Globe Crime Visualization

Roy Portas - s4356084

May 14, 2017

1 Introduction

The aim of this project is to create a web based visualization of global crime data. This will be done through a 3D model of a globe with crime data superimposed on top of it.

The visualization aims t be a easy to use tool for the general public to view crime trends around the world over time.

2 Methods

2.1 Acquiring Data and Domain Research

Data was acquired from the website [website].

Before building the web app, research was done on existing solutions, such as Google Earth and ...

2.2 Designing the Object Hierarchy

The object hierarchy allows us to define groups of objects to manage the transformations and rotations more easily. It usually takes the form of a tree structure with leaf nodes being the smallest parts of the model.

The complete model will consist of the following elements:

- The world
- The camera
- The sun (light source)
- The moon, which rotates around the world
- Crime heat maps, which are displaying on the world.

Thus the model can be representing in the following hierarchy:

INSERT TREE DIAGRAM

2.3 Version 1: Pure WebGL

The initial prototype was created in pure WebGL, which is graphics library for web browsers with an API similar to OpenGL ES2.0. The first step to creating the WebGL program is initialize WebGL and setup the shaders and buffers. To start with a simple cube was created, the shape of the cube was defined as vertices.

```
1  const cubeVertexBuffer = gl.createBuffer();
2  gl.bindBuffer(gl.ARRAY_BUFFER, cubeVertexBuffer);
3  const vertices = [
4      // Front face
5      -1.0, -1.0,  1.0,
6      1.0, -1.0,  1.0,
7      1.0,  1.0,  1.0,
8      -1.0,  1.0,  1.0,
9
10     // Back face
11     -1.0, -1.0, -1.0,
12     -1.0,  1.0, -1.0,
13     1.0,  1.0, -1.0,
14     1.0, -1.0, -1.0,
15
16     .... // Do for all faces
17 ];
18
19 gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
20
21 // We are using 3 dimensions
22 cubeVertexBuffer.itemSize = 3;
23 // We need 24 vertices to represent a cube
24 cubeVertexBuffer.numItems = 24;
```

This buffer contains all the points required to create a cube. WebGL then takes this JavaScript and converts it into GLSL, which is ran on the GPU.

After defining the position vertices, the colour buffer was created, for this example the colours was kept simple, a solid green colour for all the cube.

Once these buffers are setup, the next thing to do is to setup the shaders. For this prototype two shaders where used, a fragment shader and a vertex shader.

The fragment shader in this case just sets up the GPU to use the correct data types for the rest of the code. The vertex shader generates coordinates in the clipspace, which is used by the GPU to render the objects correctly.

Once these shaders were created two matrices were set up, one for the model view matrix, the other for the projection matrix. The model view matrix transforms coordinates into view coordinates, which is used by the GPU to display the objects. The projection matrix is used to used to manage the perspective of the scene.

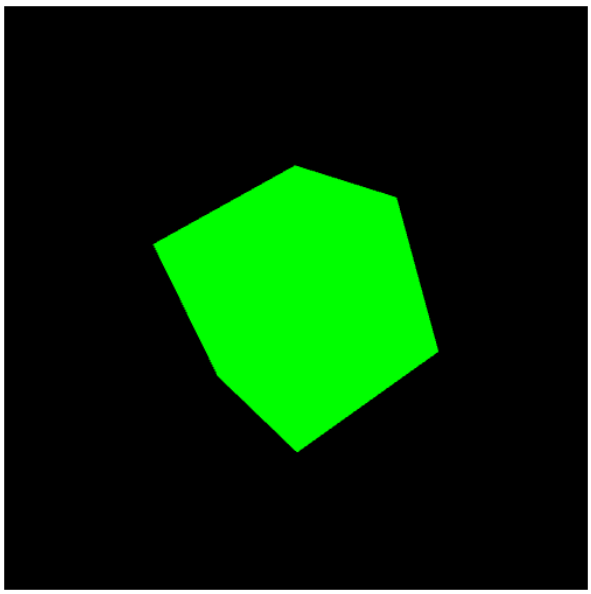


Figure 1: Simple WebGL Application

Figure 1 shows the simple application running. The program just shows a cube, without any lighting. However the code to generate this cube was quite verbose, luckily there is a variety of Javascript frameworks which provide wrappers around the WebGL interface.

One such framework is ThreeJS, which provides a lot of helper functions to make creating programs easier.

2.4 Version 2: ThreeJS

As the project became more complex, it became hard to manage all the WebGL code. Thus a third party library called ThreeJS was used. ThreeJS provides a lot of useful helper functions to write WebGL applications more easily, such support for animations, common geometries and lighting.

2.4.1 Setting up the Scene

ThreeJS provides some helper functions on top of WebGL to create scenes more easily.

For a simple application, a sphere was created in the center of the scene with a camera looking at it. The projection matrix logic is abstracted into the camera, thus for a 3D application a PerspectiveCamera was used. The PerspectiveCamera calculates the projection matrix for visible objects, which saves writing a lot of GSLS code.

The sphere was draw using a simple mesh that does not react to lighting changes and has a solid colour of blue. The result is shown in Figure 2.

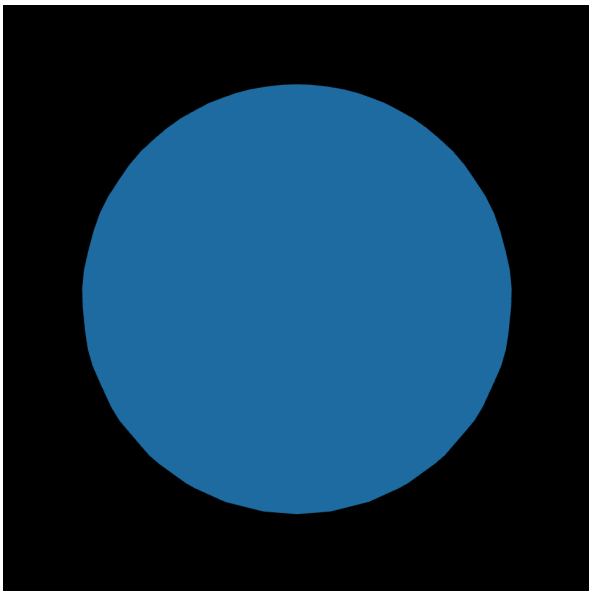


Figure 2: Plain Sphere

2.4.2 Meshes and Shading

The sphere looks plain without any lighting applied to it, thus the type of mesh was changed to one that supports shading.

ThreeJS supports Physically Based Rendering, which simulates natural light iterations with real world materials[]. PBR in ThreeJS uses image based lighting that allows objects in the scene to pick up surrounding colours from the environment.

More specifically ThreeJS uses the Metallic Roughness style of PBR. This method uses two maps, one for roughness and one of metalness. Roughness is similar to conventional specular maps, however is better optimized for real time rendering[]. Metalness defines how metallic an object is, which affects the way light is reflected from the object.

Adding PBF to the simple sphere model allows light to be reflected from it as well as adding a shadow to the sphere, which can be seen in Figure 3.

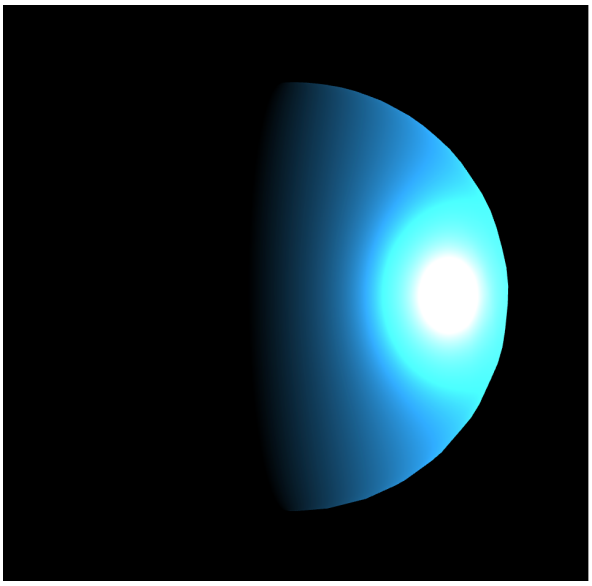


Figure 3: PBF Sphere

2.4.3 Texturing and Maps

The next step is to apply the texture to the sphere. The texture was downloaded from `http://www.shadedrelief.com/natural3/index.html`, which offers high resolution globe images for free.

Texturing the sphere in ThreeJS is easy, it just requires loading the image then applying the texture to the object. The final result can be seen in Figure 4.

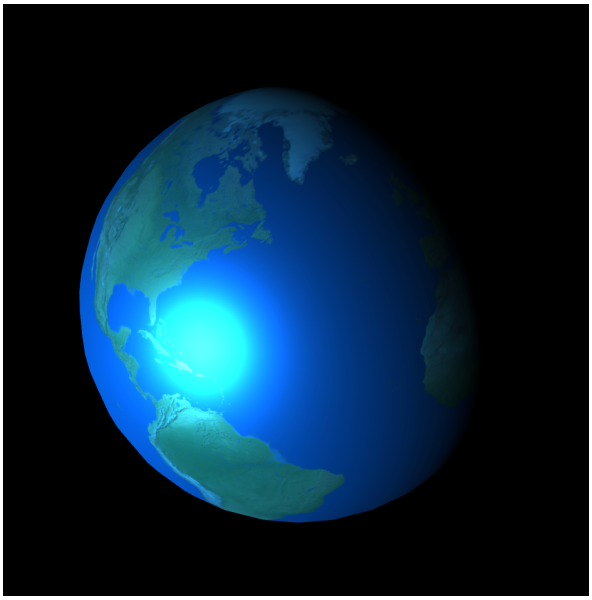
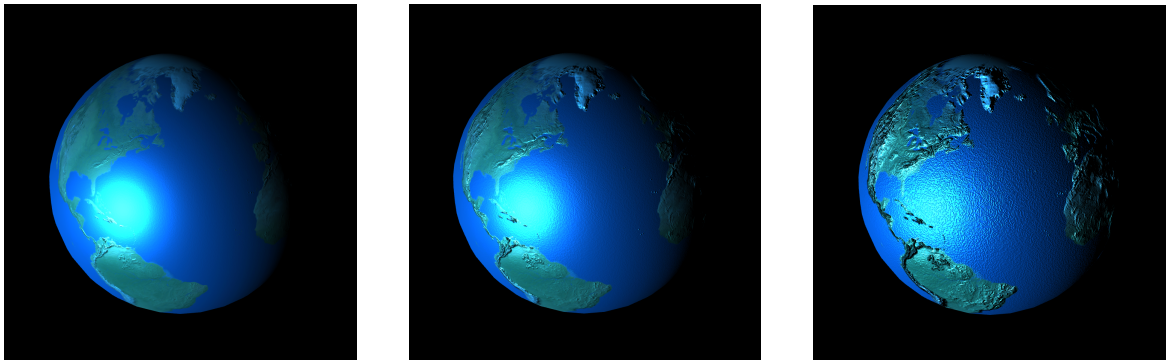


Figure 4: Textured Sphere

At the moment the lighting system is extracting the blue colour from the ocean to give the light a blueish hue. This will be corrected when the lighting is improved.

Next a bumpmap can be applied to the globe to simulate mountains and valleys on the earth. Bump maps do not modify the actual surface, instead the surface normal is is modified as if the surface has been moved. This results in a more real looking world. The bump maps were downloaded from the same site that provided the earth maps.



(a) 0.05 scale (b) 0.2 scale (c) 1 scale

Figure 5: Bump Map Comparison

Figure 5 shows different values for scales for the bump maps. Choosing the right scale is important for realism, as a too high value will over exaggerate small displacements, such as ocean sea levels. This effect is shown in the third image. However choosing a too small value will not give the desired effect and result in a flat looking planet.

The second scale was chosen, as it provided a nice looking result.

2.4.4 Lighting and Specular Maps

The next step is to improve lighting to make the scene look more realistic. To do this two lights will be used, one to represent the sun and another that follows the camera.

The sun light will light up the Earth and simulate the earth revolving around the sun. It will be the brightest light in the scene.

The second light is a smaller light that follows the camera, allowing the sun to be visible when the sun is behind the Earth.

3 Results

4 Discussion

5 Conclusion