On-the-fly Map Generator for OpenStreetMap Data Using WebGL

A CS 298 Project Report
Presented to
The faculty of the Department of Computer Science
San José State University

In Partial fulfillment
of the Requirements for the Degree
Master of Science (Computer Science)

By

Sreenidhi Pundi Muralidharan

December 2015

The Undersigned Project Committee Approves the Project Titled

On-the-fly Map Generator for Openstreetmap Data Using WebGL

By

Sreenidhi Pundi Muralidharan

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Christopher Pollett, Department of Computer Science Date

Dr. Sami Khuri, Department of Computer Science Date

Dr. Robert Chun, Department of Computer Science Date

APPROVED FOR THE UNIVERSITY

Associate Dean Office of Graduate Studies and Research Date

ABSTRACT

This project describes an approach to create an On-the-fly Map Generator for Openstreetmap Data Using WebGL. The most common methods to generate online maps generate PNG overlay tile images from a wide range of data sources, like GeoJSON, GeoTIFF, PostGIS, CSV, and SQLite, etc., based on the coordinates and zoom-level. This project aims to send vector data for the map to the browser and hence render maps on-the-fly using WebGL. We push all of the vector computation to the GPU. This means that less data needs to be sent to the browser. We have compared existing approaches to our method of generating maps and have been able to show that our method provides a faster and better solution.

ACKNOWLEDGEMENTS

I take this opportunity to thank my mentor, Professor Dr. Chris Pollett for his useful comments and remarks, which guided me while doing this project. Needless to say, I engaged myself in learning a lot through this project. Furthermore, I would like to thank my committee members, Dr. Sami Khuri and Dr. Robert Chun for their feedback, comments, and time. I would also like to thank my friends and family for their continuous encouragement and support. Special thanks to my husband Sriram Krishnan, who is my backbone, who supported me in every single way, channeled my efforts, and made them fruitful.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Openstreetmap (OSM) is a worldwide geographic database created from public domain data sources and user created data. It contains a huge amount of data for a wide variety of features, including administrative boundaries, streets, water bodies, points of interest, and buildings. Such a large dataset, being difficult to store on servers and render as-is, is fragmented into smaller chunks of uniform dimensions, say 256 x 256 pixels. These chunks are referred to as “tiles”, and are bitmap images in essence, which are in turn the building blocks for online maps. These tiles are placed next to each other in online maps to create an impression of a seamless map. The tiles thus generated are stored on a server and are rendered on-demand. Computing tiles on the server during query time is a lot slower; the user has to wait till the tiles are computed, generated and rendered. Hence, the tiles are pre-computed for each zoom-levels and are stored on the server, and rendered on-demand. On the contrary, my project tries to render maps on-the-fly, without using the concept of “tiling”.

To increase the level of detail seen in each of the tiles, these online maps use the concept of “zooming in” and “zooming out”. In order to organize these images in a

bigger invisible grid-like structure on the browser and to create a perfect map, the maps use an x/y/z coordinate system, where “x” and “y” describe the x and y coordinates of the image tile on the grid and “z” describes the zoom level. For example, http://localhost:8080/osm/0/0/0.png might show an image tile of the entire world, placed at a (0,0) coordinate position on the invisible browser-grid at zoom level 0 as shown in figure 1-1. A similar approach as described above is used in http://www.opencyclemap.org/. Hundreds of thousands of tile images are generated and stored for each zoom level, which takes up a huge storage space. My project, on the other hand, draws maps according to the zoom level specified. Thus, each of the points plotted are calculated based on the zoom level.

There are two ways in which tiles could be generated. The first one, as described above, is generating tiles as and when the user requests. If a project running on Geographic Information Systems (GIS) uses its own Web Map Service (WMS) server, like using Mapnik for instance, then generating tiles using the above-described method, loading and rendering the map may be time consuming. Latest maps (for example, Google Maps) use a professional, second way of generating tiles, i.e., “seeding” to cache the downloaded map tiles to reduce the communication between the server and the client.

Though the user experience is greatly enhanced since the user does not have to wait for tiles to be produced, it is a process that can consume lots of disk space. Further, purging such cached tiles may be cumbersome when the map data changes often.

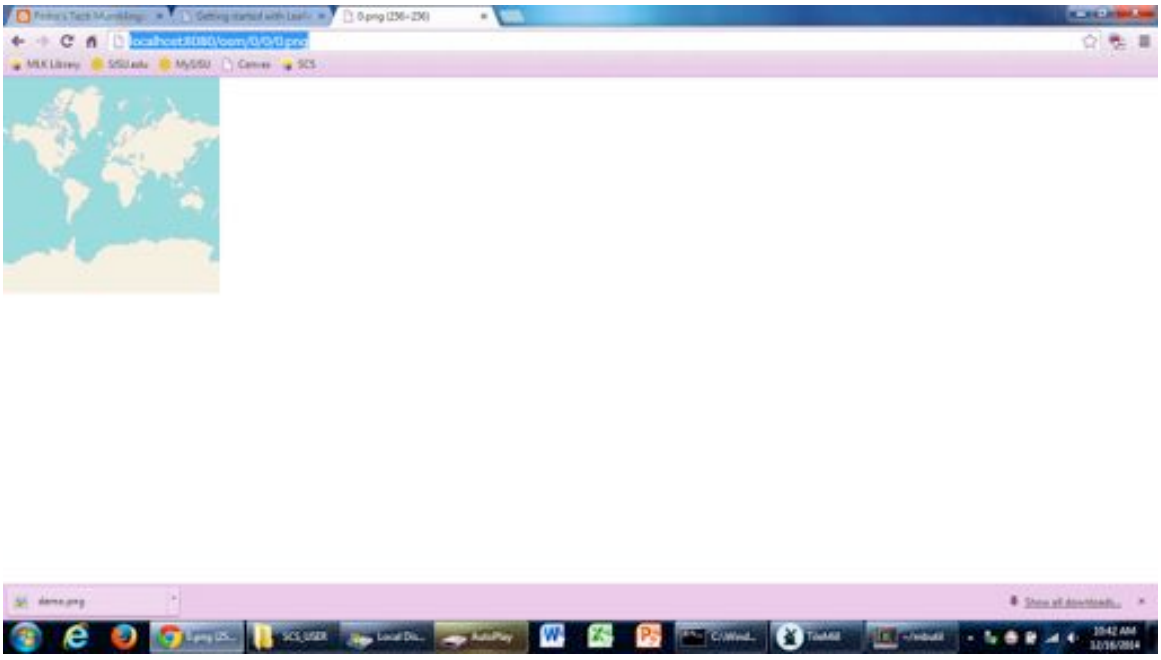

Figure 1-1: OSM data-based tile generated for 0/0/0 x,y,z values*

Through this project, I have come up with a solution of neither generating map tiles according to the users' requests nor caching the pre-generated tiles beforehand. My solution, though might seem simple, is effective and efficient in not consuming too much

* *generated using TileStache, a Python-based server application that can serve map tiles based on rendered geographic data.*

of disk space and not time-consuming either. This is because of not "pre-computing" and "generating" images of any kind. Instead, we generate maps on-the-fly, by pushing all of the vector data computation into the Graphics Processing Unit (GPU). Thus, lesser communication and data transfer happens between the server and the client (browser here). The project consists of a backend PHP script that interacts and pulls data from the database server through an AJAX request and a front-end WebGL-based rendering engine. By conducting experiments, I have been able to verify that my project provides better solution to the existing demands.

The rest of this work is organized in the following way. In the chapters following the introduction, I have discussed about my deliverables, which involves rendering the OSM data on top of a WebGL canvas; wherein I imported the OSM data into a Postgres database using osm2pgsql and wrote a postgres query for checking if a specific highway passes through a given polygon (bounding box) – in my CS 297. Continuing with my CS 298, I have tried to modify the query further to display polygons (buildings and other geometry) in the map along with lines (roads, streets and trails). I have experimented by trying to expand the bounding box according to the user's current location, resize the map drawn and other zooming functions.

2. BACKGROUND

2.1 Openstreetmap Data

The Openstreetmap is an open-source, project that collects huge amounts of raw geographic information and stores it in a specific format. It is crowd sourced as well, meaning that it collects the data from volunteers and makes it available for everyone to use for free, under Open Database License (ODL). Volunteers performing a ground survey using a handheld device such as a GPS, a camera or a voice recorder collected the initial OSM data from scratch. Recent data collection strategies include aerial photography and other government and commercial sources. This has greatly sped up the process of data-collection and has improved accuracy. This data collected have been used by many mapping applications such as MapQuest, Geocaching, etc. Figure 2-1 shows how an Openstreetmap looks.

2.1.1 OSM Data Format

The OSM data uses a topological structure, with four main elements. Though these might vary in their names with each OSM data source, these four forms the heart of OSM data and are called the “data primitives”.

Figure 2-1: A map rendering OSM data

2.1.1.1 Tags

Tags are a way to represent nodes and ways as key-value pairs. They store meta-data about these objects such as their name, type and properties. Examples of tags include:

- Highway = residential
- Exit to = Durham Fy
- Cycle way = lane
- Foot = no and so on.

A tag in an OSM XML file could be written like this:

<tag k="highway" v="motorway_junction"/>

2.1.1.2 Nodes

Nodes are geographic points with a latitude and longitude reference. These latitude and longitude points are represented as coordinated points, such as (latitude, longitude), according to World Geodetic System or WGS84, as it is widely known, which is a standard coordinate representation system for the earth. Nodes can be used to represent points of interest such as a mountain peak or a place of tourist attraction. For example, a bus-stop (node) could be marked as:

highway = bus-stop

2.1.1.3 Ways

Ways are lists of nodes that are connected, to help form a line (for a road or a river, for example) or a polygon (for a building or a lake, for example). While lines are points that are open-ended, polygons are points that form a closed loop. Ways can be depicted with the same tagging system. For example, a freeway can be depicted as:

highway = motorway

The order in which the points appear specifies which way or road should be taken. For example: If you want to move from a node, tagged by *amenity = café* to another node, *amenity = fuel*, we might draw a line representing the way and tag it as *highway =*

primary and *name* = *Main Street*. The orders in which the nodes appear also depict the traffic flow through that way.

2.1.1.4 Relations

Relations are ordered list of nodes and ways, and represent the "relationship" between nodes and ways. For example, relations can be used to represent turns on mountainous roads, long-distance roads connecting several ways, a boundary or a restriction, and the like.

There are lots of other classifications in OSM and the OSM tag representations. For example, a highway could be a motorway, or a primary, or a secondary, or a tertiary highway. Having an account with http://osm.org can help you edit the map and hence contribute.

2.1.2 OSM Data Storage

OSM data primitives or components can be stored and processed in different formats. The OSM project's main database is a Postgresql database, which has a separate table for each primitive, with numerous rows and columns containing objects for those primitives. For downloading the OSM data, several dumps are made available in several formats, the two most used ones being the XML format and the Protocol Buffer Binary

Format (PBF). Of late, data in several other formats such as shapefiles (.shp, .shx), database files(.dbf), project files(.prj) are also being used.

2.2 Postgresql Database

Postgresql, or Postgres as it is widely known, is an open-source relational database that is used by the OSM project now. The PostGIS extension that is available with the Postgresql is used for representing additional geospatial types and functions which make handling the data more easier. Importing OSM data into the PostGIS-enabled Postgres database is the basis for this project, which I will explain in the forthcoming sections.

Along with the database server running on my system, I have also used *"*pgAdmin 3*"* which is an open-source GUI-based development platform for postgres. Writing and executing queries is a lot easier using pgAdmin's rich GUI features on the contrary to executing commands at a "psql" prompt. Figure 2-2 shows the GUI features of pgAdmin 3.

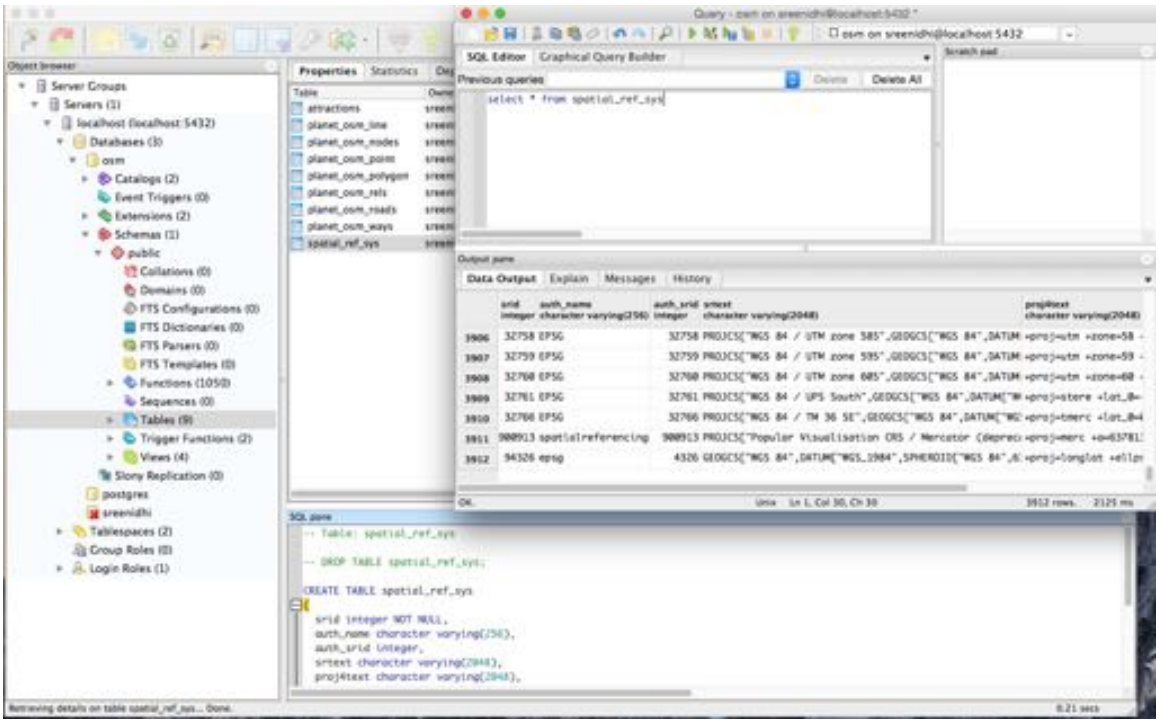

Figure 2-2: A sample screenshot showing the pgAdmin 3's GUI features

2.3 Raster Data and Vector Data – The Differences

2.3.1 Raster Data Tiling:

Initially, web maps were represented as static images that the user had to see and cannot be interacted with by the users. Most of the initial topological survey maps were static images. Later progress in the field of online web mapping used "raster tiles" or "rasterization of maps" as explained in the introduction. These kinds of maps are quicker to produce and are used in mathematical modeling and quantitative analysis. In fact, majority of today's leading mapping services like Google, Bing, Apple, etc.,use this

method of delivering the maps on the client-side, though Google Maps was the first to use this technique.

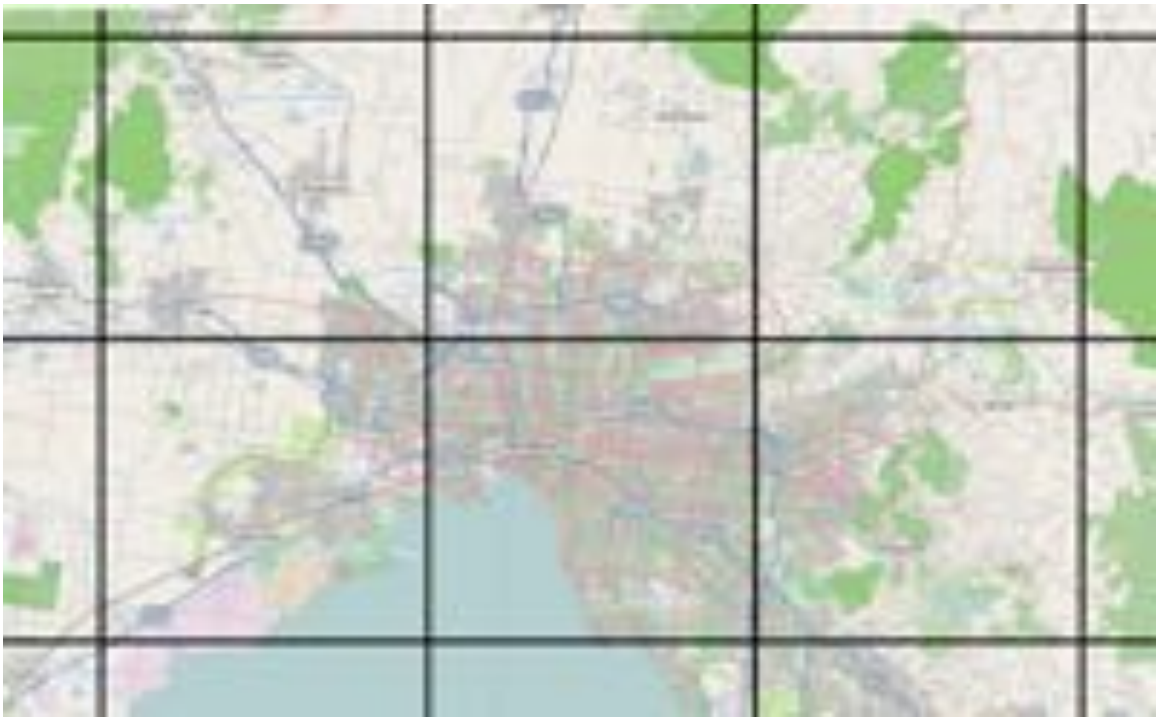

Figure 2-3: Representation of a tiled map

Figure 2-3 shows a tiled-map with lines drawn across to depict a tile. The geographic location of each tile depends on its position in the tile matrix (tiles belonging to the grid). There are several advantages to this method of tiling. For example, when the user pans or zooms, most of the tiles are still displayed while the new tiles are further fetched[10]. This was thought to improve user-side experience, in contrast to fetching a whole static map for the entire viewport.

The main disadvantage of this method is that each tile size depends on the resolution in which the data for the tile is presented. Hence, it is very difficult to represent linear features if image tiles are used. Also, storing tile sets for all the zoom levels for the entire world would be burdensome. For example, for the Openstreetmap project, a total of 18 zoom levels would produce around 5TB of data for the whole world. Storing data that is as big as this needs more disk space. Further, the rendering time for the tiles downloaded from the tile server onto the browser is more; the user has to wait till all the tiles are rendered. Of course, with high speed systems and larger disk spaces along with caching techniques[4], rendering tiles has become faster than before[8].

2.3.2 Vector Data and Vector Tiling:

Further improvements in the web mapping services led to the use of vector data[3]. Vector data for depicting geographic information is in the form of lines, points and other geometry. For example, GeoJSON† vector tiles use lines to represent roads and polygons to represent buildings and other bodies of water. In other words, data can be represented in its original form. Accurate geographical data stored is made available and no explicit

---

† GeoJSON – An open standard format (based on JSON) designed for representing geographical features.

Example: { "type": "Point", "coordinates": [30, 10] }

conversions to any other format are needed. Figure 2-4 that follows shows Mapzen's tiles for vector data[9].

Figure 2-4: Mapzen's Vector Data Tiling

Vector data processing is cumbersome and requires data cleaning at specific times. Since the topology is static, updating the vector data requires rebuilding or re-rendering the topology, which is still faster than having to redraw a whole raster image tile. Often data analysis methods might be processing-intensive, especially for larger data sets. Though vector data uses geometry such as lines and other geometry to depict actual geographical data, certain topology such as elevations can't be effectively represented. Apart from these, the main bottleneck of using vector data-based maps is performance.

Web maps are designed to be faster and existing web maps based on vector data seem to be slower in performance because of the huge amount of raw data that needs to be downloaded and rendered. Some vector tiles serve the geometric data as-is. For instance, a whole building might be included in one requested vector tile even though the building spans across different tiles. Several other vector tile sources clip the geometry according to the tile size. Hence, certain geometrical features might be chopped off.

Figure 2-5: Initial stages of this project, displaying vector data rendered on top of Google's tiled image map.

To cover up the disadvantages of both the methods, present rendering techniques use a combination of both- rendering vector data on top of raster images. Figure 2-5 is an example of this.

There are many formats for representing the vector data, used in GIS software, most of which are based on XML[1]. Though XML formats are efficient for representing spatial data, they often carry too much information, which might not support fast transmission. Hence several data compressions such as JSON compressions for the GeoJSON data make data smaller and hence transmissions faster.

2.4 WebGL

WebGL, also known as Web Graphics Library, designed and maintained by the Khronos Group is a Javascript API that helps rendering two-dimensional and three-dimensional objects and graphics on the web browser. The best part is that, it runs on its own and does not need any external plugins to be executed. It makes use of the OpenGLES 2.0 standard[12], which in turn allows GPU-accelerated rendering of effects on top of a HTML5 canvas. WebGL can also be seamlessly mixed with other javascript code to create a full-fledged working model.

Though the initial WebGL API was designed to support graphics rendering, the functionality that it could support was only minimal. Therefore, various other third-party libraries were developed which provide many high-level features. Examples of these libraries include three.js, OSG.js, CopperLicht, etc. WebGL has been used mostly in the

game industry for rendering 3D game designs, in aerospace engineering and research for rendering spacecraft models and flight simulations, particle analysis by simulation in molecular and cellular analysis, visualization of marine geo-data[2], building simulations of human face- skin rendering (example, the Lee Perry Smith head simulation) and the like.

2.4.1 Shader Code:

Though WebGL internally uses JavaScript for rendering geometry, it also has a way to manipulate the vertices; the programmer has the leverage of computing the positions, colors and textures of each vertex created in JavaScript. But this can only be done programmatically, through programmable part of the rendering pipeline mechanism. This is done by the use of GLSL (OpenGL shading Language), which uses the internal bindings of C-language programs that run in the GPU. The source code for the shader is defined in either HTML using the <script> tag or can be defined as strings that can be fed into variables in the JavaScript.

There are two main kinds of shaders[5]: "vertex shaders", which operate on the coordinate positions specified by vertex buffers, and "fragment shaders", which determine the color of each vertex (pixel). Two different types of "qualifier" variables,

the uniform variable and the attribute variable pass data from JavaScript. The uniform variables are global variables that remain the same throughout the program. Attribute variables are used to define properties for a specific vertex point. Attribute variables can be used only in the vertex shader. Apart from these two, there is also a third category, called "varying" variables. These variables are declared and set in the vertex shader and their values are used in the fragment shader. For example, for interpolating the colors between two different vertices, the color can be defined as varying attribute in the vertex shader and can be used in the fragment shader. Corresponding color buffers for these variables have to be defined in WebGL.

The output of each of these shaders is passed into special variables, for which corresponding buffers have to be defined. The output of a vertex shader is passed into a variable called *gl_Position* that specifies the position of each vertex on the screen, which is shown in figure 2-6.

The color for each of these vertices is defined in fragment shader and the output is passed to the *gl_FragColor* variable, which is again a four dimensional vector that contains the R, G, B, A values for each vertex rendered as a pixel on the screen.

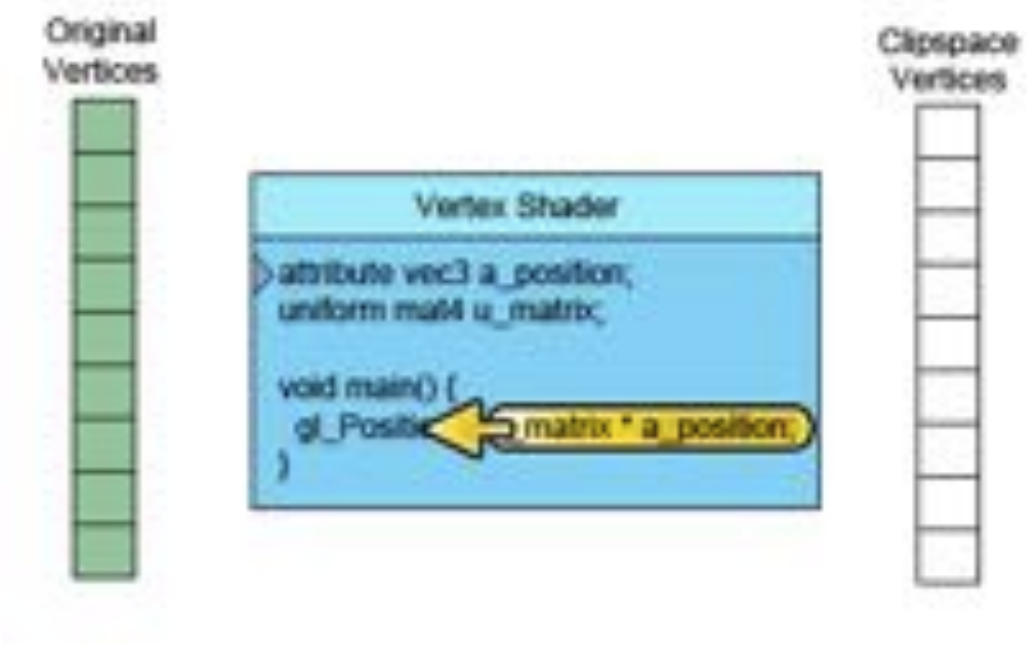

Figure 2-6: WebGL vertex shader that shows 9 vertices being processed

Table 2-1: RGB values written for each vertex of a triangle

R	G	B
0.5	0.75	0.5
0.87	0.09	0.5
0.06	0.17	0.5

2.4.2 Compiling and Linking Shader Code Boilerplate:

WebGL creates a rectangular viewport that references the HTML5 canvas. It is in this viewport, that WebGL defines the placement of the results of the vertex and fragment shaders. For this purpose, a WebGL context is created, and is fed into a context object[6].

The vertex and fragment shader functions are then called in WebGL and are compiled using the context object. The compiled program is then linked and is then made available for further use. In essence, the following sequence of steps are followed:

(i) Get the vertex and fragment shader functions from the HTML script tag, using *getElementById* and is fed into a source-code object.

(ii) The type of shader is identified from the *type* attribute of the script tag. *type="x-shader/x-vertex"* identifies that it is a vertex shader and *type="x-shader/x-fragment* identifies that it is a fragment shader.

(iii) The objects for the vertex and fragment shaders are created using the *createShader()* function, that internally uses the WebGL context object.

(iv) The source code for each of the shaders are added using the source code objects, passed as parameters to the *shaderSource()* function.

(v) Both the shaders are compiled using the *compileShader()* function.

(vi) A shader program object is then created using the *createProgram()* function. This shader program object is used for linking the shaders and contains the variables necessary for passing the data between the CPU and the GPU.

(vii) The vertex and fragment shaders are then attached to the shader program object using the *attachShader()* function.

(viii) The attached shaders and compiled shaders are now linked to the shader program object using *linkProgram()* function.

(ix) The created program object is now made available for use by the current rendering state using the *useProgram()* function.

2.4.3 WebGL Buffers

Buffers are a way of sending data to WebGL; they send vertex information to the GPU. The following are the steps to create a buffer and insert data into it:

(i) The buffer is initially created using *createBuffer()* function.

(ii) The buffer is then bound by using *bindBuffer()* function, that is in turn used by the program object created earlier. This function also says that the buffer created is the current buffer to be worked upon.

(iii) The data to be used by the program is then copied into the buffer created by using the *bufferData()* function.

(iv) Once the data is fed into the buffers, the shader's attributes are applied to it. The locations (of the vertices) specified to the attributes must be disclosed to WebGL, using the *getAttribLocation()* function.

(v) Next, the data from the buffers must be used. *vertexAttribPointer()* specifies that the data must be obtained from the buffer that was bound recently and specifies certain formats such as number of components per vertex, the stride (how many bytes of data must be ignored to get from one vertex to the next), the type of data (Byte, Float, Int, etc.), the offset (how far in the buffer is the data exactly stored) and so on.

(vi) Lastly, the buffer is enabled using the *enableVertexAttribArray()* to do all of the above functions.

The shader linking-compiling and the buffer set-up steps are carried out in almost all of the programs written in WebGL. Hence, most of the code is boilerplate, except for very few minor changes.

2.4.4 Drawing the Object

Once the buffers are bound and the shaders are linked, the next step is to draw the scene or object using draw() calls, either as lines or as triangles. There are many different

types of draw calls, the two most important ones being *drawArrays()* and *drawElements()*. While *drawArrays()* makes use of the buffers created using the above said steps, *drawElements()* requires another buffer along with the usual ones, called the "index" buffer. The index buffer specifies the indices of each of the vertex to be drawn. The catch here is that, a vertex can be drawn with or without using the indices. It is really according to the programmer's needs that an index buffer is created. For each of the vertex drawn, the corresponding index buffer has an index associated with it. It is with the help of these indices that the draw call knows which vertex must be drawn after the current vertex. For example, when drawing a cube with six faces, the index buffer tells which vertices must be joined together, with the help of the indices.

The above draw calls can draw geometry like points, triangles, lines, line loops, line strips and triangle fans. This mode of drawing is specified in the draw call using the WebGL context object created.

3. PRELIMINARY WORK

3.1 Database Setup

Initially, I started off with an idea to plot points and draw geometries (places of interest) within a bounding box. There are many sources that allow downloading OSM data from them directly or by using mirrors. The most famous among them is the http://geofabrik.de. This has downloads for each continent, county, state, city, individually and the whole world at once. As a first step, I downloaded the OSM data for North America as a .pbf file. PBF is one of the many formats that the OSM data is available in, as discussed earlier. OSM data, downloaded from the Internet cannot be used as-is and must be imported into the PostGIS-enabled Postgres database using some specific tools. I used "osm2pgsql[7]" which is a command-line based program that makes this import easy. Osm2pgsql runs on most of the operating systems, I have used a Mac OS X for this purpose. It can be installed either via macports or homebrew, for which I chose the latter. I installed "postgres.app" and "pgAdminIII", which is a GUI development platform for PostgreSQL database.

I created my database as "osm" and and added "postgis" functionality to the

database. Postgis is a database extender for Postgres, which allows support for storing spatial geometry and also allows running of location-based queries. Finally, I also added Google's spherical Mercator projection to the database, 900913 (which is the leet speak for "google"). Osm2pgsql creates the following tables when data is imported into the Postgres database, which is also shown figure 3-1.

(i) **planet_osm_line**: holds all non-closed pieces of geometry (called ways) at a high resolution. They mostly represent actual roads and are used when looking at a small, zoomed-in detail of a map.

(ii) **planet_osm_nodes**: an intermediate table that holds the raw point data (points in lat/long) with a corresponding "osm_id" to map them to other tables.

(iii) **planet_osm_point:** holds all points-of-interest together with their OSM tags - tags that describe what they represent.

(iv) **planet_osm_polygon:** holds all closed piece of geometry (also called ways) like buildings, parks, lakes, areas, etc.

(v) **planet_osm_rels:** an intermediate table that holds extra connecting information about polygons.

(vi) **planet_osm_roads:** holds lower resolution, non-closed piece of geometry in

contrast with “planet_osm_line”. This data is used when looking at a greater distance, covering much area and thus not much detail about smaller, local roads.

(vii) **planet_osm_ways:** an intermediate table which holds non-closed geometry in raw format.

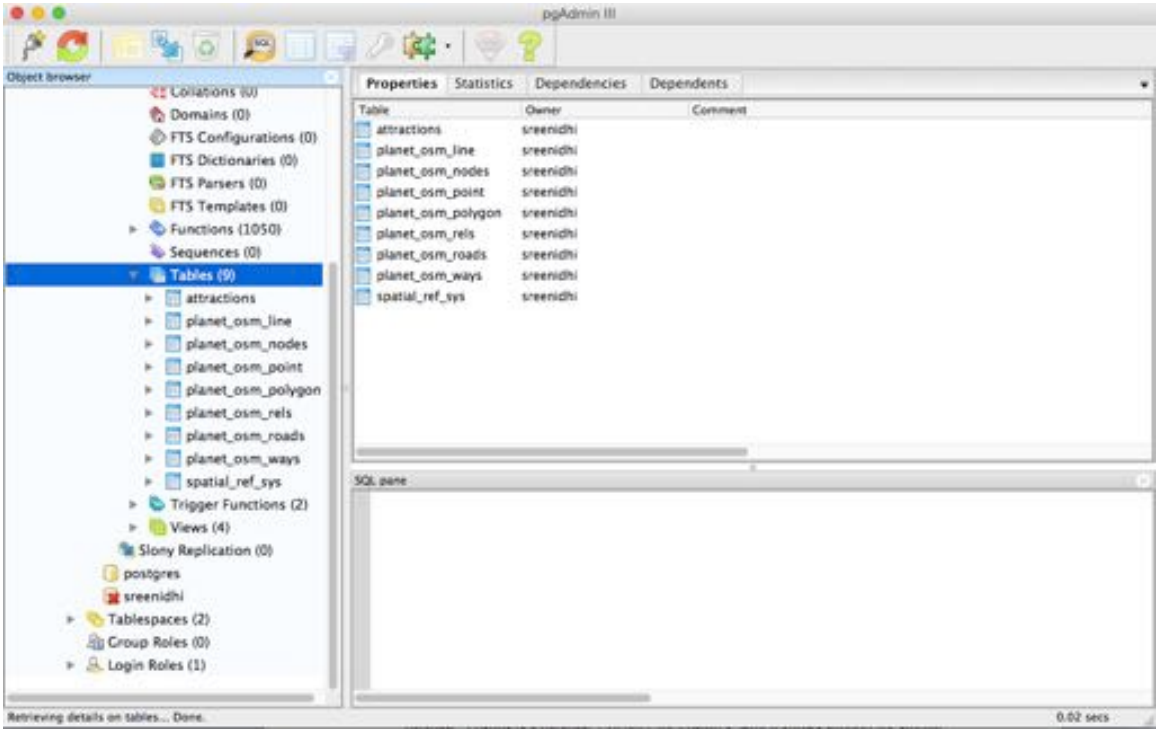

Figure 3-1: pgAdmin 3 showing the tables created by osm2pgsql when importing data into the Postgres database

3.2 Local Server Setup

This project was written and tested on Mac OSX, which comes with the Apache webserver[11] and the PHP scripting language.

The default file location for running on this webserver in a mac is the /Library/WebServer/Documents folder. Though Apache can be started by typing http://localhost on a web browser and checking that the “It Works!” message is displayed, the PHP bundle has to be enabled manually. Checking if PHP is enabled can be done by creating a phpinfo() document and running it on the local webserver.

Finally, a machine that can be tested on, that needs to have high-end graphics processing unit is required.

4. DESIGN OF ON-THE-FLY MAP GENERATOR

Figure 4-1 illustrates how the map generator draws the geometry for the vector data fetched from the Postgres database.

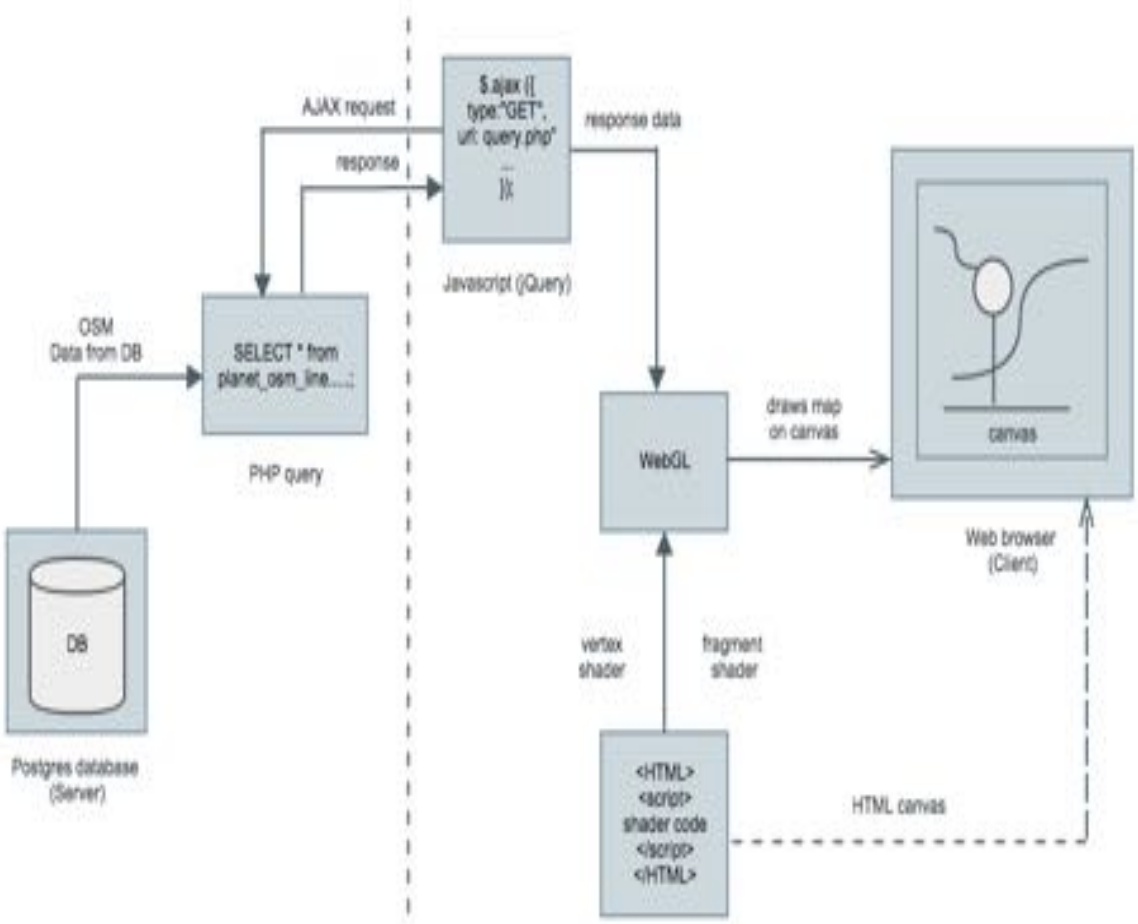

Figure 4-1 Design of On-the-fly Map Generator

The flow of creating maps started off with the Postgres database server running. Data was pulled from the database with the back-end PHP code requesting data from the

database using Postgres queries. These queries were carefully created and executed to return the desired data. The PHP server code was triggered off by an asynchronous request (AJAX call) that is written using plain JavaScript and jquery. The PHP in turn returns the data fetched to the WebGL . On the other hand, WebGL also gets the shaders to be used with the data from the HTML. WebGL internally has buffers that can manipulate the data. Finally, WebGL draws the required geometry on the HTML5 canvas which is present on the web browser (the client).

4.1 The HTML5 Canvas

The canvas is defined using HTML5 <*canvas*> tag. The canvas width and height are then defined. It is using this canvas that the WebGL context is created. Once the canvas is defined and the WebGL context is created, we draw on the canvas.

4.2 The Database and the Query

The backend is written using PHP which connects to the Postgres database, using the user credentials. Here, the database to be connected is *"osm"* which has the data organized in tables as explained in the previous chapters. The idea is to create a bounding box, taking the current positional coordinates[13] as the center. All geometry such as roads and places of interest within this bounding box are drawn. To do this, two separate

queries, one for roads (lines) and the other for points of interest (polygons) are executed one after the other and their results are stored in corresponding arrays. The query is a bit complicated as it makes use of spatial referential IDs or SRIDs, which are a way of representing the spatial data. The query makes use of spatial geometry constructors that can ease the processing of spatial data.

4.2.1 Spatial Geometry Constructors

The spatial geometry constructors used in the query are explained in this section:

(i) **ST_Transform:** This returns a new geometry with its coordinates transformed to the SRID specified. The syntax is:

geometry ST_Transform(geometry g1, integer srid);

(ii) **ST_MakeEnvelope:** This creates a rectangular bounding box specified by the minimums and maximums (of latitude and longitude coordinate points), along with the SRID. The syntax is as follows:

geometry ST_MakeEnvelope(double precision xmin, double precision ymin, double precision xmax, double precision ymax, integer srid=unknown);

(iii) ST_Intersects: This is used to check if any of the geometry spatially intersect with the bounding box created. If yes, returns true and false otherwise. The syntax is :

boolean ST_Intersects(geometry geomA , geometry geomB);

Using the above said constructors, the final query is constructed as follows, for polygons. A similar query is constructed for representing lines.

```
SELECT name, ST_AsText(ST_Transform(way,4326))
FROM  planet_osm_polygon
WHERE ST_Intersects(way, ST_Transform(ST_MakeEnvelope(($lon_min),
  ($lat_min), ($lon_max),
  ($lat_max), 94326), 900913))
```

Here, the MakeEnvelope constructor creates a bounding box using the parameters specified- they are calculated by getting the current geographical position coordinates from the browser and adding and subtracting a specific value from them to calculate the minimums and maximums. The Transform constructor transforms the “way” geometry into a way that can be read, which is calculated by specifying the SRID for WGS84 projection, which is 4326. The latitude longitude points are now legible as shown in figure 4-3:

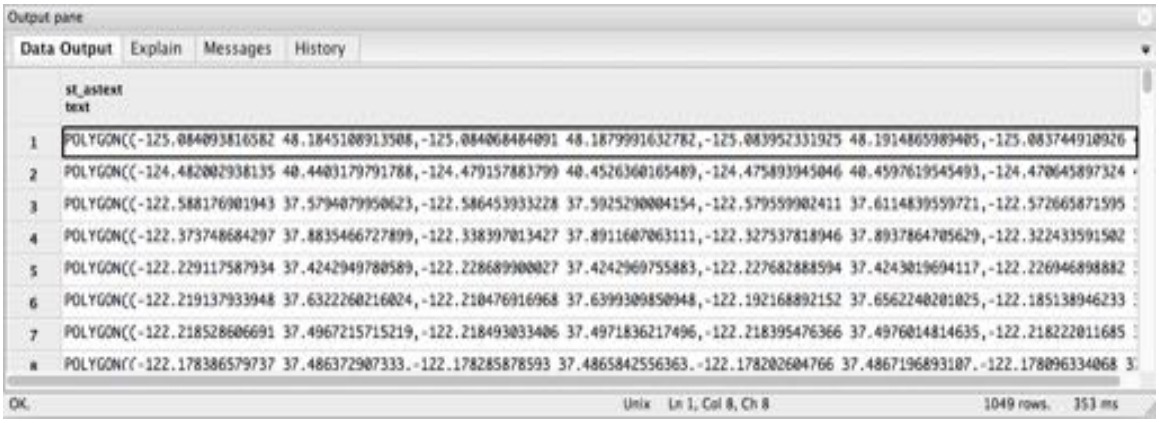

Figure 4-3 Using ST_Transform and SRID 4326

By using ST_AsText, the coordinates would be returned as a bunch of geometrical lat-lon coordinates (Well Known Text or WKT), which is a way to represent coordinate data, as shown in figure 4-4.

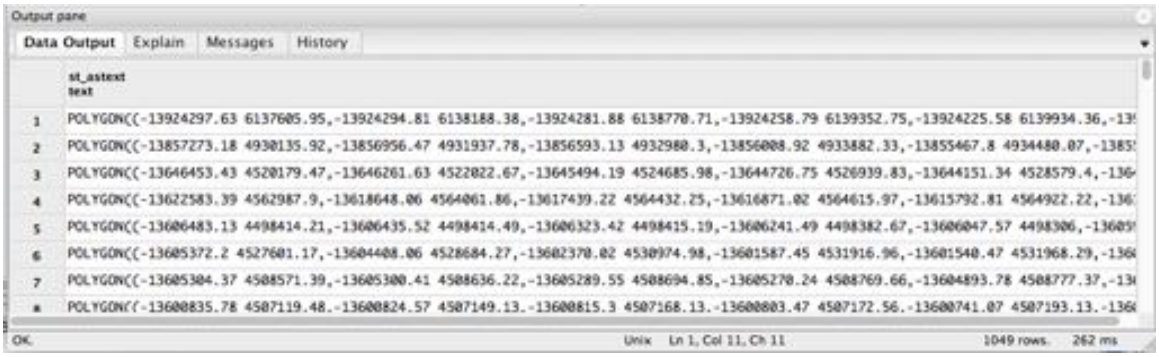

Figure 4-4 Geometry coordinates without using SRID

The resulting data returned is sent to WebGL as a set of object arrays, using *json_encode()*. Each object thus contains a set of latitude-longitude points represented for the geometry to be drawn.

5. IMPLEMENTATION

5.1 The Data

The object arrays thus returned from the server are passed on to the "success" parameter of the AJAX call. Since the received data is of a raw JSON Object form, it now has to be processed by WebGL. Moreover, the latitude and longitude coordinates cannot be plotted directly on a canvas that considers pixels. Hence, a conversion from the latitude-longitude coordinates to pixels, which the WebGL can manipulate, is done. Each of the latitude-longitude coordinates corresponds to one or many pixels in WebGL. There are specific mathematical functions that do this. In particular, the following formula is used for the conversion of latitude-longitude coordinates to x-y pixel coordinates:

$$x = \frac{\lambda + 180}{360}$$

$$y = \frac{1}{2}\ln(\,1 + \sin\varphi * \frac{1}{1 - \sin\varphi})$$

where x,y are the pixel coordinates, φ, λ are the latitude-longitude coordinates.

5.2 The Map

The converted pixel coordinates have to be scaled and transformed into coordinates that present accurate places that correspond to the latitude-longitude points,

as shown in figure 5-1 (these points are taken from the browser where my map is generated).

```
Object {x: 41.25492173454222, y: 99.13079313336024}
Object {x: 41.25468966660089, y: 99.1306520858833}
Object {x: 41.25464551656533, y: 99.13042320313387}
Object {x: 41.25456036426666, y: 99.13021450652735}
Object {x: 41.254553529080894, y: 99.13010910420037}
Object {x: 41.25453819783823, y: 99.1300187137206}
Object {x: 41.254553529080894, y: 99.12994774282004}
Object {x: 41.25455014343111, y: 99.12965096432075}
Object {x: 41.254504021930664, y: 99.12901876870046}
Object {x: 41.25473724853334, y: 99.1567143123368}
```

Figure 5-1: Converted x,y pixel coordinates

For this transformation, a scaling and a translation matrix is used. Drawing complex geometry with hunderds of thousands of lines and polygons needs pretty complex code to be written. Further, the WebGL *draw()* calls need to update the points each time for all the points plotted. To make things easier, translation matrices are used. These are defined once in JavaScript and the translations are done in the shader. Since the shader takes care of each vertex plotted, the translations are applied to each point(vertex) and have to be written only once. Scaling is done similar to translation. For example, changing the size of the point plotted may be done through scaling.

I had also set up an index buffer[6] to draw polygons. Closed lines represent polygons, since the starting and end points are the same. To differentiate between the end point of the first polygon and the start point of the next polygon, index buffers seem helpful here.

Further, to differentiate between roads and buildings drawn, I had given different colors; yellow lines represent roads while grey boxes represent building or points of interest. These colors are defined in JavaScript and are interpolated in the fragment shader.

While the colors are represented using RGBA values and hence, is a Float32Array. The pixel points, after the conversions are also represented using a Float32Array while the index buffer containing the integer indices to the points (vertices) are represented using a Uint16Array.

With all the buffers set up, the shaders compiled and linked, WebGL is now ready to draw. Since index buffers are used, the *drawElements()* call should be used to draw the vertices. The *drawElements()* renders primitive geometry that are indexed by the element array data which is the index for each vertex. The syntax is as follows:

context.drawElements(mode, count, type, offset);

The mode specifies which mode is to be used (for example, lines, triangles, points, etc.) to draw the geometry. Since I have the pixel coordinates (converted from latitude-longitude coordinates) representing vertices and I am trying to draw two-dimensional shapes of the geographic data, I have used *gl.LINES* to connect between vertices. As explained before, since polygons have the same starting and ending point, they would have lines connected between pixel points, starting at ending at the same point. Hence, polygons appear as closed, geometrical, rectangular boxes (in 2D).

The final map drawn looks like this:

Figure 5-2: Map generated for OSM data using WebGL

5.3 Zoom levels

Any map drawn must have zooming in and out capabilities available. Minor details can be looked at while zooming in while an overview of the map is presented to the user while zoomed out. Here, the initial zoom level is set to 15, from which the zoom levels are increased and decreased according to the buttons pressed by the user.

Figure 5-3 Map zoomed out at zoom level 12

The buttons from zooming in and out are created using HTML's *<input>* tag. These buttons are placed on the bottom-right corner of the canvas as shown:

Figure 5-4 Controls for zooming in and out

5.4 Panning around and Resizing the canvas

The map also has “panning around” feature, though, shows the map that is drawn only for the data fetched. When the user pans, a specific value is either incremented or decremented from the current position’s latitude-longitude values. The map then pans around according to the button selected. Figure 5-5 shows this.

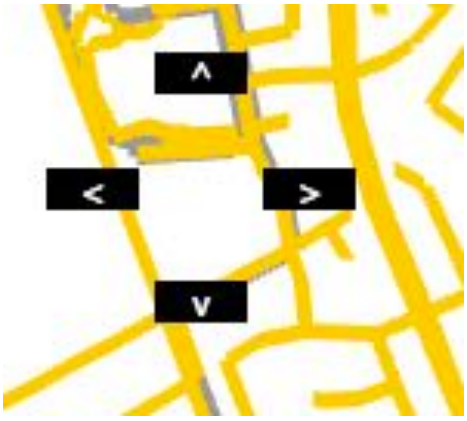

Figure 5-5 Controls for Panning around

As far as resizing the canvas is concerned, when the user resizes the browser window, the canvas inside also resizes according to the display width and height. Hence the map drawn is not distorted. Figure 5-6 explains this.

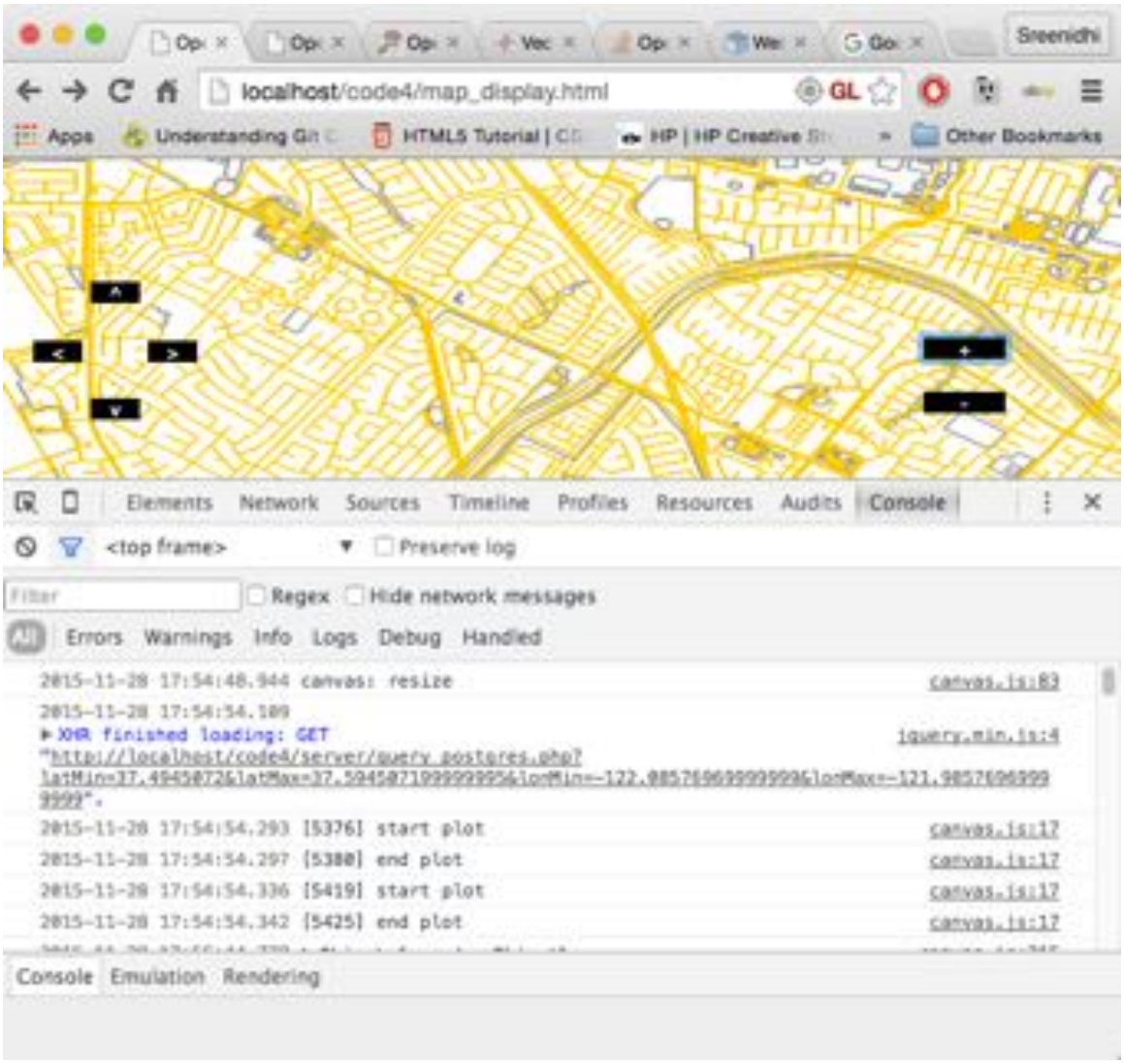

Figure 5-6 Resized browser window with the canvas resized accordingly

6. EXPERIMENTS AND RESULTS

To check the efficiency of the maps drawn, I tested its rendering time with that of the existing tile generators. To do this, I set my a tile server on my local machine, which generates tiles according to the bounding box and zoom levels specified, along with a rendering engine that renders the generated tiles on the browser.

6.1 Setting up a Tile Server

Since I already have the data imported into my Postgres database using Osm2pgsql, I installed mapnik[14], which is an open-source toolkit for rendering maps. Mapnik can be directly downloaded and built from its sources, but Mac OSX versions have a mapnik bundle that is available through the "homebrew" which is a command-line tool for installing packages. Hence, I installed the latest version of mapnik using the following command:

$brew install mapnik

Once mapnik was installed, I also installed the mapnik tools that were released by the Openstreetmap project, to make effective use of the mapnik rendering library:

$svn co http://svn.openstreetmap.org/applications/rendering/mapnik

Mapnik also uses certain prepared files to generate images for coastlines and oceans. This is thought to be faster than reading the entire database for each of the zoom levels. I installed these prepared files (shoreline, world boundaries and natural earth data) using the command-line tool "curl".

Mapnik tools has *generate_xml.py,* along with osm.xml which has the styling rules for generating tiles. These can be edited and can have our own set of rules. The following is the command given:

$./generate_xml.py osm.xml --dbname osm --user sreenidhi --port 5432 --extent -13645007.61 4483388.83,-13540684.11 4587617.36 --accept none

where "extent" is the bounding box defined. After this, the *generate_image.py* script should be executed. This script, on successfully executing, generates an image.png file, which is the image of the tile specified by the bounding box.

To generate tiles, the *generate_tiles.py* has to be modified a little. The bounding box has to be well defined in the python file along with the zoom levels. I had generated tiles for the above said bounding box (Newark city, CA), for zoom level 13. When this python script is executed, it generates tiles in a special hierarchy of folders, identified by the zoom level. For example, the tiles generated for the bounding box specified looked like this:

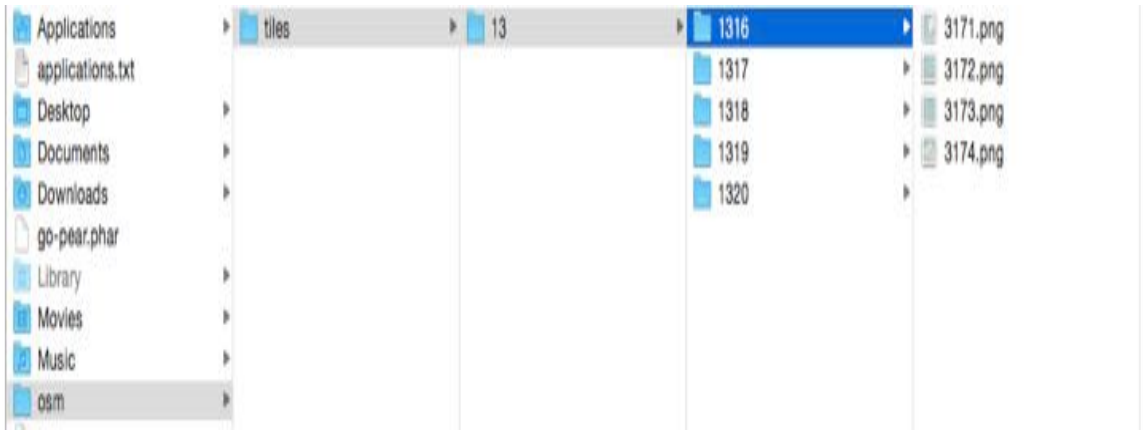

Figure 6-1 Special hierarchies of folders that hold images of tiles

Now that the tiles are generated, there must be a means of rendering them. This entire hierarchy of folders is moved to the web server's document root folder. I used OpenLayers[15] to render the generated tiles on the web browser. I downloaded the latest version of OpenLayers. Further, I had to create a HTML file that calls the OpenLayers.js. The zoom levels and the bounding box (in WGS84) are specified and the path to the image files is included in the OpenLayer's instance. Finally, these images are reassembled in the client side and are rendered in the browser:

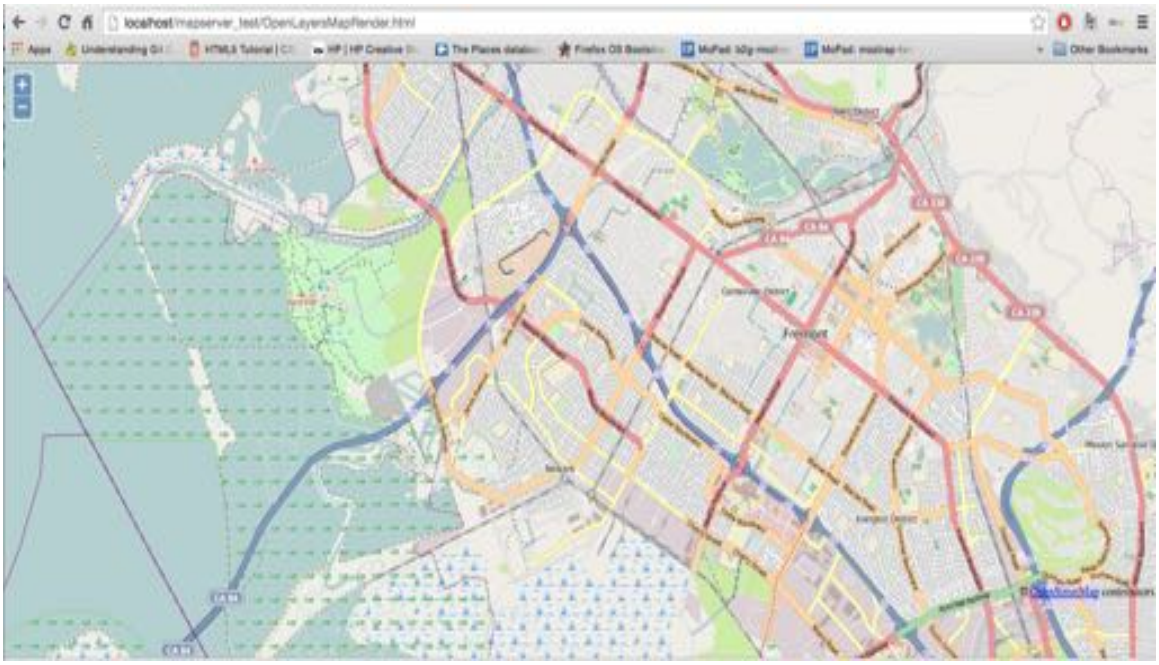

Figure 6-2 Map tiles rendered using mapnik and OpenLayers

6.2 Timing Tests

The whole process of rendering tiles versus rendering vector data using WebGL was timed, taking into consideration, a particular zoom level and a bounding box of coordinates. The results are as follows:

Test Bench Used: Macintosh OSX Yosemite, with 1.8 GHz dual core CPU, 4GB memory, an Intel HD Graphics 3000 GPU with a speed of 350-1350 (Boost) MHz with 12 unified pipelines and a browser that supports WebGL- Google Chrome, version 47.0.2526.73 (64-bit) with WebGL 1.0 enabled. The number of threads was 2, which

corresponds directly to the number of CPU cores. Further, to reduce the data size returned from the query, I had already converted the lat-long values to floats (from string), which reduced the data size to some extent and further truncated the last 3 digits after the decimal point. With this data in hand, I further used "gzhandler" in PHP to further compress the data and reduce its size. The float-converted, truncated, compressed data was thus reduced to a considerable size of 684 KB.

Comparisons between the data download time for the two methods yielded the following results:

Table 6-1 Average data download times (in seconds) for both methods

Type of data download	**Pass 1**	**Pass 2**	**Pass 3**	**Pass 4**	**Pass 5**
Time taken for the query to execute and get back data (query time)	1.88 s	1.95 s	1.89 s	1.86 s	1.79 s
Time taken to generate tiles (Mapnik + OpenLayers)	3.86 s	3.88 s	3.91 s	3.85 s	3.89 s

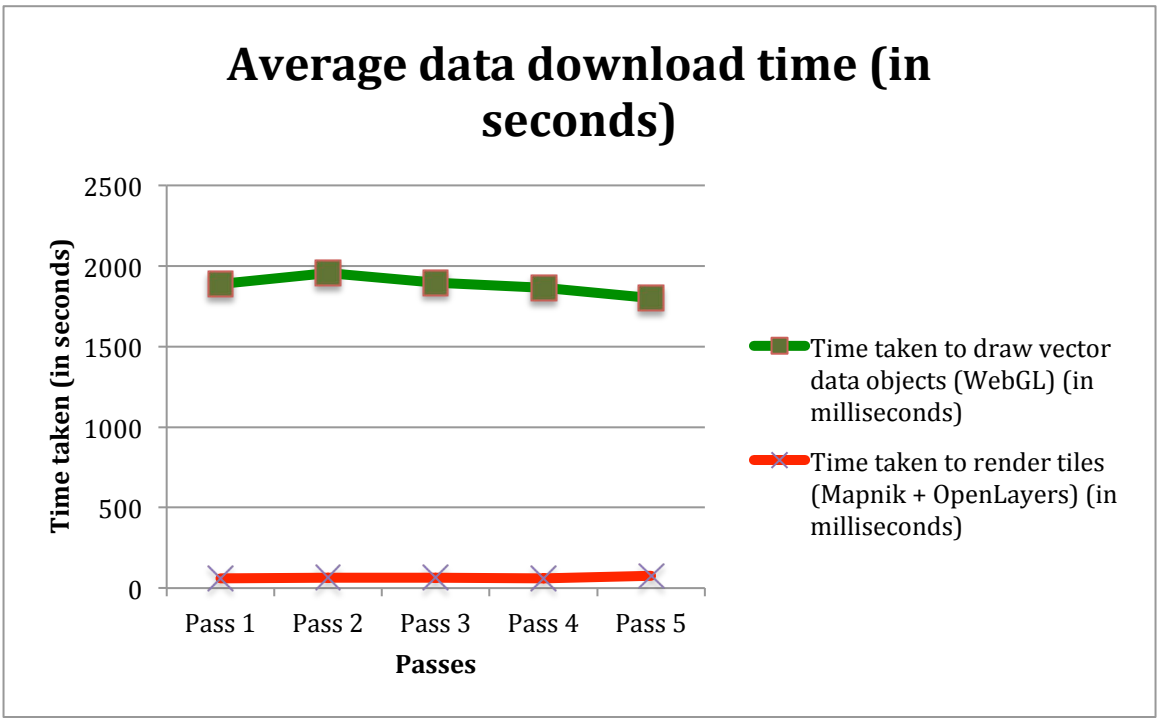

Figure 6-3 Average data download times (in seconds) for both methods

From the table and the graph, it is evident that the AJAX request used in this project takes a lesser time than the tile generation process. The average time taken for the former 1.87 seconds while the latter takes 3.71 seconds, which says that using this project has a time reduction of 38% approximately for data download through AJAX calls.

On a real-world scenario, further comparisons between the two different methods were as follows. Considering a particular bounding box, for a particular zoom level, say 15, the traditional way of online maps would opt to get the corresponding tiles from the tile server and then render them on the browser. If the tiles are cached at different zoom levels for the whole world, then the cached tiles are returned accordingly. We could then calculate the tiles sizes (in bytes) for all the tiles returned and check if the data size returned from the query for the same bounding box and zoom level is lesser or greater.

In our approach, we could try to cache the query results. If we could pre-compute the bounding boxes and the corresponding query results for the whole world, just as the tiles are pre-computed, we could hope that the cached query data would return data that is lesser in size than the tiles. Then, the data that is sent to the browser to be drawn would be lesser, and hence would reduce the network traffic. With this argument, I calculated the byte sizes for both the tiles and the query results. Having the bounding box parameters for Newark, CA, I was able to find that while the tiles returned from my tile server were 762 KB in size (in total), the data that my query returned after truncation and compression was around 684 KB, which is an 86% reduction of data size from the original 5 MB.

Comparing the size of the tiles and the size of the query data returned, the following results were observed:

Table 6-2 Comparison of tile sizes versus queried data size, for bounding boxes

Places observed, according to bounding boxes	**Size of tiles**	**Size of query data**
Fremont	762 KB	684 KB
San Jose	2.4 MB	1.7 MB

Assuming that the bounding box is pre-computed (just one bounding box), the average rendering times were also calculated, which were as follows:

Table 6-3 Average Rendering times (in milliseconds) for both methods

Type of rendering	**Pass 1**	**Pass 2**	**Pass 3**	**Pass 4**	**Pass 5**
Time taken to draw vector data objects (WebGL)	8 ms	6 ms	7 ms	6 ms	9 ms
Time taken to render tiles	61 ms	63 ms	63 ms	60 ms	76 ms

Since I am experimenting only for one bounding box (calculated according to current position), these results are comparable. If this is extended on a larger scale and if bounding boxes and the corresponding query results are computed for the whole world, then the results might be competitive. Thus, lesser data would be sent to the browser to render and hence network traffic would be considerably reduced.

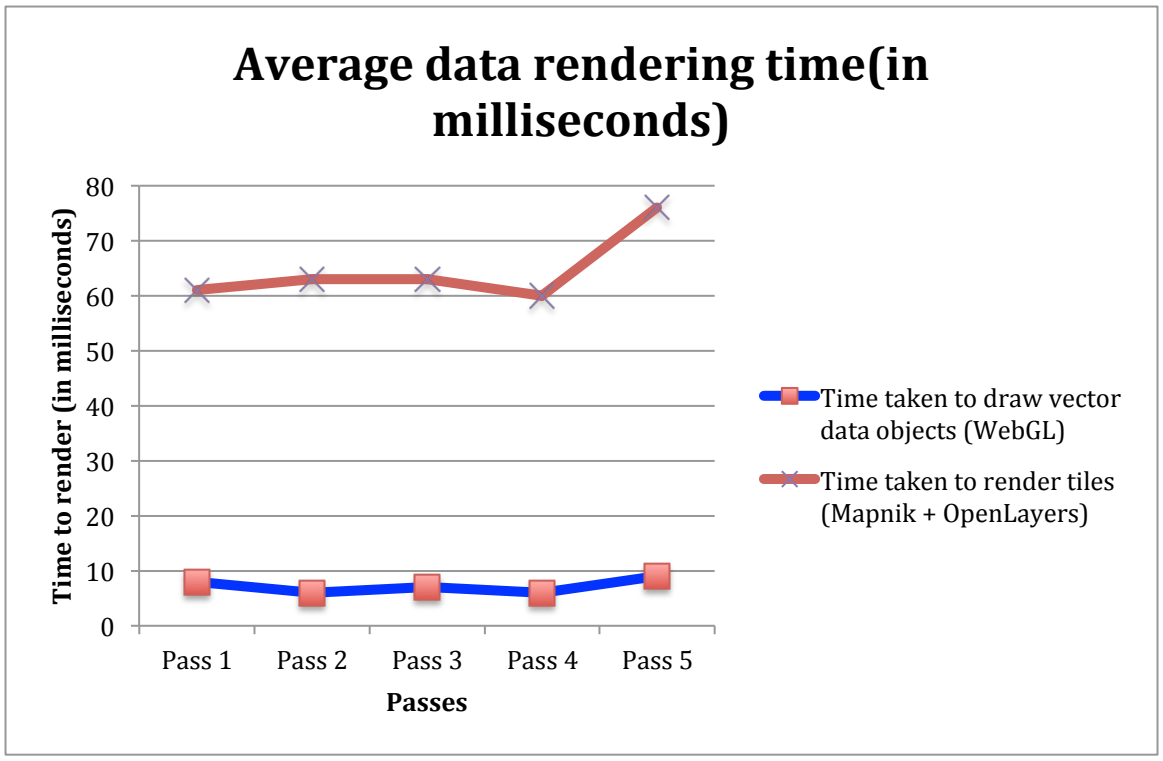

Figure 6-4 Average Rendering times (in milliseconds) for both methods

7. CONCLUSION AND FUTURE WORK

7.1 Conclusion

This project aims to show that drawing primitives for the OSM vector data using WebGL is an on-the-fly approach, without having to store any pre-drawn geometry. The data, according to the user's location keeps changing and hence the corresponding geometry is drawn on-the-fly. While the traditional tile servers rendering raster images are slowly being changed to tile servers that show images for vector data, my project, if implemented full-fledged might be a new way of generating web maps. This project, if implemented in full-scale, might considerable reduce the network traffic and the data being sent to the browser for rendering. Of course, the users need to have a better system with WebGL enabled and better hardware that support graphics, but as the world of technology is evolving, this might not be a big bottleneck and neither might thwart people from using this system.

This project in short, helped me learn a lot about online maps and their functionalities. It helped me analyze their disadvantages and upon discussing with Prof. Chris Pollett, I was able to provide effective solutions to overcome them. Initial stages started off with having Google maps underneath, to check if the geometry that I drew is

on the right place. Lots of tweaks were then implemented, starting from removing the Google Maps underneath the canvas to removing all dependencies on third party APIs; this project is solely written in plain Javascript and WebGL.

7.2 Future Work

Lots of enhancements can be made, as this is very much in its beginning stage. For now, even lakes and other water bodies are considered as polygons. Osm2pgsql does not have a separate table that stores the names of water-bodies. So, as a first extension, I will try to pull out the geographical coordinates of water bodies by performing a filtering on the query. These water bodies can then be shown in different colors, perhaps the standard one, blue. Next, I will also try to implement a filtering mechanism, which creates bounding boxes according to the user's search query. For this, I will have an input text box that the user can use to search. Further, I can also try to differentiate between normal residential streets, expressways, freeways by performing a filtering on the available data. Next, I will also try to extend and draw the geometry for bigger bounding boxes. All this, however requires further experimentations to make sure that the final product I create is usable by everyone.

Bibliography

[1] Behrens, Jan. Segmentation of OpenStreetMap Data: Generating, Merging, and Distributing Tiles. University of Bremen, September 2011.

[2] Resch, Bernd, Wohlfahrt, Ralf and Wosniok, Christoph. Web-based 4D visualization of marine geo-data using WebGL. Cartography and Geographic Information Science, 2014.

[3] Gaffuri, Julien. Toward Web Mapping with Vector Data, Geographic Information Science. Volume 7478, 2012.

[4] Loechel , J. Alexander and Schmid, Stephen. Caching techniques for high-performance Web Map Services. Proceedings of the AGILE'2012 International Conference on Geographic Information Science, Avignon, April, 24-27, 2012.

[5] Parisi, Tony. Programming 3D applications with HTML5 and WebGL. O'Reilly Media, 2014.

[6] Learning WebGL, 3D Programming for the Web. Retrieved on December 30, 2014, from http://learningwebgl.com/blog/?p=28

[7] LearnOSM, Learn OpenStreetMap Step-by-step. Retrieved on January 4, 2015, from http://learnosm.org/en/osm-data/osm2pgsql/

[8] MapsGL, by Google Maps team. Retrieved on January 10, 2015, from http://www.chromeexperiments.com/detail/mapsgl/

[9] Mapzen, Vector Tiles. Retrieved on February 5, 2015, from https://mapzen.com/projects/vector-tiles/

[10] Opencycle Map, the Openstreetmap Cycle Map. Retrieved on March 25, 2015, from http://opencyclemap.org/

[11] PHP Manual. Retrieved on June 20, 2015, from https://secure.php.net/manual/en/index.php

[12] WebGL Specification, Khronos Group. Retrieved on July 8, 2015, from https://www.khronos.org/registry/webgl/specs/1.0/ - 2.2

[13] Pollett, J. Christopher. Maps: Determining Location in HTML5, CS175 Lecture Notes, from http://www.cs.sjsu.edu/faculty/pollett/175.1.14f/Lec26112014.html - (4)

[14] Building Tiles with PostGIS Openstreetmap Data and Mapnik: Your own Openstreetmap. Retrieved on November 20, 2015, from http://www.bostongis.com/PrinterFriendly.aspx?content_name=generating_osm_tiles

[15] Using your own custom built OSM tiles in OpenLayers. Retrieved on November 22, 2015, from http://www.bostongis.com/PrinterFriendly.aspx?content_name=using_custom_osm_tiles