded News aggregators and RSS feeds to begin web scraping. Fortunately for me, I found a particular aggregation site/web-service with a well-implemented RESTful design called newsnow.co.uk (NewsNows). Within the NewsNow specifications, its states that “[NewsNow] links to tens of thousands of publications, from top news brands to alternative news sources” and updates its feeds and information continuously. For me, this meant that I only needed to code up one minimalistic web-scraping tool to provide diverse news within my application. More importantly, this meant that it would be maintainable as well.

In conjunction to solving the mystery of which news aggregator(s) to use for developing the application, I was also trying to address essentially (2) and (3), the interactive portion of my website, simultaneously. In the beginning, I was not opposed to the idea of using Java to create a 3D globe; however, I was quite hesitant to dive into coding an interactive globe from scratch, or using any of the Java 2D or 3D libraries, considering the amount of trouble I was already having in finding a suitable way to scrap all of the news I wanted in an efficient way. It was within this turmoil that I discovered a neat web-based 3D rendering library called Three.js (THREE) An open sources project on GitHub, by Ricardo Cabello (@ mrdoob), “The aim of the project is to create a lightweight 3D library with a very low level of complexity — in other words, for dummies.” This description was reassuring, and through more research I was found multiple projects which had been developed using THREE. A lot of man hours had been devoted in order to develop and cultivate the Three.js graphics library. Moreover, most of the math and GPU interactions have been abstracted away to provide a simple and powerful solution to Web based OpenGL. Discovering THREE lead me to firmly decided that I would develop my application for the web, using JavaScript, as well as HTML and CSS, so I dropped any consideration of Java. After researching for quite a while, I

had compiled the necessary information which I would use to build the skeleton for my application, and therefore It was time to move on to (4).

Based on my research, were many ways to go about hosting an application and structuring a web-application/web-site. Popular server-side languages to use include PHP, Ruby, Python, Scala, Java, and conveniently JavaScript. Each language has its own dedicated community and its own benefits, however, I was not interested in trying out each because I just wanted to begin development and not make this project any more complicated than it already seemed it would be, so out of convenience I decided to go with JavaScript and utilize the Node.js web application framework.

To my understanding, the Node.js framework, otherwise called Node, is essentially a server-side implementation of the JavaScript language and provides a means for developers create scalable and event driven web applications. With a few lines of JavaScript, you can implement your own server. To facilitate a smoother development process and for the serve side of applications, people in the Node community have worked together over the years to develop a package-manager, called Node Package Manager (NPM) alongside Node. With NPM, developers can publish their Node modules (essentially the equivalent of a Java Classes) to be used in the development of any number of web-applications. In the industry, Node applications are built around utilizing Node modules, and are essentially structured through a Node module dependency tree.

Utilizing a node Module is as easy as follows:

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 1337;

http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World\n');
}).listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

The require('http') statement at the top of the code is essentially how this feature of node works. The more modules you require inside your node application, the larger this dependency tree becomes. With this feature, I have been able to utilize pre-existing modules to work through problems (1) through (4), and have learned how to develop my own application specific modules for my application server. In a nutshell, the NPM and Node.js paradigm has helped in the development of my application, however, it has also caused quite a few issues. I will share my thoughts on this paradigm in the Conclusions and Discussion section of my paper.

Application Development

To begin development, it was necessary to format my project in the fashion of a Node.js application. Fortunately, this was taken care of using the Webstorm IDE (Jetbrains ©), a web-development IDE that automatically structures application projects according to a developer's desired frameworks. I simply obtained a student subscription to Jetbrains, downloaded the Webstorm IDE, and had the IDE generate a Node.js application project boilerplate. From here on, it was a matter of reading up on the Node.js documentation and going through Node.js tutorials to utilize the framework in practice. In the rest of this section, I will describe the development of my application by defining the project in terms of the specifications and the

problems in more general terms. These terms are as follows and will be referenced by their numbering:

1. The Web Scraping
2. The Globe
3. User Interface (UI)
4. The Web-Server
5. Database

Web Scraping

Beginning with (1), I had already experimented using HTML parsing and DOM manipulations tools, so I searched NPM for something similar to jsoup, and stumbled upon cheerio.js. Cheerio.js is a fast, flexible, and lean implementation of core jQuery designed specifically for the server, i.e. it was the perfect module to fit my needs. Coding up the web-scraper involved combining two reliable and robust Node modules, Cheerio.js and Request.js. Request.js was designed as a simple to make HTTP calls, and even supports HTTPS and follows web-site redirects. However, even with these two robust already made libraries ready to fit my needs, I needed someway of passing queries to my scraper without having to think too much about traversing the DOM of an HTML page in any "intelligent" manner. This was accomplished by designing a JSON file to associate each country with a URI of the NN website, and for the sake of simplicity and consistency, I decided to consider only countries included in the United Nations ISO-3316 Country designations. The reasoning behind this decision was that United Nations only provides these codes to countries that are internationally recognized, so therefore the application would only provide news for countries that are internationally recognized. Continuing with the idea of having a non-intelligent web-scraper, I thought it would be worth while

as well to implement a URL validation module. Utilizing the same two modules in my web-scraping module, upon the loading of the application determine whether or not it was possible to load the necessary URL's from NN and Extract the relevant news data.

The Globe

Moving onto (2), while one of the most challenging aspects to work with has been one of the more interesting and engaging parts of this project. Before I really dive into my experience in creating the globe of GlobeHub, I would like preface this section with note of warning that this section is indeed the longest as I will be discussing more than just the implementation of THREE WebGL, but also aspects of Internet and Browser Security and JavaScript as a language.

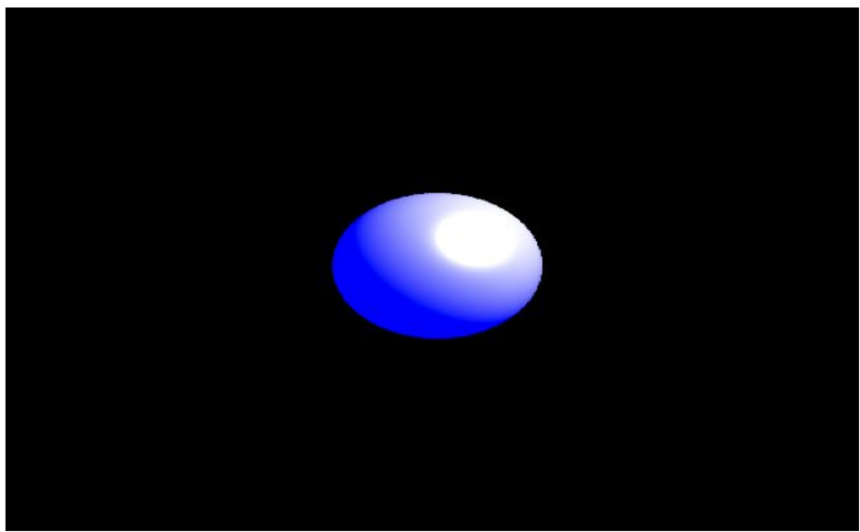
While THREE does abstract away most of the calculations surrounding WebGL, learning to use THREE has also required that understand JavaScript as an interpreted client-side language, and how to properly implement it for use in the browser. Based on some research and scattered readings I came to understand that while JavaScript is a synchronous programming language i.e. code is executed in a predictable way or line by line, much of its use on the web is also on following practices that are scalable and responsive. This relates to dealing with unpredictable events i.e. asynchronous events, like a mouse click or DOM manipulation. In order to match these event-driven interactions, JavaScript supports an asynchronous callback programming model. This is made possible by defining functions in JavaScript as first-class objects. This means that JavaScript allows the passing of functions as arguments to other functions, which can later be executed when there is time in the execution flow of the JavaScript engine in the browser. When trying to base an entire application around WebGL using JavaScript, this knowledge was especially useful when understanding why my own WebGL loaders and JSON loaders would return undefined or lock up my application.

Other issues I experienced during this WebGL process mainly involved Same-Origin-Policy (SOP) and Cross-Origin-Resource-Sharing (CORS) in browser security. These two measures were encountered when trying to deal with writing my WebGL pre-loaders, as well as dealing with (1) and (4). For very legitimate reasons, browsers restrict http requests initiated from within web-scripts. With SOP, this prevents "badguy.com" from being able to access to get access "mycoolsite.com" and do malicious things. However, some sites are not necessarily trying to make malicious requests to other sites, and this is where CORS comes in. To explain further, if "justaregularguy.com" needed to make a request to "mycoolsite.com", then the domains would tell the browser to relax on the SOP, and let them communicate with one another. Basically, all browsers simply restrict script-based network calls to their own domain to make the internet a little safer for everyone involved. Now that I have attempted to address these two issues I will move on with how I made this 3D globe.

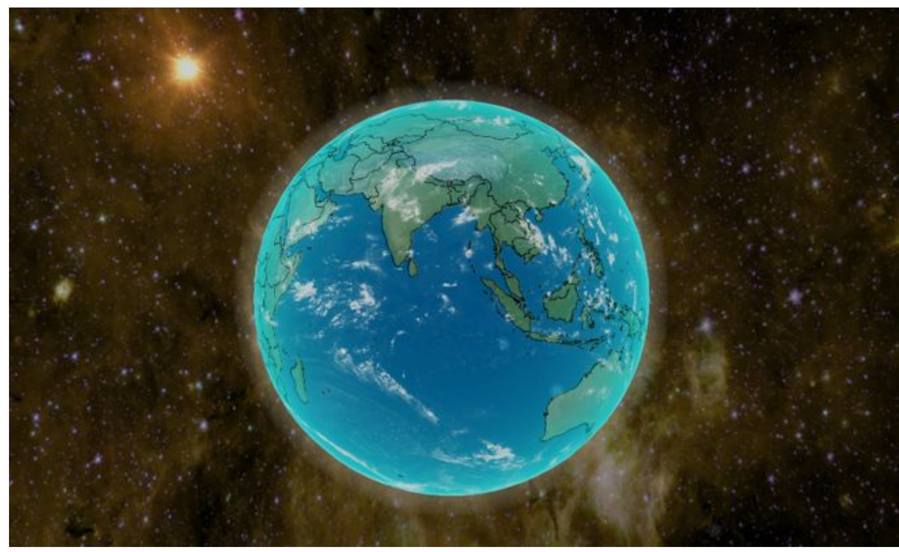
In the beginnings of my making my interactive globe, the code was as simple as writing a few lines of JavaScript, much like the Node.js server-implementation. There is a trend in development to making things as simple to use possible and I am incredibly grateful for all of the brilliant developers out there who make it their mission to provide easy to use simple coding abstraction. With the following lines of code included inside an HTML script tag:

```
<script src="js/three.min.js"></script>
<script>
        var scene = new THREE.Scene();
        var camera = new THREE.PerspectiveCamera( 60, window.innerWidth/window.innerHeight, 0.1, 100000 );
        var renderer = new THREE.WebGLRenderer();
        renderer.setSize( window.innerWidth, window.innerHeight );
        document.body.appendChild( renderer.domElement );

        var geometry = new THREE.SphereGeometry( 1, 64, 64 );
        var material = new THREE.MeshPhongMaterial( {specular: new THREE.Color: 'white' } );
        var sphere = new THREE.Mesh( geometry, material );

        var ambitenLight = new THREE.AmbientLight( 0x0000ff );
        var directionalLight = new THREE.DirectionalLight(0x0000ff, 1.0 );

        scene.add( ambientLight );
        scene.add( directionalLight );
        scene.add( sphere );
        camera.position.z = 3.0;
        var render = function () {
                requestAnimationFrame( render );
                renderer.render(scene, camera);
        };
        render();
</script>
```

If all was implemented correctly, the code above would render the following image, exactly as is, in the browser:



Taking baby steps, and with a ton of research, a lot of trial and error, and a lot of googling asking questions on stack overflow, I learned how to combine a few other JavaScript libraries together and succeeded in changing this simple blue lighted sphere into something like this:



I began the process of learning how to produce the image above by plowing through quite a few of the examples the documentation of THREE. Once I had the basics down, and a grasp of how to utilize the data stored within the various classes, I implemented the functional aspects of the globe for use in the application, while also learning a ton about the powers of WebGL and JavaScript as a language. The main problems I had to solve to make the globe pictured above were as follows, and from this point onwards I will refer to them by their lettering:

a. Globe Interactivity

b. Cartesian Coordinates Conversions

c. Texture Mapping, and Cultural Boundaries

d. Pre-loader for Textures and THREE.Object3D (Models)

e. Optimizing performance (Used and Considered)

Addressing (a) was probably the most straightforward problem that I needed to deal with during the development of this application. On GitHub, there were already quite few open source libraries created by the developers of Three.js to interact with 3D objects in Three.js. There are orbitcontrols (rotate the camera with

mouse clicks and drags), fly controls (move camera around in the scene to any position), drag controls (drag objects around) so by testing out a few of them, I was able to implement client-side camera rotation through mouse clicking and dragging.

Figuring out (b) was solved by using the equations for converting Cartesian Coordinates (x,y,z) to Spherical-Polar Coordinates (r, θ, ϕ). This was accomplished by using one of the classes provided by Three.js, THREE.RayCaster class. The THREE.RayCaster is used almost exclusively in Three.js to create picking rays for interacting with 3D Meshes in Three.js. In fact, the source code for the camera rotation libraries are built using the THREE.RayCaster. Using this class and its built in methods, I was able to project a 3d ray from the mouse, through the camera, into the rendering scene, and then check if that ray had intersected with my globe. If the intersected object was indeed the globe, then I would pass in the (x,y,z) of the point of intersection and convert that into the appropriate (r, θ, ϕ). To check that (b) was working accordingly, I coded up a THREE.Mesh generator to render meshes onto the surface of my globe only at (r, θ, ϕ) defined by both the Prime Meridian and the Equator.

While (b) was effectively tested, I thought it would be prudent to simultaneously address (c). If I wanted to create a convincing model of Earth to act as the globe for my application, I needed to make sure that when texturing (mapping an image to the surface of 3D Objects in Three.js) a mesh with an appropriate image of the earth that my coordinate calculations would line up with the texturing in a convincing way. To texture images, the Developers of Three.js have yet again conveniently provided there own THREE.TextureLoader and THREE.Texture classes. Essentially, you download an image file (.png, .jpg), and call the THREE.TextureLoaders.load() method on the desired image file’s relative path. Once the image is loaded, you can either pass it into a Texture Object pass then pass that into our mapping properties of our desired 3D Objects THREE.Material

object property. You can also pass the loading function directly into a 3D Objects Material object property as well, because JavaScript supports first-class functions. Once the texture property is appropriately defined, the Three.js library takes care of the rest of the rendering and texturing for you.

Upon discovering this functionality of Three.js, I did a little googling and found a large collection of raster map data tailored towards visualizing Earth from space. These were curated by a cartographer named Tom Patterson, and were made for visualizations. In addition to standard texturing, these data maps also came in formats to be used for Specular Mapping (simulating light intensity), and Bump Mapping (simulating bumps and wrinkles i.e. depth). Three.js supports these types of mappings, as well as a few others, when generating meshes, so I utilized these images as well.

Before moving on, I would like to refer back to the beginning of this section, where I made a point about describing a few aspects of modern-day web security. Here is where that knowledge became invaluable in addressing issues I began having while trying to host my application. Up until this point in my application, I had been using a node-module by the name of 'http-server' to test the WebGL. Which enabled me to load my own image files and JSON files with ease. I thought it would be best to start implementing server-side functionality, so I decided to roll my own server using a few of the node-core libraries. Unfortunately for me, configuring my server to deal with CORS was unfruitful, and even by reading source-code and documentation, I was not able to solve this issue. As a temporary workaround, I disabled my browsers web-security, and continued working through the application.

I was able to accurately test the mapping of coordinates to my sphere, and provide an accurate spherical polar coordinate calculations and a great looking set of textures. The globe looked great at this point, but adding a map of country boundaries was going to make it look even better. Unlike mapping a solid image, I

approached this problem by applying a transparent one. This is made simple by utilizing some of the built-in methods for texture mapping that Three.js provides. With Three.js, applying transparency to a texture is as simple as mapping a texture that is missing some coloring values within its pixels, and setting the transparent property of the Material to true. This method was used in order to create the illusion of rotating clouds over a globe, and with this method in mind I began searching for a way to draw my own map to texture on my Earth as well.

Eventually, I stumbled upon a JavaScript library called D3.js a JavaScript library for manipulating documents based on data, developed mainly by Mike Bostock, a renowned computer scientist and data-visualization specialist. With D3.js, manipulating HTML, SVG(Scalable Vector Graphics), and CSS is made easy, by providing numerous methods for setting attributes or styles, manipulating DOM nodes, and changing HTML or text content. To utilize D3.js, I combined its functionality with a JSON format known as GeoJSON. GeoJSON is a format for encoding a variety of geographic data structures. An example of GeoJSON is as follows:

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry": {
        "type": "Point",
       "coordinates": [0, 0]
      },
      "properties": {
        "name": "null island"
      }
    }
  ]
}
```

The JSON is additionally utilized to describe properties associated with a geometry such as in the example above, where the geometry is associate with the name “null island”. The resultant SVG formed from combining D3 and GeoJSON were quite

pleasant; however, loading a GeoJSON can be quite cumbersome, as some of the file sizes for GeoJSON can get very large. In the case of a GeoJSON file describing the geometries for an entire political map of the earth, this was definitely the case.

Coincidentally, Mike Bostock has also addressed this issue and had created his own Extension for GeoJSON known as TopoJSON. TopoJSON accomplishes this by removing redundancies in Geometry definitions. Using D3 and TopoJSON, and following some tutorials for data-visualization in D3, I was able to generate an SVG of the all political-boundaries of as defined by the associated governments of the Earth. To convert the SVG to .png file to be used as a texture, I used a tool called SVGCrowbar, which saves SVG as an SVG file to your computer and then passed the SVG file into some JavaScript which converts an SVG image files into .png.

Once all this was finished, I was able to apply the technique for rendering transparent textures to the .png of the Earth's political-boundaries, and produce texturing on image shown previously. A more in-depth explanation and listing of the tools I used to do all of this is available in the README.txt file located in the “misc” directory of the GlobeHub project folder on GitHub. There was a bit of extra finagling using some other THREE Extension libraries developed by Jerome Etienne, called THREEX, to generate an atmosphere for the earth. These extension libraries are meant to be used unobtrusively with THREE, so there was not much trouble in getting the code to work appropriately.

Now that the globe was for the most part coded up, it was time to start thinking about some of the structural design of my application. When it comes to loading HTML, embedded images or graphics begins rendering as soon as the HTML is loaded. In the case of my application, where WebGL was involved, this was not going to fly. Watching a 3D Earth render is interesting, but it is not very aesthetically pleasing. Therefore, I decided to write my own graphics pre-loader for the application, as well as my own data-preloaded for the JSON. THREE also provides its own THREE.LoadingManager class, to be used with the TextureLoader.

Its function is to provides a means of debugging issues that could be experienced with loading files into THREE. Writing the pre-loaders was a problem of its own, and it was the point where I needed to start thinking about JavaScript in an Asynchronous way. Taking this into consideration and reading plenty of documentation I decided to write the texture preloader first, as it was impossible to render any Object in a meaningful way without loading the images for Texturing first.

To do this I utilized the JavaScript Promise Object. The Promise is used for deferred and asynchronous computations and is as a wrapper placed around functions or code whose computations cannot be measured simply. In order to use a promise, you place a code block inside a Promise, and then place a callback with either the resolve() and reject() callback functions. Essentially, each call to TextureLoader.load() is wrapped in its own Promise. If the loading fails, then the Promise is given a callback to reject() with an appropriate error message, otherwise it is given a callback to resolve() and is passed the loaded Texture. To keep track of all Promises made, I stored every Promise into an Array. Once all Promise have been decided (rejected or resolved), then a call to Promise.all() is made which takes in the iterable Array. If all Promises made have been resolved, then a callback function is passed an Object containing all of the loaded Textures necessary for the application.

Once I figured out how to write one preloader, writing the was more manageable, as I had come to understand the use of callback functions more effectively and efficiently. I thought it would be interesting to try out different methods for preloading, so I implemented each differently for my own learning. I will not dive into the application code or design in this section any more as the commented code provides a clearer explanation. With all of the pre-loading and rendering taken care of It was finally time to try and optimize performance of the WebGL.

The best way, I have found, to optimize WebGL performance involves coding in GLSL. The basic usage of GLSL involves writing two separate programs, called a Vertex Shader and a Fragment Shader. Simply put, a Vertex Shader is code that interacts with a 3D Object vertices i.e. the points in 3D space that are used to form Polygons (Triangles) that define a Geometry. The Fragment Shader computes the desired RGB values within each Polygon and then the GPU renders those on the screen. THREE abstracts all of this away within its THREE.Geometry and THREE.Material classes, which together are combined into a THREE.Mesh to render 3D Objects. I did not really have a lot of time to get into writing my own GLSL code, however this is the best way to guarantee good performance on within the browser. GLSL code directly utilizes a computer GPU, and offloads a lot of the heavy lifting for rendering images made in the browser.

Another method if simplifying browser complications that I looked into was using JavaScript's Webworker Object. A Webworker is essentially a way to thread processes in the browser, using JavaScript. Unfortunately, I did not implement any Webworker into my code due to time constraints, and my lack of understanding how to do so effectively. To address optimization of WebGL, I have instead messed around with the Polygon count and reduced lighting, as these are both easy to change and can seriously slow down performance if used heavily.

User Interface

That last section was pretty long, so I will try to keep this one short. Implementing (4) has been one of the more relaxing aspects of this project. HTML and CSS are a pretty straightforward language to get a grasp on, and why for my purposes there wasn't to much diving into understanding a lot about the language, there is a bit of a learning curve when trying to implement good design into an HTML document, and this is based on conventions of when to use classes and ids

for CCS styling, and how to use as little code as possible to effectively deliver your content. To learn these concepts, and I have not by any means done my research about HTML, I basically looked through a ton of Example Three.js web-applications, notably the The Google Chrome Arms-Globe Experiment for coding style and implementation. The color scheming and CSS ideas for my web application come mainly from the Promotional website for Hollywood motion picture Independence Day: Resurgence (coming to a theater near you this Summer 2016).

To plan out a structure for the UI of the application, I basically decided I would need one node for each feature. These are the WebGL node, and then the applications main specifications i.e. the Pocket Node, the Feed Node, the Watch-List Node and the Profile Node. I divided the page up based on each feature. To keep the user interface minimalistic, there is a single button to expand and collapse each of nodes on the page. Excluding the one for WebGL. Each button, maps to a jQuery function that toggles the CSS positioning of each feature node in and out of the user view window. Within each feature node, a button is present to access the functionality of the feature. Below is a screenshot of the application UI design in full:

Within each of these feature nodes displayed on the screen, content is meant to be dynamically generated based on user input. This input is passed to through a dialog box, and sent to the database on the server to be saved and processed, and is then sent back to the client to be displayed. Currently, the application is able to generate news at the click of a location on the globe, but due to dependency issues and problems involving package binaries for my database implementation using a mongodb ODM called Mongoose.js, I was unable to persist data through login sessions. However, given more time to figure out issues with the binaries, and understanding how to put together a server side implementation of Node.js involving NPM package distributions, I am confident the issues could have been resolved, and hopefully in the future I will take the time to fully implement these features for my own purposes.

Web-Server

As stated in the sections of this paper entitled “Specifications Problems and Planning”, I have been using Node.js as a framework to host a locally run web-server for my application. When configuring and testing the UI and Globe implementations, I still had not configured a server to work correctly with CORS, and instead had been using a node module called “http-server”. The module's purpose is to serve static HTML pages with CORS enabled and nothing more. In order to meet my applications specifications, I would need to find a way to implement CORS and make a server that could do more than just server static content. And I did eventually write my own server implementation, however this involved depending on the functionality of another Node module called Express.js

Express.js is essentially a Node Web-Application framework that provides a robust set of features for building your web applications, and automatically comes with CORS configured. Express.js is an incredibly useful framework for

web-developers interested in creating web-servers that are designed in a RESTful way, as the Express.js API allows developers to route client requests in a simplistic manner. Given that my website was designed to be a SPA, this feature was only used to implement user authenticated login and registration functionality. With CORS finally enabled, it was time to address the issue of connecting my UI and Globe with my server-side web scraping module I had developed for the application.

In any regular circumstance, a web developer is aware of how to use JavaScript appropriately for client-server communications. This sort of JavaScript programing is known as AJAX. AJAX stands for Asynchronous JavaScript and XML and simply put, it is the use of the XMLHttpRequest Object to communicate with server-side scripts. However, considering my luck so far in using Node libraries, I thought it would be worth trying to find a convenient module to abstract all of this away. Fortunately, I happened upon just the appropriate library, called Socket.IO.

Socket.IO enables real-time bidirectional event-based communication between the client and server. What this meant for me was that I did not need to depend on the use of routing pages to implement a server side. To use socket.io is as simple as follows:

In our Node server code

```
io.on('connection', function (socket) {
  socket.emit('news', { hello: 'world' });
  socket.on('my other event', function (data) {
    console.log(data);
  });
});
```

In our HTML

```
<script>
  var socket = io('http://localhost');
  socket.on('news', function (data) {
    console.log(data);
    socket.emit('my other event', { my: 'data' });
  });
</script>
```

Essentially, sending a message between the client and server of an application is as simple as writing two blocks of code like the ones above. Using Socket.IO provides a means of pass data back and forth between the client and server for the entirety of the application, and would be crucial for me in understanding how to implement the Pocket and Watchlist features of my application, as well as the database.

Database

With an understanding of the structure my application, all that was left to do was implement a Database. In theory, this part of the application process should have been fairly straightforward. Unfortunately, it ended up being the most frustrating part of writing this application, which I will explain in the Conclusion of my paper.

Given the amount of hype on the web for using NoSQL databases, I thought it would be worth trying to implement my own. NoSQL is a class of database management systems (DBMS) that do not follow all of the rules of a relational DBMS and cannot use traditional SQL to query data. mongodb in particular is one of these databases with a seemingly highly regard implementation, so I settled on

using it to implement some of the database features for my application, and of course I would use Nodes wide-variety of package distributions to do so.

Mongoose.js is a Node module used for providing a simple Schema based solution for modeling application data for mongodb. To provide an example, consider the following code below:

```
var Article = new Schema({
    body   : String
  , date   : Date
});

var Pocket = new Schema({
    name      : String
  , articles  : [Article]
});

var Post = mongoose.model('Pocket', Pocket);
```

The code above is very similar in its syntax to the how JavaScript defines an Object. I do not have any in-depth experience using mongodb outside of writing this application, however mongoose is apparently an incredibly intuitive solution for a lot of issues experienced by those who write and maintain a mongo database for larger applications. From my understanding, this is because of the non relational way in which mongodb stores its data, and to further explain what this means I will try to provide an example related to my application.

Using the Schema examples above, let's say that you want to embed a basic Pocket with a few Articles. As time goes on and our application starts to grow and gain new features, the definition of these things, our Pocket and Articles, being to slowly evolve with our application. This evolution in data structure, combined with the loosely-typed language of NoSQL, makes it very easy to accidentally save data-types inconsistently within a NoSQL database such as mongodb. Thus, errors that are made will be saved into your database forever, and unbeknownst to you until your application decides to crash one day, and then you are forced to figure

out why, or sacrifice all of your data. Therefore, Mongoose provides a sort of safety net between the application code and your database, by defining a default Schema for the data passed into the database.

Given that this was just a senior capstone project, and not a full blown web-application meant to be scalable (although it could be!), I thought it would be a good learning experience to implement a mongodb database, and it ended up being a learning experience with a little extra experience thrown in for good measure. While trying to implement mongoose into my application, I encountered a few issues with inconsistent dependencies in my applications dependency tree. I was able to solve this issue eventually, which involved me frantically pouring over stackoverflow and mongoose.js documentation until I discovered that it involved inconsistent mongodb binaries for two separate modules within my Node application, which had not been addressed in an update of mongoose. While I did solve the issue, unfortunately for me I was unable to fully implement the database dependent features by the capstone deadline i.e. the Pocket and the Watchlist.

Future of the Project

Given that this project was mostly constructed for educational purposes, I do not believe I will get feedback or have any issues with people using this application for anything other than a reference for creating their own application. That being said, the design and implementation of the application could be greatly improved, and I would like to explain how by again referring back to each main aspect of the development process mentioned in the previous section.

Considering (1) while, I do not think Web Scraping is necessarily a bad way of getting the data I needed for the application, it is not 100% legal. I would try and move away from utilizing web scraping as a means of gather news. That being said, I will try to provide a little bit of background on the legality of Web Scraping (at least

in the US). As long as the web scraping is not disruptive to the host of the content, then the web scraper does not commit a crime as defined in the Computer Fraud and Abuse Act.

Addressing (2), I had discussed a little bit about Optimizing WebGL performance in the previous section which involved GLSL code. Understanding and implementing some GLSL code into the application would be a great learning experience. Since I did not have the time to do so, I would instead like to address a better method for looking up a country's location on the Globe, which involves GLSL code. Based on the methods discussed in a blog post about the programming behind the Arms Globe Experiment, it is possible to approach locating a country by utilizing a lookup table and a grayscale indexed map, and the applying a convenient “country highlighting” graphic by using GLSL code. I did try and reproduce this, and was successful using D3.js and TopoJSON, I reproduced a look up map for each country and output a JSON file containing ISO-3166 country codes and the corresponding look up grey-scale values.

For the sake of convenience, I will address (3)-(5) all at once, since the UI and Database are dependent on each other for providing the service of the application. For (3), there is issue with the way my UI is implemented. I cannot seem to scroll properly i.e. the focus of the mouse on the screen when trying to identify if a DOM element is scrollable does not work. This is a problem in terms of usability. In regards to (4), I would like to remove the dependence of my application on Express.js, because for my purposes I do not need a such a framework, as I only used Express.js as a means of implementing CORS. Finally, for (5), I would like to use a separate implementation of a NoSQL database, as apparently mongodb suffers from misleading documentation, and essentially a poor implementation. As a Database meant to perform well for web applications that are meant to scale, mongodb falls short. This is more accurately explained in the source, but I will provide a concise description about why mongodb is not a reliable database. While

it claims to not have any consistency issues, mongodb allows document reads and writes to see old values of the documents within the transactions. This is generally bad when you are working with generally large sets of data, and is a general concern given that many applications on the web are dependent on mongodb.

Discussion and Conclusions

Despite the issues with the Database, I believe this project has turned out quite well. Since starting, I have gained a wealth of knowledge about the kinds of tools that exist for developing web applications, and for application programming in general. The sheer amount of potential with the tools available for anyone with access to a computer today is astounding. The use of online repositories, such as GitHub, have provided a standard on the web of developing programs that are meant to be useful, well maintained, and open-source. More importantly, communities can actively participate in the development of such programs and discuss ways to make them more comprehensive. These discussion pave the way to optimizing, and standardizing development.

That being said, Node and NPM are indeed an incredibly useful open-source tools that have been developed with a lot of care. They provide a service which facilitates rapid growth and development and open-ended discussion. However, the Node-NPM paradigm, given its relatively recent appearance in the industry and rapid growth in popularity, has not been provided with an environment that is necessary to match the required robustness and comprehensiveness for open-source tools today. Anyone who understands Node, or has read the documentation, are liable to distribute their own modules for use in production. The number of modules already distributed is enormous, and the vast majority of them are underutilized, or essentially useless. There are no restrictions to the kind of modules people are allowed to publish, and more importantly, would-be

developers are not accustomed to properly documenting and maintaining their modules. This gives rise to alarming and immediate issue with NPM

All Node.js applications, which are essentially modules of their own, are dependent on the robustness of each module they are built on top off. If a module is not well maintained or robust, it has the potential to collapse the application from within. More problematic, if someone should remove a module from NPM, then any application built around or using the removed module will crash immediately. This is a serious issue and given the size of dependency trees for applications that run on Node, if even one dependency fails, the application fails. This issue must be addressed by the greater Node community and hopefully a suitable solution will be provided sometime in the near future.

I would like to conclude this paper by thanking my capstone advisor, Prof. Steven Janke, for giving me an opportunity to pursue as free form of project as this. The amount of exposure I have had over the past four months has been incredibly valuable to me, and this will most definitely prove to be an asset for me in the coming years.

Bibliography - (Informal)

Independence Day: Resurgence Website - http://www.warof1996.com/

Google Arms Globe Experiments - http://armsglobe.chromeexperiments.com/

Passport.js, User Authentication for Express.js - http://passportjs.org/

Mongoose.js, the Mongodb ODM - http://mongoosejs.com/

Socket.io Client-server communication - http://socket.io/

Express Web-Server Framework - http://expressjs.com/

Node.js Web Application Framework - https://nodejs.org/en/

Three.js , WebGL - http://threejs.org/

D3.js, Data Driven Documents - https://d3js.org/

GeoJSON - http://geojson.org/

TopoJSON, GeoJSON Extension - https://github.com/mbostock/topojson

SVGCrowbar - http://nytimes.github.io/svg-crowbar