

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

CS 297 Project
Presented to
Dr. Christopher Pollett
San José State University

In Partial fulfillment
Of the Requirements for CS 297: Master's Project(Part A)

By
Sreenidhi Pundi Muralidharan
December 2014

ACKNOWLEDGEMENT

I would take this opportunity in thanking my friends and family for the support given and my mentor, Professor Dr. Chris Pollett for guiding me while doing this project.

INTRODUCTION

This project is an approach to create an On-the-fly Tile Generator For Openstreetmap Data Using WebGL.

OpenStreetMap (OSM) is a worldwide geographic database created from public domain data sources and user created data. It contains a vast amount of data for a variety of features, including administrative boundaries, streets, waterbodies, points of interest, and buildings. Such a large chunk of dataset, being difficult to store on servers and render as-is, is fragmented into smaller chunks of uniform dimensions. These chunks are referred to as “tiles”, and are bitmap images in essence, which are in turn the building blocks for online maps. The concept of tiles is then applied to the available geodata in terms of vectors such as lines, nodes and polygons. OSM data can be used in different formats; I have used an XML format of the OSM data (files use .osm extension).

Existing tile generators generate PNG overlay tile images from a wide range of datasources, 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 thereby render the tiles on-the-fly using WebGL. Here, all of the vector computation is pushed to the GPU(called the shader code). This also means that lesser data needs to be sent to the browser. Henceforth, I have written a WebGL program for rendering OSM data.

Lastly, I experimented with Tile Generators□ to study how tiles are generated in the existing vector-based tile generators. I have written in detail, about my findings, in this report.

The following are the deliverables that I have done, to understand the essence of WebGL and the working of tile servers. I will now discuss the organization of the rest of the paper.

In the next section, I have discussed about my first deliverable, which involves understanding the WebGL API and finding out how to write WebGL programs. This deliverable is an effort to convert the HelloWorldOpenGL program, from the book, "Foundations of 3D Computer Graphics", by Steven J. Gortler, into a corresponding WebGL program.

Section 2 describes the contents of my second deliverable. I had created presentation slides for OpenstreetMap and Vector tiles. This deliverable helped me understand the OpenStreetMap basics and what vector tiles are.

Following this is section 3 in which I have described what I did for my third deliverable. In this deliverable, I imported 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).

The last section describes deliverable 4, in which I have experimented with the existing “Tile Generators” and have studied the time they take to render tiles. This report includes a detailed analysis of my findings.

DELIVERABLE 1 - WebGL PROGRAM: THE LOGIC BEHIND

This deliverable was to create image texture maps using WebGL. Texture mapping refers to the process of applying (mapping) a texture image to the surface of an object. Texture mapping is done to produce textures that give the look and feel of real-world objects and where interpolating with colors might be a tedious task. The pixels that make up the texture image are called texels (texture elements), and each texel codes its color information in the RGB or RGBA format.

The process of texture mapping affects both vertex and fragment shaders (if a shader program is being used). This is because, it applies the texture coordinates to each vertex and then applies the corresponding pixel color extracted from the texture image to each fragment. Here, two images, smiley.jpg and reachup.jpg are used for creating the textures. These images are overlapped onto each other, to create a new texture. This deliverable is an effort to convert the HelloWorldOpenGL program, from the book, "Foundations of 3D Computer Graphics", by Steven J. Gortler, into a corresponding WebGL program.

The deliverable consists of three different files, HelloWorldWebGL.html, HelloWorldWebGL.js and shader-utils.js. The HTML part of the code defines the canvas needed to draw the texture-mapped images and specifies the URL of other javascript files. The HelloWorldWebGL.html file contains the following piece of code:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Hello World WebGL</title>
  </head>
  <body onload="main()">
    <canvas id="myWebGLContext" width="512"
height="512">
      </canvas>
      <script src="webgl-debug.js"></script>
      <script src="webgl-utils.js"></script>
      <script src="shader-utils.js"></script>
      <script src="HelloWorldWebGL.js"></script>
    </body>
  </html>
```

The above html code defines the canvas on top of which the WebGL drawing is done. The canvas is given an id `myWebGLContext` which is 512 x 512 in size. Further, the HTML code, specifies the URL of the external javascript files such as `HelloWorldWebGL.js`, `shader-utils.js`, `webgl-utils.js` and `webgl-debug.js`

WebGL's error reporting mechanism involves calling `getError` and checking for errors. Having to put a `getError` call after every WebGL function call is difficult and can increase redundant lines of code, there is a small library to help with make this easier.

The `webgl-utils.js` file contains functions every webgl program will need a version of one way or another. For example, this file contains functions for setting up a WebGL context and messages that should be displayed on getting a browser that supports/doesn't support WebGL.

The following code is the content of `HelloWorldWebGL.js`:

```
var VSHADER_SOURCE =
'attribute vec4 a_Position;\n' +
'attribute vec2 a_TexCoord;\n' +
'varying vec2 v_TexCoord;\n' +
'void main() {\n' +
'  gl_Position = a_Position;\n' +
'  v_TexCoord = a_TexCoord;\n' +
'}\n';
```

This defines the vertex shader that specifies the texture coordinates. These are then passed on to the fragment shader.

The fragment shader has the following code:

```
var FSHADER_SOURCE =
'#ifdef GL_ES\n' +
'precision mediump float;\n' +
'#endif\n' +
'uniform sampler2D u_Sampler0;\n' +
'uniform sampler2D u_Sampler1;\n' +
'varying vec2 v_TexCoord;\n' +
'void main() {\n' +
'  vec4 color0 = texture2D(u_Sampler0,
v_TexCoord);\n' +
'  vec4 color1 = texture2D(u_Sampler1,
v_TexCoord);\n' +
'  gl_FragColor = color0 * color1;\n' +
'}\n';
```

This pastes the texture image into the geometric shape specified. The texture coordinates are then specified in the `initVertexBuffers()` method as follows:

```

function initVertexBuffers(gl) {
    var verticesTexCoords = new Float32Array([
        // Vertex coordinate, Texture coordinate
        -0.5,  0.5,  0.0, 1.0,
        -0.5, -0.5,  0.0, 0.0,
         0.5,  0.5,  1.0, 1.0,
         0.5, -0.5,  1.0, 0.0,
    ]);
    var n = 4;

```

In the above lines, the vertex coordinates (-0.5, 0.5) are mapped to their corresponding texture coordinates (0.0, 1.0), and so on. The following lines then define a buffer object using which, the positions of the vertices are written into the vertex shader:

```

    var vertexTexCoordBuffer = gl.createBuffer();
    . . .
    // Write the positions of vertices to a vertex
    shader
    gl.bindBuffer(gl.ARRAY_BUFFER,
vertexTexCoordBuffer);
    gl.bufferData(gl.ARRAY_BUFFER,
verticesTexCoords,      gl.STATIC_DRAW);

```

The next step is to get the storage locations of the position of the vertices and the texture coordinates and load them into the buffer. The following lines of code help us achieve that:

```

var FSIZE = verticesTexCoords.BYTES_PER_ELEMENT;
//Get the storage location of a_Position, assign and
enable buffer
var a_Position = gl.getAttribLocation(gl.program,
'a_Position');
gl.vertexAttribPointer(a_Position, 2, gl.FLOAT,
false, FSIZE * 4, 0);
gl.enableVertexAttribArray(a_Position); // Enable
the assignment of the buffer object
// Get the storage location of a_TexCoord
var a_TexCoord = gl.getAttribLocation(gl.program,
'a_TexCoord');
gl.vertexAttribPointer(a_TexCoord, 2, gl.FLOAT,
false, FSIZE * 4, FSIZE * 2);
gl.enableVertexAttribArray(a_TexCoord); // Enable
the buffer assignment

```

The next step is to prepare the texture image for loading, and request the browser to read it. This is done by the `initTextures()` method.

```
var texture0 = gl.createTexture();
var texture1 = gl.createTexture();
```

The above code creates a texture object (`gl.createTexture()`) for managing the texture image in the WebGL system. The line below gets the storage location of a uniform variable (`gl.getUniformLocation()`) to pass the texture image to the fragment shader:

```
var u_Sampler0 = gl.getUniformLocation(gl.program,
'u_Sampler0');
var u_Sampler1 = gl.getUniformLocation(gl.program,
'u_Sampler1');
```

In the next step, it is necessary to request that the browser load the image that will be mapped to the rectangle. An image object is required for this purpose. The following lines of code illustrate this:

```
var image0 = new Image();
var image1 = new Image();
// Register the event handler to be called when
image loading is completed
image0.onload = function(){ loadTexture(gl, n,
texture0, u_Sampler0, image0, 0); };
image1.onload = function(){ loadTexture(gl, n,
texture1, u_Sampler1, image1, 1); };
// Tell the browser to load an Image
image0.src = 'smiley.jpg';
image1.src = 'reachup.jpg';
```

Next, the `loadTextures()` function has to be modified as follows:

```
function loadTexture(gl, n, texture, u_Sampler,
image, texUnit) {
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, 1); // Flip
the image's y-axis
    // Make the texture unit active
    . . .
    // Bind the texture object to the target
    gl.bindTexture(gl.TEXTURE_2D, texture);

    // Set texture parameters
```



```

        gl.texParameteri(gl.TEXTURE_2D,
gl.TEXTURE_MIN_FILTER, gl.LINEAR);
        // Set the image to texture
        gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
gl.UNSIGNED_BYTE, image);
        . . .
        if (g_texUnit0 && g_texUnit1) {
            gl.drawArrays(gl.TRIANGLE_STRIP, 0, n);    //
Draw the rectangle
        }
    }
}

```

In `loadTexture()`, predicting which texture image is loaded first is difficult, because the browser loads them asynchronously. The sample program handles this by starting to draw only after loading both textures. To do this, it uses two global variables (`g_texUnit0` and `g_texUnit1`) indicating which textures have been loaded.

These variables are initialized to false at initially and changed to true in the ‘if statement’. This if statement checks the variable `texUnit` passed as the last parameter in `loadTexture()`. If it is 0, the texture unit 0 is made active and `g_texUnit0` is set to true; if it is 1, the texture unit 1 is made active and then `g_texUnit1` is set to true. Following this, `gl.uniform1i(u_Sampler, texUnit)` sets the texture unit number to the uniform variable.

Finally, the program invokes the vertex shader after checking whether both texture images are available using `g_texUnit0` and `g_texUnit1`. The images are then combined as they are mapped to the shape, resulting in the screenshot as follows:

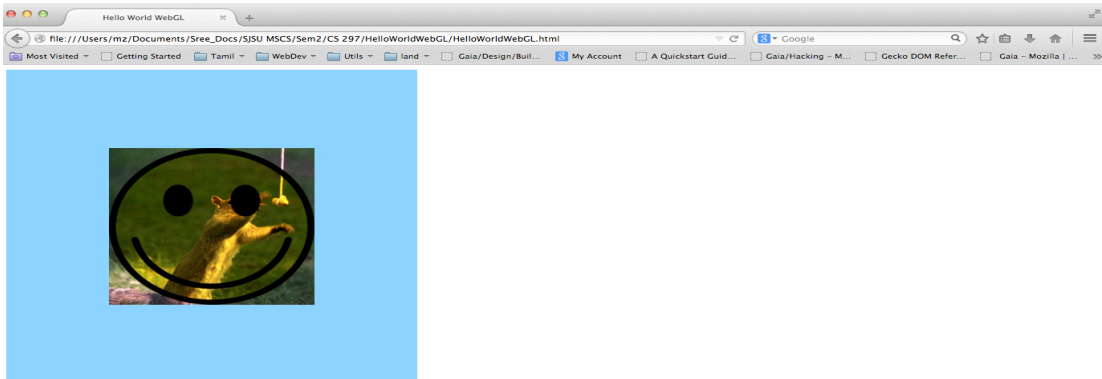


Figure 2-1: Screenshot of the WebGL program that uses texture-mapping

DELIVERABLE 2 - THE STUDY OF OSM DATA AND VECTOR TILES

This deliverable was to understand the OSM basics and 2D vectors. I have created two powerpoint presentations, one on OSM basics and one on vector tiles. I have described each in turn, below.

2.1 OpenStreetMap Data

OpenStreetMap is a very large wiki of XML data about the geography of the entire planet. It is in fact, a collaborative project to create a free, editable map with the information provided by contributors. The initial map data was originally collected from scratch by volunteers performing systematic ground surveys using a handheld GPS unit and a notebook, digital camera, or a voice recorder. The data was then entered into the OpenStreetMap database.

More recently, the availability of aerial photography and other data sources from commercial and government sources has greatly increased the speed of this work and has allowed land-use data to be collected more accurately by the process of digitization. When especially large datasets become available, a technical team manages the conversion and import of the data.



Figure 1-2 Map rendered using OSM data

The following are the data structures used primarily in OSM:

2.1.1 Tags

Tags are a way to represent a point or a node in the map. They are represented as key-value pairs. Examples of tags could be:

- 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"/>
```

2.1.2 Nodes

A node or a point can be used to mark a particular place in the map. Nodes are marked using tags as explained before. For example, a bus-stop (node) could be marked as:

```
highway = bus-stop
```

2.1.3 Ways

Ways are represented as lines in the map. Ways could be rivers, roads, a fence, etc. A way connects two or more nodes. Ways can be depicted with the same tagging system. For example, a freeway could be tagged as:

```
highway = motorway
```

The order of nodes in which they appear specifies which way 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.

2.1.4 Closed ways (Areas or Polygons)

An area is a closed way and can represent lakes, buildings, etc. For example, a plaza would be an area tagged with

```
highway = pedestrian and area = yes
```

2.1.5 Relations

Relations are groups of nodes, ways, or areas. The following are its types:

- Route: includes interstate routes, cycling routes, and bus routes
- Multipolygon: areas with multiple parts or holes
- Boundary: for administrative boundaries
- Restriction: to describe turn restrictions

There are lots of other classification in OSM and the 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.

2.2 Vector Tiles

Vector tiles are packets of geographic data, shaped up as pre-defined tiles. For a browser to render a map, the map data is requested by the browser(client) as a set of "tiles" corresponding to square areas of land of a pre-defined size and location. The server then returns vector map data, which has been clipped to the boundaries of each tile, instead of a pre-rendered map image. Compared to an un-tiled vector map, the data transfer is reduced, because only data within the current viewport, and at the current zoom level needs to be transferred.

Vector tiles for rendering OpenStreetMap data were first proposed in March 2013 and are supported by Mapnik, the most widely used renderer of OpenStreetMap data. The tile server pipeline, TileStache includes a vector tile provider called VecTiles. Deliverable 3 describes the setting up of Mapnik and TileStache for rendering geographic tiles.

DELIVERABLE 3 - IMPORTING OSM DATA INTO A POSTGRES DATABASE

In this deliverable, I downloaded a small area of OSM data in XML format. I chose the state of CA and used 'osm2pgsql' which is a command-line based program that imports OpenStreetMap data to postGIS-enabled PostgreSQL databases. 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.

The following are the steps followed in importing OSM data into the postgres database:

Step 1: Create and prepare the database using the following commands:

```
createuser -U postgres <username>
createdb -U postgres UTF8 -O <username> gis
```

where 'gis' is the name of the database to be created.

```
createlang -U postgres plpgsql gis
```

installs the language plpgsql

Consecutively, adding postgis functionality to the database is important and is done as follows:

```
psql -U postgres -d gis -f
PATH_TO_POSTGRES/share/contrib/postgis-
1.5/postgis.sql
```

Finally, adding a projection (specifically spherical Mercator) is done. This is represented by 900913 in PostgreSQL database. The file 900913.sql is downloaded from the following github repo: <https://github.com/openstreetmap/osm2pgsql>

```
psql -U postgres -d gis -f PATH_TO_FILE/900913.sql
```

Step 2: Adding/Importing OSM data into the database created:

Change directory to where you unzipped osm2pgsql:

```
cd PATH_TO_OSM2PGSQL
```

Import OSM data:

```
osm2pgsql -U postgres -s -S ./default.style
PATH_TO_OSM_FILE/osm-california.osm
```

Osm2pgsql creates the following tables when the data is imported into the postgres database:

- **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.
- **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
- **planet_osm_point:** holds all points-of-interest together with their OSM tags - tags that describe what they represent
- **planet_osm_polygon:** holds all closed piece of geometry (also called ways) like buildings, parks, lakes, areas, etc.
- **planet_osm_rels:** an intermediate table that holds extra connecting information about polygons
- **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
- **planet_osm_ways:** an intermediate table which holds non-closed geometry in raw format

As a next step, I tried to write an SQL query, which expresses the following condition: 'Given a bounding box represented by latitude and longitude coordinates, check if the highway 'I 5' passes through the bounding box, or not'. As a solution, I retrieved the latitude and longitude for the desired bounding box using the following query:

```
SELECT
  ST_AsText (ST_Transform (ST_GeomFromText ('POINT(' | | '-
1361466073' / 100 | | ' ' | | '451747960' / 100 | | ')',
900913), 94326));
```

This gives the latitude and longitude for the given positional point, taken from planet_osm_point, as shown below:

Output pane	
Data Output	Explain Messages History
	st_astext text
1	POINT(-122.302571660907 37.5601802504715)

Figure 3-1: Output of the above query

The above coordinates correspond to the upper bounds of the bounding box. The same query is used to find out lower bounds of the bounding box. The, using the coordinates for the bounding box, we could check if the highway 'I 5' passes through the bounding box or not. The following query produces the desired result:

```
SELECT
  osm_id,
  highway,
  ref,
  name
FROM planet_osm_line
WHERE highway IS NOT NULL
AND ref = 'I 5'
AND ST_Intersects(way, ST_Transform(ST_MakeEnvelope((-
122.302571660907), (37.5601802504715), (-122.436330806712),
(37.8074531813721), 94326), 900913));
```

In the above query, 94326 is the SRID (spatial reference ID) for Web Mercator projection used and can be added to the spatial_ref_sys table. This table contains all of the spatial projections used. The following query helps adding web mercator projection:

```
INSERT INTO spatial_ref_sys (srid, auth_name, auth_srid,
proj4text, srtext)
VALUES (94326, 'epsg', 4326, '+proj=longlat +ellps=WGS84
+datum=WGS84 +no_defs ', 'GEOGCS["WGS
84", DATUM["WGS_1984", SPHEROID["WGS
84", 6378137, 298.257223563, AUTHORITY["EPSG", "7030"]], AUTHORI
TY["EPSG", "6326"]], PRIMEM["Greenwich", 0, AUTHORITY["EPSG", "8
901"]], UNIT["degree", 0.01745329251994328, AUTHORITY["EPSG", "
9122"]], AUTHORITY["EPSG", "4326"]]);
```

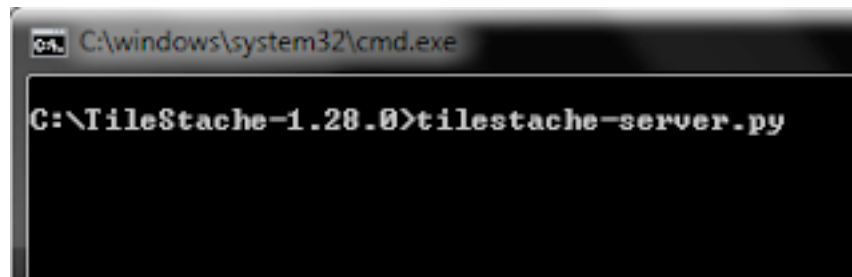
DELIVERABLE 4 - STUDY OF CURRENT TILE GENERATORS IN TREND

In this deliverable, I have studied the existing tile generators and their nature of generating tiles. I have installed the following:

- Python 2.7 - Mapnik runs on python. Python can be installed on a Mac by either macports or homebrew. On windows, download the python installer and install the same. Make sure to update the PATH variable with the location of python.
- Mapnik - An open source toolkit for rendering maps. Available on Mac OS by initiating the following command: `brew install mapnik`. Make sure to update the PATH variable with the location of mapnik.
- TileMill – TileMill is, as stated in their homepage, an interactive application for creating beautiful maps. It uses mapnik at its core to render the maps and uses very simple UI and CSS-like language to style them.
- TileStache - TileStache is a Python-based server application that can serve up map tiles based on rendered geographic data.

I have used a Windows environment here. Installing mapnik creates a few demo tiles, which gives a clue that mapnik has been setup right.

After installing TileStache, it is run using the "tilestache-server.py" script, inside the scripts folder as follows:



```
C:\windows\system32\cmd.exe
C:\FileStache-1.28.0>tilestache-server.py
```

Figure 4-1: Starting Tilestache server

The above implies that the server is running on localhost, on port 8080., as shown below:

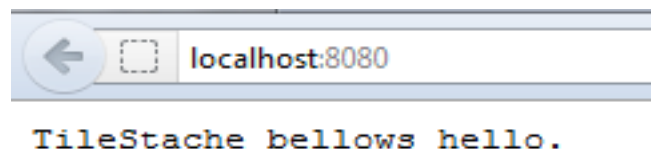


Figure 4-2: Tilestache server running on localhost

Using the default tilestache server configuration file produces a world-tile as shown below:

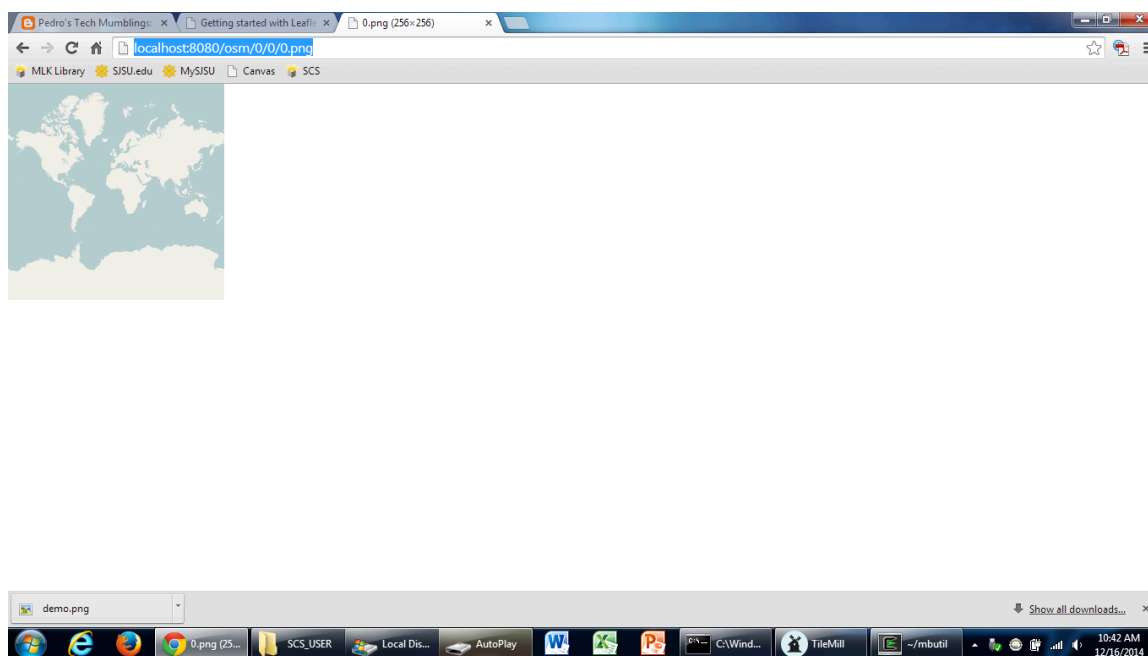


Figure 4-3: Default configuration produces world tile

Now, using TileMill, I have attempted to create a new project. I have downloaded the osm xml files for California from <http://download.geofabrik.de/north-america/us/california.html> and the shape files for California roads from <http://scec.usc.edu/internships/useit/content/primary-and-secondary-california-roads>. The shapefile format is a popular geospatial vector data format for geographic information system (GIS) software. I have tried to add a new layer and tried to import the shape files into the TileMill project created.

Adding the shape files produces desired styling effects in TileMill. After this, I have used the 'Export' option in TileMill to export the shapefiles into the desired format. I have chosen MBTiles as the chosen format as this deliverable is intended for generating tiles. Further, I have used Cygwin on Windows to install github and mbutil, which is a command line tool to generate tiles.

Running the following command at the Cygwin prompt produces tiles for the shapefile imported in TileMill:

```
$mb-util OSM-california.mbtiles OSM-california
```

The following screenshot shows the tile generated:

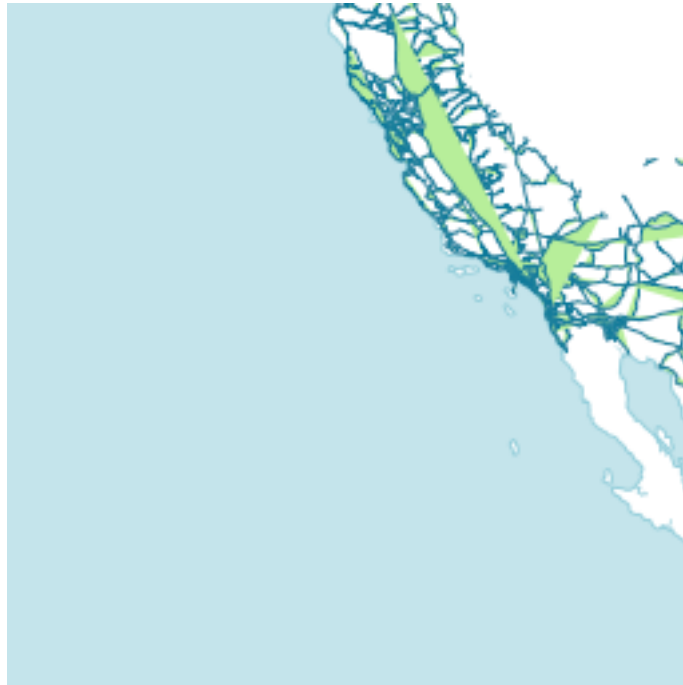


Figure: 4-4 Tile Generated using TileMill for Californian Roads

Upon investigating, I have concluded that using a CPU might take anywhere between 0.5 seconds to 2 minutes (tile generation for the whole world) using traditional tile generating methods. Using shader programs that run on GPUs will definitely accelerate the process of tile generation.

CONCLUSION

I now know how to query for a particular highway road within a bounded box of latitude and longitude. I would further work on how to create 2D shapes: points of interest like buildings, lakes, roads, etc., within a bounded box of latitude and longitude coordinates using shader programs.

This CS 297 project helped me understand the basics of tile generation and WebGL shader programs. It has given me a solid foundation for the basis of my future dissertation. My CS 298 project would involve studying more about tile generators and hence help me write shader programs that run on GPUs to produce tiles for the world, on a large scale. Since these tiles are to be generated on the fly, there is no requirement of building a tile server to store the generated tiles. Of course, I will implement a mechanism to cache the already generated tiles, so that it further speeds up the entire process.