Globe

2022 Computer Science NEA

https://github.com/pedroddvo/globe

Pedro Oliveira

2022

Contents

Chapter 1

Analysis

1.1 Background Research

1.1.1 Graphics

For my project, I am thinking of using a third party graphics library to make it simple to create a sphere, without having to go low level.
I could write the graphics from scratch using `WebGL`, however I feel like this is quite low level, and requires a lot of unnecessary time waste when I could just import a bare bones library.
A library that interests me is `THREE.js`[1], it is specifically a graphics library, so no time spent having to work with object physics. It also has strong documentation, a wide variety of examples, and a lot of real world uses. It's free, and it's also built to be a single file, which means it'll be easy to import into my project. An alternative is `babylon.js`[2], it comes with many tools allowing for easy production, as well as a graphical tool that allows you to chain systems and create materials and textures without a single line of code. The problem I have with this is that I think it's too much, I don't think these tools will save me much time, and are hard to move to other computers.

I will probably create my project using `THREE.js`.

[1] https://threejs.org
[2] https://www.babylonjs.com

1.1.2 APIs

For the retrieval of country information there are two ways of doing it, calling an API and retrieving the information that way, or storing all the information on a local database and retrieving it that way.
The reason I am choosing the former is because I think requesting information from an API is a simpler option. It does have its drawbacks, however. I have plotted some pros and cons regarding this:

API Requesting		Local Database	
Advantages	Disadvantages	Advantages	Disadvantages
Easy	Potentially slow	Fast	Requires hosting
Simple	Request limitations	No request limits	Complex
No storage space needed	Not tailored	Customizable	Large file size

Since I am not seeking speed, and I am theoretically requesting with no limits, I think that I should use API Requesting to keep my project backend simple.
I now need to choose a good API, the attributes I am looking for when searching for an API are:

- It must be free to use, or be publicly hosted.
- It must be relevant to this project.
- It should require no login.
- It should contain information in regards to a specific country. (in other words, be able to accept `POST` requests)
- It could contain extra relevant information.

The first public API I came across was 'worldpop.org'[3], it contains numerous interfaces for demographics, and has real world use for demographic research. It contains data like, births, urban changes, population history, etc... What I concluded from looking at worldpop was the fact that it is too specific, for example, to find the population of a country, it requires a certain radius of resolution - essentially, I cannot simply search for a country and retrieve its population.

[3]https://www.worldpop.org

The second public API I came across was 'countriesnow.space'[4], I like it for its simplicity. It's a lightweight API with quite descriptive yet simple documentation and allows for filtering using SQL-like queries. It contains data such as population data for a country and city, flags, currencies, capitals, cities, ISO codes and states. It's not too in depth, however is very simple.
The third API I came across was the WHO (World Health Organisation) API[5], it has some what in-depth documentation, however, is too specific for my use case. It has extremely extensive data - from road safety, to nutrition, to AIDS statistics.
Another API which I came across was The World Bank's API. Their API is very simplistic, and does not require any POST requests. Instead, a simple, well formatted URL including options will output the data needed.

Example requests:

Example 'worldpop.org' request:

```
https://www.worldpop.org/rest/data/pop
{
  "data": [
    {
      "alias": "pic",
      "name": "Individual countries"
    },
    {
      "alias": "wpgp",
      "name": "Global per country 2000-2020"
    }
  ]
}
```

Example 'countriesnow.space' request:

```
https://countriesnow.space/api/v0.1/countries/population
{
  "country": "nigeria"
}
```

Example WHO request (a simple GET request):

```
https://ghoapi.azureedge.net/api/INDICATOR
or filtered:
https://ghoapi.azureedge.net/api/INDICATOR?\$filter=Dim1 eq 'MLE
    ' and date(TimeDimensionBegin) ge 2011-01-01 and date(
    TimeDimensionBegin) lt 2012-01-01
```

Where `INDICATOR` is the indicator ID found on the WHO website. An indicator ID is the data to be requested.

[4]https://countriesnow.space
[5]https://www.who.int/data/gho/info/gho-odata-api

With this information, I can conclude that the 'countriesnow.space' example is quite short, and straight to the point. It's also readable in itself, it retrieves the population of the country of Nigeria.

For my use case, I am thinking of using a combination of the WHO API and the 'countriesnow.space' API. Initially, I can retrieve trivial data such as population statistics using 'countriesnow.space', and for some more interesting data I could include some indicators from the WHO API. I could also use The World Bank's api for environmental data, economical data, and more from their World Development Indicators.

1.2 Analysis of Current & Similar Systems

The goal of this section is to learn from what similar and current systems, websites and tools that have a similar goal as my project.

1.2.1 Github

On the github website[6], there contains a beautiful graphical globe, similar to my idea. The globe looks something like this: It contains a nicely deep

Figure 1.1:

blue coloured sphere, with an atmosphere, where the countries are stylized as dots, with lines representing messages being sent to and from the dots. It's a nice artistic touch, however I am more interested in their implementation, as analysing their process will greatly benefit my project.
Luckily, I encountered a blog post[7] which greatly describes their thought process and which algorithms they may have used to render this globe. By

[6]https://github.com
[7]https://github.blog/2020-12-21-how-we-built-the-github-globe/

reading through this blog post, I've learned a couple of things which I may use to integrate into my project:

- They used `THREE.js` to render the globe.
- They didn't use a texture mapped onto the sphere to represent the countries, instead they use about "12,000 five-sided circles to render the earths regions". In the blog post, they provided a small algorithm to create the regions.
- They allowed the user to see their own location by marking a point on the globe. I think this is an interesting idea and I might do something similar in my project.
- They drew the lines between regions using **Bezier Curves**:

```
curve = CubicBezierCurve3(startLocation, ctrl1, ctrl2,
    endLocation)
```

- They performance optimized their code. Since Github is massive, they probably need to do this so their globe runs smoothly on all machines. This may be an idea I need to focus on if my program becomes too slow, however since I am only rendering one sphere, I don't think this should be a problem. Should it be a problem, I may need to research ways of optimizing my code.

1.2.2 Earth Day 2018 - Plus 360 Degrees

"Earth Day 2018 - Plus 360 Degrees"[8] is a cinematic experience over the earth. It was created for Earth Day to remind people about the beauty and uniqueness of the planet.

Figure 1.2:

This website creates a beautiful and realistic representation of the earth. The fact that you can run this on your browser amazes me, so I think it would be smart to try and take some things down. Unfortunately, this isn't such a simple case as Github, there is very little information on how they created the website. However, in their credits[9] they do include some information.

- They rendered the earth with `THREE.js`
- They animated the sphere with `Greensock`

Some things I want to note, that I have thought of when seeing this website:

- I like the way they used the clouds, it adds a layer of realness, and I may incorporate this into my project using shaders or by layering a sort of transparent texture on top of the sphere.

[8] https://earth.plus360degrees.com
[9] https://earth.plus360degrees.com/credits/

- Their world texture is very highly detailed, and I think that also adds a layer of realness. I think I want to focus on making my earth as 'real' as possible, as that may keep my client interested on my project.

1.2.3 Experiments with Google - WebGL Globe

Google has featured in their 'blog' "Experiments with Google" a WebGL globe[10]: "The WebGL Globe is an open platform for geographic data visualization. We encourage you to copy the code, add your own data, and create your own."

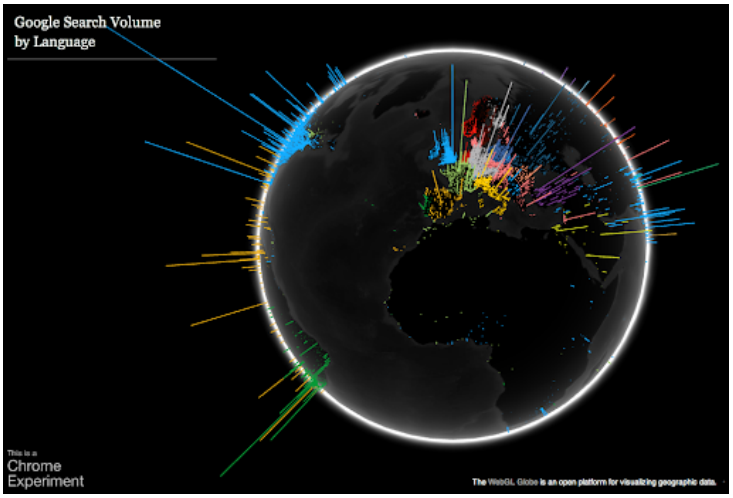


Figure 1.3:

This is a project I may take a lot of inspiration from, as it features many of the features that I need in my project, and includes code[11] that I may take inspiration and learn from. It features:

- Latitude / longitude data spikes
- Color gradients based on data values
- Mouse control and wheel to zoom

I analysed their code and development tactics, and picked out some things that I may use for my project.

- They render their sphere and data using `THREE.js`
- They format their data using `JSON`

[10] https://experiments.withgoogle.com/chrome/globe
[11] https://github.com/dataarts/webgl-globe

- They render the data using boxes as 'points', here's some snippets of code that I found that show this, the comments start with '//' and are notes that I have taken:

```
// Here they create the geometry for a point (as shown
    it is a box of a length)
geometry = new THREE.BoxGeometry(0.75, 0.75, 1);

// Each point is now a mesh of that box.
point = new THREE.Mesh(geometry);
```

That concludes my observation, however it doesn't show how they actually plot the points, that actually happens in a function `createPoints()`:

```
// This is how they determine the plot length
// _baseGeometry is a variable that contains data
   about where each point is, and each vertex of the
   point
var padding = 8-this._baseGeometry.morphTargets.length
   ;
// Iterate through all points
for(var i=0; i<=padding; i++) {
  // Push all the vertices to the target where they
     will all be rendered
  this._baseGeometry.morphTargets.push({'name': '
     morphPadding'+i, vertices: this._baseGeometry.
     vertices});
}
```

Furthermore, they have a function `addPoint()`, which I find too verbose to feature here, however, I will take snippets of the function and analyse each one. This is how they position each point, this will be extremely useful!

```
// phi is the 'latitude'
// theta is the 'longitude'
// see https://en.wikipedia.org/wiki/
   Spherical_coordinate_system
var phi = (90 - lat) * Math.PI / 180;
var theta = (180 - lng) * Math.PI / 180;

// each point coordinate is positioned using a
   combination of trigonometric functions
// again, see https://en.wikipedia.org/wiki/
   Spherical_coordinate_system
```

```
point.position.x = 200 * Math.sin(phi) * Math.cos(
    theta);
point.position.y = 200 * Math.cos(phi);
point.position.z = 200 * Math.sin(phi) * Math.sin(
    theta);

// remember that point was declared before as a Mesh
    object, thus it is declared as an actual 3D object
    in the world.
```

- They rendered the texture in the fragment shader, for which I will place a snippet of here. What I find interesting is that they render the texture on the shader, and not using `THREE.js`, which provides a way of placing textures on a sphere. I analysed the Javascript code, and I noticed that their implementation of this is actually small. All it does is load the texture, and draws it using the fragment shader (all in two lines of code!). This gives me hope, perhaps the actual texturing job of the project may not be as difficult as I thought.

  ```
  uniform sampler2D texture;
  varying vec3 vNormal;
  varying vec2 vUv;
  void main() {
    vec3 diffuse = texture2D( texture, vUv ).xyz;
    float intensity = 1.05 - dot( vNormal, vec3( 0.0,
        0.0, 1.0 ) );
    vec3 atmosphere = vec3( 1.0, 1.0, 1.0 ) * pow(
        intensity, 3.0 );
    gl_FragColor = vec4( diffuse + atmosphere, 1.0 );
  }
  ```

  As can be noted, they sort of 'diffused' the world texture along with the atmosphere. My initial idea was to render a sphere, map a texture on the sphere and create another transparent sphere which acts as the atmosphere. This is not what Google decided to do, and instead combined the two into one sphere. I'm not sure what to take of this, and I just think that they are two different implementations of the same idea, and I don't think there is much benefit of doing it all in one sphere (I want to elevate the atmosphere a slight bit).

- The user can 'drag' the sphere around with their mouse, this is definitely something I want to take notes in and integrate into my project.

1.3 Prospective Users and Client

The goal of this section is to provide an insight as to my prospective users, with an example client.
The kind of prospective users I seek, are mainly students, potentially geologists or cartographers. My project allows them to rotate a virtual globe, and have a from-space view to the landmasses and oceans. Furthermore, they can click on countries, allowing them to read some useful data about the country. They can also try and test their knowledge of country borders, and try and guess which country they are clicking, to test their accuracy.

1.3.1 Client

My end user and client, Will, is a knowledge source for geography. He thought that my project idea was interesting, and that he would definitely enjoy testing his knowledge of country borders, and comparing the data of countries depending on their location and position of the globe. Me and Will discussed about some potential objectives for my project:

- **Gamification**
  Will and I discussed about a way of gamifying my project, so as to make it more widely available for people who want to be entertained. He thought that creating a quiz would be interesting, for an addition to the user interface to contain the country name, and that the player had to find the country to win.
  In my opinion, this would definitely be a great addition to the project. However, I don't think that it is a very necessary addition.

1.4 Solution Proposals

The goal of this section is to bring all that I have researched, and apply and transform these ideas onto my project. This research was essential and massively beneficial, and I have learned a lot of ideas. While researching, I was able to map these ideas onto my project in a high-level manner.

1.4.1 Rendering of the globe

This is crucial problem that I **had** to find a solution for. I needed to find an efficient, simple to implement method for rendering my globe. Thankfully, with the current and previous systems analysis, most of which use the same `THREE.js` framework which I will be using, I think that I have a good overview of what I need to do. Firstly, I must find an appropriate solution for the most basic and strapped down idea, which is actually displaying the sphere. After some research into `THREE.js`, this is what I have noted.

In `THREE.js`, objects are declared in three steps.

1. Firstly, the geometry of the object must be initialized. The geometry of an object is essentially data about the shape of the object. Since I am creating a globe, I should obviously use a spherical geometry.

2. Secondly, the material of the object must be declared. The material of an object is exactly what it describes. It is quite an arbitrary term, however I have found it to be defined as what will be displayed on the *surface* of the object. This is a crucial step, as I need to somehow map the world, the landmasses, and the oceans on the surface of the sphere. From previous projects, each one does this step differently. For example, one of them doesn't map the texture on the material, and instead uses a fragment shader to colour each pixel the same colour as the texture. One other project may use the `THREE.js` '`MeshBasicMaterial`' which already allows for texture mapping as a parameter.
   I think, that for my project I should keep it simple and do it the second way, which is simply using the already present `THREE.js` method. However, if I should have some time to spare and to improve the graphics of my project, I may switch to using a shader as it provides much more control.

3. Thirdly, and finally, the mesh of the object is declared. The mesh is the final step to the object and contains data about the position of the

object, it's orientation, and other physical attributes. It combines the material and geometry of the object and can finally be displayed on the screen. This is a simple task in `THREE.js` and can easily be done in a single line of code. Further on in the code, for example, if I need to rotate the sphere, I can use the mesh of the object's rotation property to rotate it.

1.4.2 User Input

This is another crucial problem for which I must prepare for. User input is a crucial part of my project as it is how the user will interact with the project. For the most part, I don't think that I will require any keyboard input, as all of the functionality can be available with just the mouse; this is useful as I can just focus on mouse input, keeping my project simple. Here are some proposals on how I may solve this problem:

- The sphere must rotate on some form of input. Previously, where I analysed some real examples of current systems, I noticed that most of them do this by using mouse dragging. When the mouse button is held and moved, the sphere rotates accordingly. This is the method of input I will be using for my project. The way I am thinking of approaching this problem, is by registering some sort of 'event' when the user holds down the mouse button. When the mouse button is held down, the sphere can rotate along with the mouse. I should experiment with the sensitivity of which the sphere rotates, so as to not make it too slow at rotating, or too fast.

- When a country is clicked, an information pop up should appear. Keeping it in topic, I will focus on the mouse input for this bullet point. The way I am thinking of approaching this, is by sending out some sort of an invisible 'ray' on a mouse click. When this 'ray' intersects a country, it should display the appropriate information about the country. This will be quite a complex problem, and will definitely require some experimentation and deep algorithms / maths further on.

1.4.3 API requests & country information

This is another crucial problem for which I should pre-emptively solve. The basic overview of this problem is that I need to create a method of requesting the API, formatting the data, and displaying the formatted data on user input. I will split this problem into steps:

1. **API requests**
   During previous research into APIs, I concluded that there were two main APIs that I will use to request and gather the information needed to be displayed. To put it in simple terms, I need to send a request to a server, which will in turn send my desired data back to be processed. This is quite a simple problem, and thus I don't think that it will require much thought, it is however quite important that I get this right.

2. **Formatting the data**
   We need to format this data which is sent back to us in a manner where the user can read it, and that the developer can also easily manipulate and store the data. This problem is quite API specific, but from what I have researched and through example requests, I think I can simply retrieve the data, and simply store it in a Javascript object. Each country has its own slight differences in data, and so I will need to deal with this accordingly. This data must then be formatted in a way that the user can read it, so numbers should be displayed in a readable format.

3. **Displaying the data**
   My thought towards displaying the data is to, on user input, create a dialogue box, near or next to the cursor. This can be done using `THREE.js` 2D objects. Inside the dialogue box, the data must be presented in a readable fashion. Overall, the dialogue box should fit with the theme of the project, and so must the data.
   The data could be presented in a multitude of ways, such as small graphs inside the dialogue box or text. This will be something I should think about later in the project, but however is not very high priority. Initially, I think that I will simply just render text inside the dialogue box.

1.4.4 Formatting

This is an 'optional' step, once I have the data layed out from the API queries, I can work on formatting the data so the user can see the data. I want this to be as user friendly as possible, so that anyone can use the website, and understand the data being displayed. I have some ideas layed out for this:

- **User Interface** I think a vertical bar on the right side of the screen, containing each bit of data, each item being layed out on top of another, would be a good final user interface I can work towards.

- **Graphs** This is an ambitious step, however having graphs would definitely be a plus, as the user can easily see how the data changes through time.

1.4.5 Graphics

Another 'optional' step, I can make the user input more user friendly and tactile if I add some sort of feedback to the globe.

- A little idea I have, which stems from the Github globe, is how they have lines coming to and from countries. For example, when you click on the globe, a line can stem from the position clicked to the country selected.

- Another interesting idea which I have thought of, is having two world textures, one where the countries are in greyscale, and one where the countries are in color - wherever the mouse hovers, a diffusion occurs between the two textures and the greyscale emerges radially from the cursor.

- I think that stars would be a great addition to the project, and to also make each of them blink. Since I need thousands of stars, I need to think about how I can optimize the large amount of stars, so as to not slow down the webpage.

1.5 Modelling of Problem

My project can be modelled in a series of connective flow diagrams:

1.5.1 Rendering & Animating

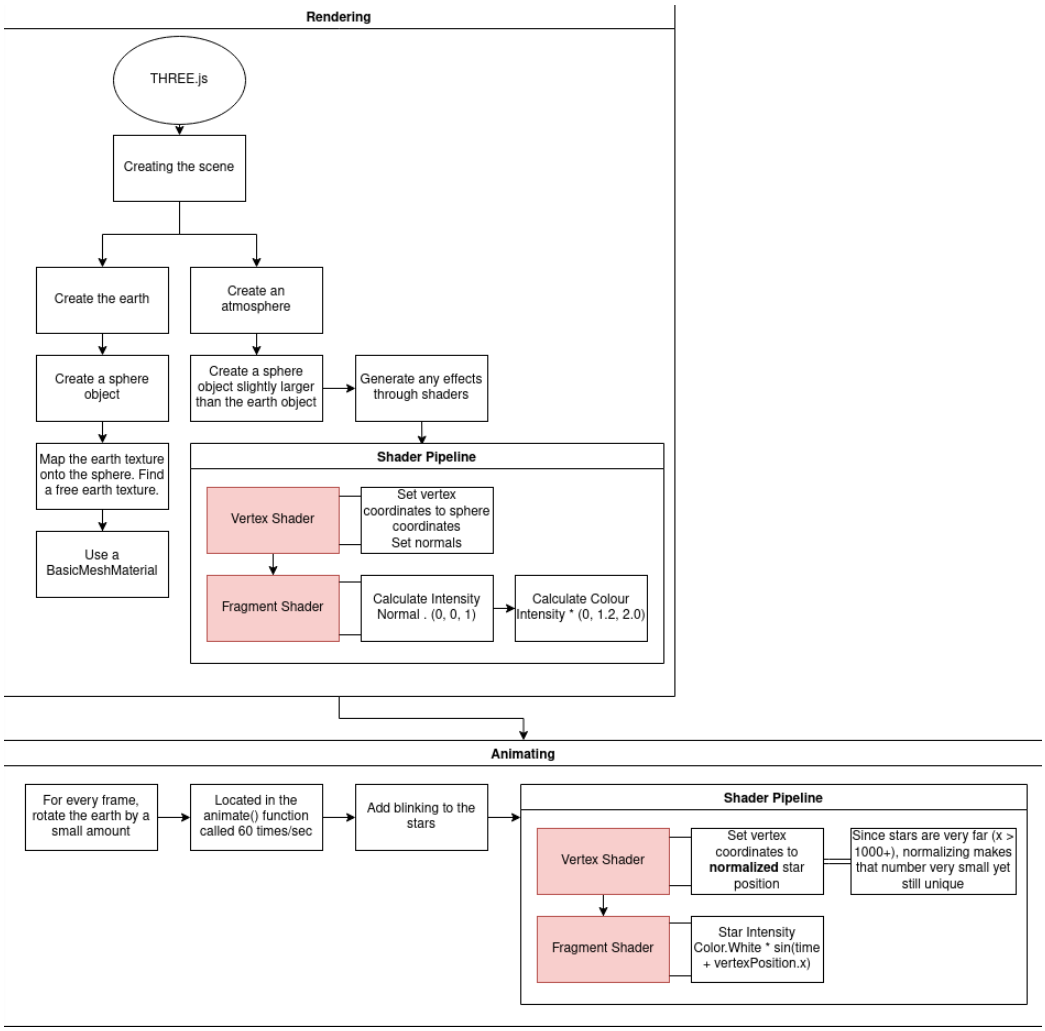


Figure 1.4:

1.5.2 User Input

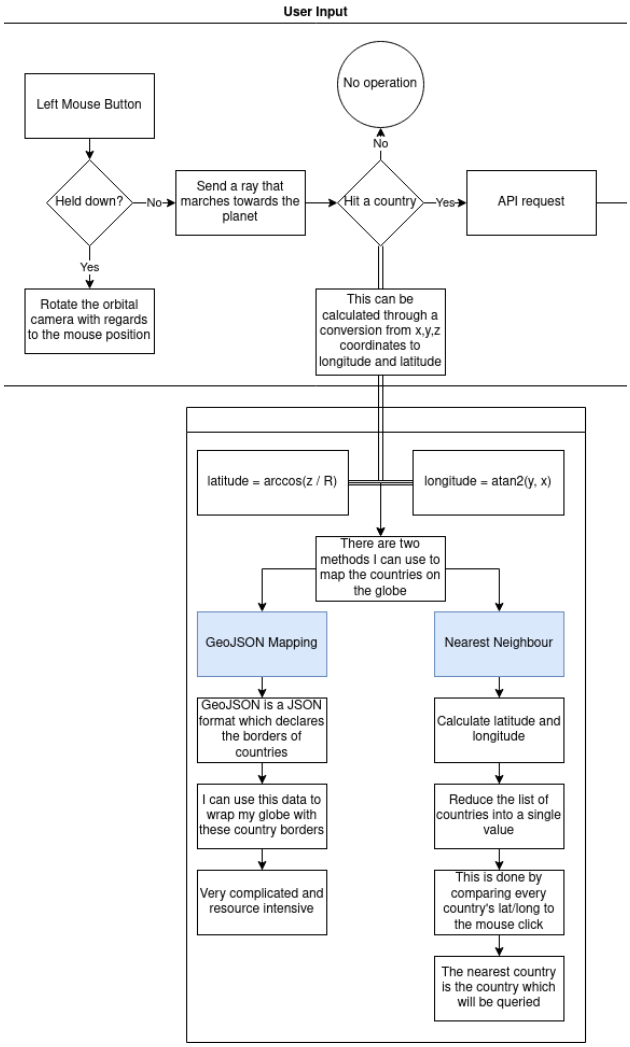


Figure 1.5:

1.5.3 API Querying

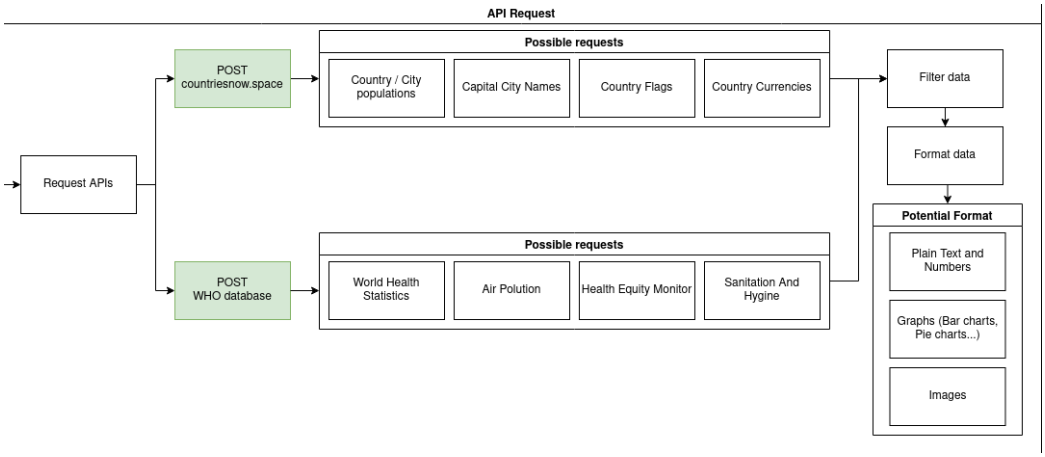


Figure 1.6:

1.5.4 Conclusionary Model

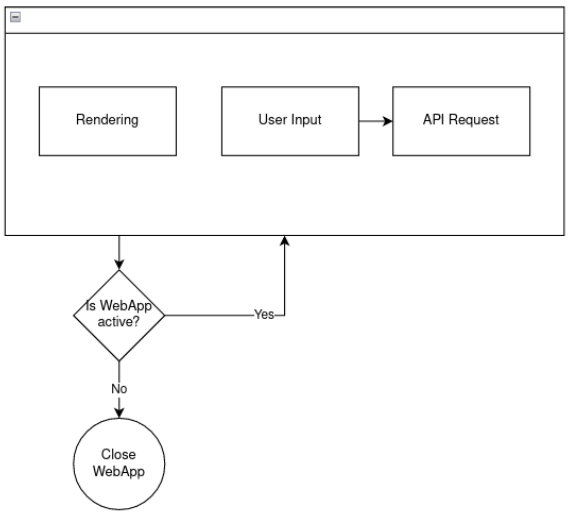


Figure 1.7:

1.6 Critical Path of Stages

Each objective can be ranked and sorted by importance:

- **Rendering the Globe**
  This is the core of the web application, without the globe, the web application would be bare. Thus, this is the most important step, and the first step to be completed.

- **User Input**
  User input controls whether the user can rotate and see the entirety of the globe, and it also dictates which country is selected, which can then be sent off as requests to the suitable APIs. Therefore, this is the second most important objective that needs to be completed.

- **API Requesting**
  Ultimately, and also importantly, the selected country must be sent to one or many API servers to retrieve useful data. This is not so simple, as there are also other steps involved:

  - **Formatting**
    The API response must be formatted so as to be readable to the user.
  - **Displaying**
    The API response must then be displayed to the user. This section will be written in HTML and CSS.

- Additional Objectives:

  - **Graphical Improvement**
    Eye candy such as stars. Improvements to the API UI in general.
  - **Optimization**
    Depending on how slow our webpage is, we may need to apply some layers of optimization to make our webpage faster, and more responsive.

1.7 Objectives

Through the research I have compiled, I can determine the main future objectives that I have to work towards.

1.7.1 Main objectives

- **Rendering** I obviously need to render the globe, that is of upmost importance, the whole web application *is* based on the globe after all.
- **User Input** The next main objective involves user input - I need the user to be able to move the globe around, and be able to select countries using the mouse.
- **API Querying** I need the user to be able - depending on the selection - to recieve information about the country. This involves API querying, JSON, text formatting and CSS & HTML.

1.7.2 Additional & Further objectives

When I overcome the challenges of the main objectives, I can then focus on working towards extra, optional objectives.

- **Formatting** When I have a basic, simple UI which displays some information about the country which is selected, I can start displaying more data.
- **Graphics** I can also work on making the globe and the interactions have greater user feedback by using graphics. For example, a line which connects to the country to signal that you have clicked it. I think some stars would also make the project look more inviting.
- **Graphs** If time is available, I can work on converting simple number data, into graphical data with plots. Since the APIs provide a wide range of time for each data point, I can plot the data.

1.8 User Requirements

A reason that I picked a webpage based approach to the entire project, was that webpages usually load the same for all computers. A problem which arises is old web browsers such as Internet Explorer, which do not support a lot of the recent Javascript and CSS versions. Since my project does not use complex CSS, I don't expect that to be a problem. However, I need to be aware of bugs that may appear on older web browsers, and test my webpage on a multitude of web browsers.
In conclusion, there is no user requirement, except for a **web browser**.

Chapter 2

Design

2.1 High Level Overview

The goal of this section is to provide a high level, top down overview of the entire project. I have combined all the ideas that I have taken from the research, and compiled them down into many steps in order of priority, including algorithms and pseudocode snippets for which I may look back upon and integrate into the real process.
From what I have compiled, I have created an overview from the compilation of these ideas.

The Project
The project which I will be developing is a web based application, which contains a well presented and realistic globe representing the earth, allowing for a satellite view of the globe. The globe will appear suspended in space, upon which it can be manipulated using the mouse, where it can be rotated, moved around, and interacted with. These interactions may do many things. The dragging of the mouse, for example, will rotate the sphere, the hovering over countries will highlight them, allowing them to be clicked. Upon a mouse click, a country is selected, and useful information about the country is displayed, allowing students in analytics courses or people with interests in country data to note, all displayed in a unique manner, allowing for a unique experience. Not only may it be useful for students, or people in data science; people whom appreciate the beauty of the earth may enjoy toying around with the globe.
I was partially inspired by similar projects, such as Google Earth, I thought that the way the globe was represented was interesting, and thought that I may undertake in a similar project, adding my own unique twist.

2.2 Description of Algorithms

The goal of this section is to compile some algorithms written in pseudocode format, it contains a code based approach to the project problems.

2.2.1 Rendering of the globe

Before rendering the globe, we must set up the scene for which our globe can be displayed in, and we must define how the camera will see our object:

```
GlobeScene = Scene()
Camera = PerspectiveCamera({
  field of view = 75 degrees,
  aspect ratio = WINDOW WIDTH / WINDOW HEIGHT
})

// The renderer is the HTML component where our sphere can
   be seen
Renderer = WebGLRenderer({
  size = WINDOW WIDTH, WINDOW HEIGHT
})
ADD Renderer TO HTML PAGE
```

The globe will be rendered in many steps, for example, the earth's sphere will be displayed, then an atmosphere will be layered, then perhaps clouds will be layered in between the atmosphere and the earth, etcetera.
All spheres will be initialized the same way. Taking inspiration from how `THREE.js` declares 3D objects, I have a vague idea on how this will be done:

```
Radius = 50
WorldTexture = LoadTexture("world.png")
GlobeGeometry = SphereGeometry(Radius)
GlobeMaterial = Material({
  texture = WorldTexture
  colour = NO COLOUR
})
Globe = Mesh(GlobeGeometry, GlobeMaterial)
Globe.position = 0, 0, 0
ADD Globe TO SCENE
```

The atmosphere will be another sphere, layered on top of the globe, with a slightly larger radius. The atmosphere will be rendered slightly different from the globe. Since it changes depending on where you look at it, it would be appropriate to use a shader that will colour the pixels in a way that looks

like an atmosphere.

Atmosphere Sphere Code

```
Radius = 60
AtmosphereGeometry = SphereGeometry(Radius)
AtmosphereMaterial = ShaderMaterial({
  shader = LOAD SHADERS
  colour = NO COLOUR
})
Atmosphere = Mesh(GlobeGeometry, GlobeMaterial)
Atmosphere.position = 0, 0, 0
```

Atmosphere Shader Code

This code won't be trivial. We need to think of a way of rendering an atmosphere. Firstly, what does an atmosphere look like? Let's think about this like a programmer - an atmosphere can be rendered like a transparent sphere, where a slight blue tinge can diffuse outwards towards the edges of a sphere. In short, from the eyes of a person, the sphere gets more and more transparent towards the middle of the sphere. Therefore, the sphere gets updated so that the middle looks transparent wherever you are looking from. This is where shaders come in handy, because we can work directly on the pixels procedurally, and also because we have the vertex data of the sphere, allowing us to know where all the points are on the sphere, and apply a colour to the pixels in each point depending on where the point is.

Let's tackle the problem of transparency. We want the blue tinge to slowly and gradually become lighter and more transparent towards the middle of the sphere, but first, we need to know how close to the 'middle' we are to the sphere. By doing some research, I have concluded that to do this, we need to know the **normal** of each point of the sphere. The **normal** of a point is the perpendicular vector from the surface of the object (in this case a sphere), here is a visual of what a normal is[1]:

The 'N' represents the **normal**. Each vertex of a sphere has a normal

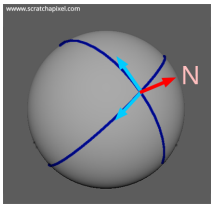


Figure 2.1:

[1]https://www.scratchapixel.com/lessons/advanced-rendering/rendering-distance-fields/basic-sphere-tracer

vector, pointing *away* from the sphere. This is exactly what we want, because normals that are pointing *towards* us can be measured. Think of a sphere, the normals at the very edge of a sphere will point exactly left or right, where as points directly in the middle of a sphere will point towards you. For example, in the figure above, the normal is pointing slightly towards us. But how do we measure how much the normal is pointing towards us? We need to use a vector operation called the **dot product**. The dot product combines two vectors into a single value, the resulting value tells you what amount of one vector *goes in the direction of another vector*. For example, lets say we have a box and an inclined ramp, and we push it **up the ramp**. The box has a *horizontal* component and *vertical* component to the force vector. So the dot product in this case represents the total amount of force going in the **direction up the ramp**.
How can we integrate this knowledge into our shader code however? We need to find how much the normal to the sphere points towards us. From the camera lens, forwards is the **z axis**. Therefore, if we use the dot product against the normal and a vector where the x, y components are zero and the z component is, lets say, 1, then the resulting value will tell us how much the normal points towards us!

$$Normal \cdot \begin{bmatrix} 0.0 \\ 0.0 \\ 1.0 \end{bmatrix} = \text{Measure of how strongly the vector points towards camera}$$

We can now use this formula and multiply the result by the colour of the atmosphere that we want, in this case, a slightly cyan colour. This is the pseudocode I came up with:

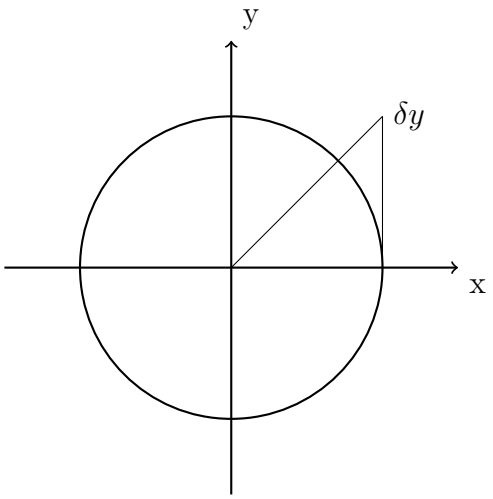
```
Intensity = DotProduct(Normal, Vector(0, 0, 1))
Atmosphere = Vector(0, 1.2, 2.0) * Intensity
Pixel Colour = Vector(Atmosphere.XYZ, 0.5)
```

2.2.2 User Input

After the globe is rendered, we can handle our user input. We are focusing primarily on mouse input, as I feel like keyboard input is not necessary.
Firstly, we need to move our globe. Since the globe is a sphere, I am thinking of implementing a simple orbit control system.

Orbital Controls - Rotation

Orbital controls involve clamping the camera into a sphere. Thus, the camera cannot move outside the edges of the sphere.
Let's think of this as a two dimensional circle first:



We need to clamp the **magnitude** of the vector δy to be equal to R, the radius of the circle. Therefore:

$$\frac{R}{\|\vec{\delta y}\|} \cdot \vec{\delta y} = \text{Clamped position}$$

Orbital Controls - Zoom

Zooming is simpler, all we need to do is add a fixed constant to our radius of the sphere, and update the camera on the scroll wheel.

$$R = R + \delta R$$

Country detection

There are two ways I can go about selecting a country on click:

- **Country Mapping** Involves mapping a GEOJson file / database to the globe, which includes the borders of each country, which then complicated math and required optimization allows for selection of a country.
- **Nearest Neighbour** Involves converting click coordinates into suitable latitude / longitude coordinates, which then, iterating through each country (through a JSON file containing each countries name and their respective latitude & longitude coordinates) and finding the closest match using an algorithm.

I think that the first way is definitely more accurate, and more interesting, however, I cannot think of an algorithm in my head which can quickly do this, without involving some seriously complex formulae. The second method is less accurate, however, I think that it is good enough, and I can adjust it for more accuracy later on. Firstly, we need to convert our mouse coordinates (X, Y, Z) to latitude and longitudinal coordinates. [2] XYZ coordinates can

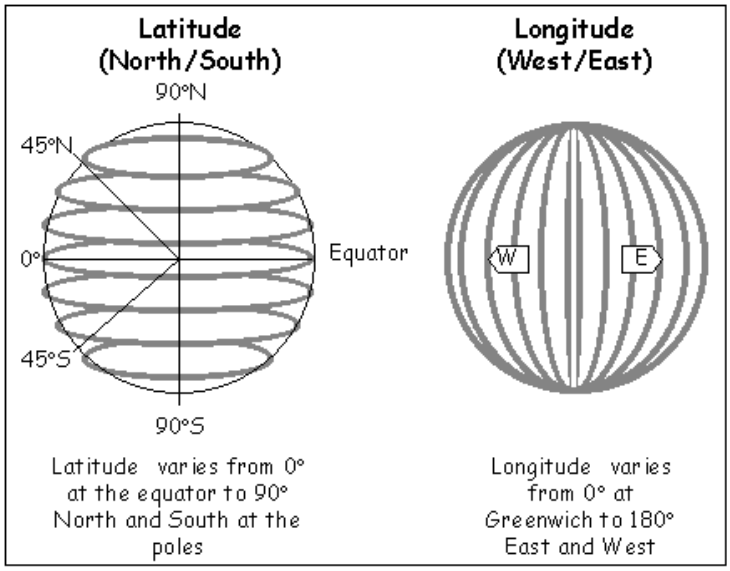


Figure 2.2: Latitude and Longitude

simply be converted to latitude and longitudinal coordinates through this formula:

$$\text{lat} = \arcsin(y)$$

$$\text{lng} = \text{atan2}(z, x)$$

Where x, y, z are **normalized** coordinates. We can convert this into pseudocode:

```
lat = asin(y)
lng = atan2(z, x)
```

Now that we have the coordinates in latitude and longitude, we can run the algorithm:

[2] https://blogs.sap.com/2018/06/15/preparing-latitude-longitude-data-for-showing-in-sap-lumira-and-sap-analytics-cloud/

Nearest Neighbour

```
lat = asin(y)
lng = atan2(z, x)

// Latitude cannot be greater than 90
// Longitude cannot be greater than 180
minLat = 90
minLng = 180
countryName = ""

Foreach country in jsonToArray("countries.json") Do
    // Difference in lat & lng from the mouse click and '
       country'
    diffLat = country.latitude - lat
    diffLng = country.longitude - lng
    If diffLat < minLat and diffLng < minLng Then
        minLat = diffLat
        minLng = diffLng
        countryName = country.name
    End If
End Foreach
```

When the algorithm is completed, the final `minLat` and `minLng` coordinates are the coordinates of the country, and `countryName` is the name of the country, provided that the `countries.json` file contains JSON formatted like this:

```
[
    {
        "name": "Algeria",
        "latitude": 28.0339,
        "longitude": 1.6596
    }, ...
]
```

Now that we have the name of the country, we can finally, query the api with the name of the country.

2.2.3 API Querying

Since we now have access to the name of the country selected through the algorithm, we can query the API. Depending on the `countries.json` file layout, each country data table may include a `name` (the name of the country) or may include an ISO Alpha-2 code for example `AF` for 'Afghanistan'. The

benefit of picking a json data file with Alpha-2 codes is that Alpha-2 codes are more generalized - each country has a unique Alpha-2 code, which is the same everywhere. This is not the same as a country name, which may differ from codebase to codebase.
I have researched many free APIs and Banks to compile some of the more interesting, and must have data for API Querying.
countriesnow.space

- Country Population

  ```
  Send Post Request to "https://countriesnow.space/
      api/v0.1/countries/population"
  With Json '{ "iso2": (alpha-2 code) }'
  ```

- Country Flag

  ```
  Send Post Request to "https://countriesnow.space/
      api/v0.1/countries/flag/images"
  With Json '{ "iso2": (alpha-2 code) }'
  ```

- Capital City

  ```
  Send Post Request to "https://countriesnow.space/
      api/v0.1/countries/capital"
  With Json '{ "iso2": (alpha-2 code) }'
  ```

- Country Currency

  ```
  Send Post Request to "https://countriesnow.space/
      api/v0.1/countries/currency"
  With Json '{ "iso2": (alpha-2 code) }'
  ```

The World Bank
The World Bank's API works on `GET` requests, making our life simpler, as we don't need any input Json.

- Country GDP (most recent value)

  ```
  Get Json from "http://api.worldbank.org/v2/country
      / (alpha-2 code) /indicator/NY.GDP.PCAP.PP.CD?
      mrv=1&format=json"
  ```

- Population Growth (most recent value) (annual %)

  ```
  Get Json from "http://api.worldbank.org/v2/country
      / (alpha-2 code) /indicator/SP.POP.GROW?mrv=1&
      format=json"
  ```

2.3 Description of Data Structures

The goal of this section is to provide a description of the kinds of data structures which I will use in my project. My project does not use many complicated data structures, however it makes use of the `JSON` format, which in itself is a data structure. I also extensively make use of many `THREE` data structures.

2.3.1 JSON

The `JSON` format in itself is a data structure. It is made of multiple components:

- Numbers
- Strings
- Booleans
- Arrays
- Objects

The more complicated components are arrays and objects. Arrays represent a list of components, however objects represent a hash table. A hash table contains pairs of keys and values. In this case (`JSON`), keys are strings, and values are components. In a hash table, keys get mathematically converted into a hash number in O(1) time for lookup. Therefore, hashtables are very fast at looking up keys. This makes `JSON` a great data structure to hold hundreds of countries, as it is fast and readable.

2.3.2 THREE.js

`THREE.js` also contains some crucial data structures which are used for mathematical computation, and as buffers to hold hundreds of values to be sent off to a shader pipeline.
A common data structure used in `THREE.js` and shader pipelines are **vectors**. They simply store three floating point values, but in themselves are used to store positions or colors. `THREE.js` provides dozens of functions that manipulate vectors.
Another common data structure used in shader pipelines are **matrices**. Matrices are simply two dimensional arrays of equal width and length. Matrices are used to store object transformations in a fast and space efficient way. They can also be transformed, rotated, scaled.

2.4 User Interface

The goal of this section is to visually represent my thoughts about creating an user interface.

2.4.1 The Globe

The main attraction of the webpage is probably the globe. The globe in its most simple form, is a sphere, with a world texture mapped onto it.[3]

Figure 2.3: The globe

The globe will initially strike too much contrast between the blues and greens with the black of space. Thus, I think adding an atmosphere would improve the look of the globe. Optionally, I can also include clouds, using a texture with a transparent background, or some sort of cloud forming algorithm.

Figure 2.4: The atmosphere and clouds

[3]https://unsplash.com/s/photos/globe

2.4.2 Input Feedback

Input feedback is important, as with no user feedback, such as no change on click, the user may potentially think the webpage is slow or broken. To prevent this, I think that on click, a pin drops on the location clicked, much like Google Maps.

Figure 2.5: The pin, circled in red

This could be simplified as just a simple, small red sphere on click, which fades away over time.

2.4.3 API Interface

On click, when a country is selected, the country selected will be sent out to many APIs to retrieve information such as GDP, Population and Currency. I need to format this in a way that the user can easily understand the statistics, and can see them change on a country click.
I think that a nice, and simple UI works here, such as a vertical bar on the left or right side of the screen, which simply displays the information in a vertically stacked layout.

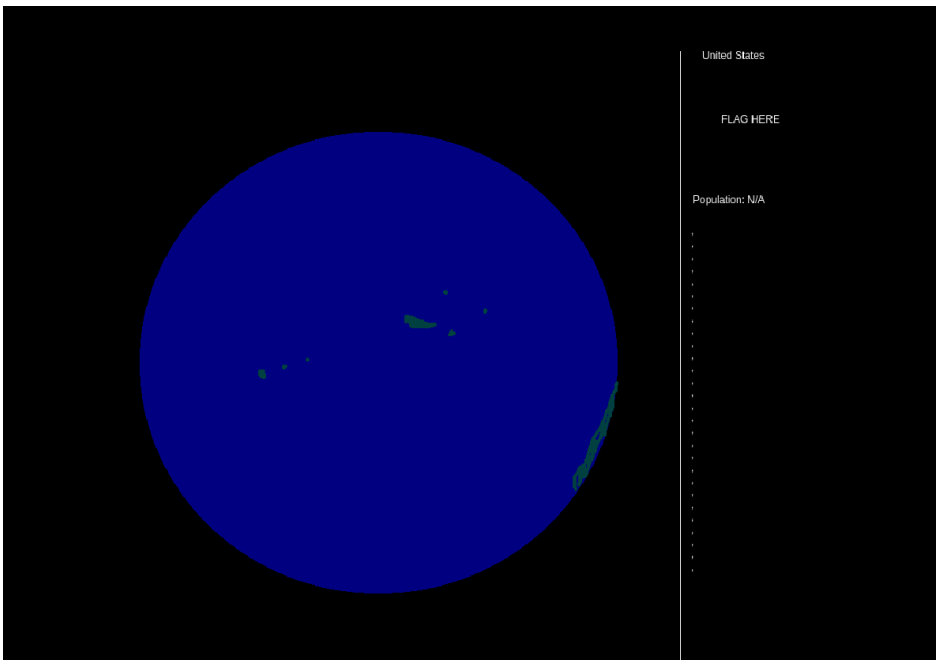


Figure 2.6: The simple layout

2.5 System security and integrity of data

Since my project does not contain the handling of passwords or hidden data, it contains the plus of already being secure. Furthermore, I made sure all data being passed around is public data, and that it is impossible for private data to be spilled. In fact, I would probably open source this project, that's how confident I am with its security. The only possibility of some security breach was if I were to call an API with some API key, however, I made sure that was not the case as I planned to use specifically public APIs.

Chapter 3

Technical Solution

3.1 Set up

Since I am developing a dynamic website, using javascript modules is against the CORS policy, [1] therefore I have decided to use an npm (javascript package manager) package[2] `live-server` to host my website on a live server. It is a command line application, and when ran using `live-server` simply searches for an `index.html` file and hosts it using an IP address with a port. By default, it hosts on `localhost:8080`, allowing for easy debugging of your web page.
Firstly, let's create a simple project structure to host our project. Initially, I created a basic `index.html` file. Since most of our graphical interface will be written in javascript, all I have to specify is the import of the actual javascript file, and some extra tags.
The `<script type="module" src="js/main.js">` tag imports the `main.js` file in the `js` folder.
I set the title of the project as seen by this line: `<title>Globe</title>`, and then disabled any sort of margin on our page by the CSS within the `<style>` tags.

[1] https://stackoverflow.com/questions/52919331/access-to-script-at-from-origin-null-has-been-blocked-by-cors-policy

[2] https://www.npmjs.com/package/live-server/v/0.8.0

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Globe</title>
    <style>
      body { margin: 0; }
    </style>
  </head>
  <body>
      <script type="module" src="js/main.js"></script>
  </body>
</html>
```

js/main.js

```
console.log("Hello world");
```

After running `live-server`, the output in the web browser is:

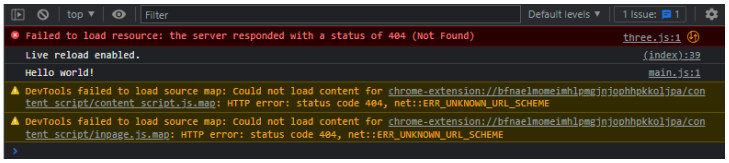


Figure 3.1:

3.2 THREE.js

Now that I have set up a basic frame upon which to build our application on, we need to import `THREE.js`, the graphical framework we will be using referenced in the Analysis section. `THREE.js` is, fortunately, very simple to set up. All we need to do is import the javascript file in our `main.js` file. Since `main.js` is a module, we can easily import other javascript modules from the javascript file. However, before importing the file, we need to install the `THREE.js` library. Since it's a single-file library, all we need to do is navigate to their website[3] and click the `download` button, and import the file downloaded.
It's easy to import files in javascript, this is our javascript file currently:

js/main.js

```
import * as THREE from "./three.js";
console.log("Hello world");
```

As an overview, the `import` statement allows for importing files, we use the `*` wildcard to import **everything** from `three.js` and alias it under a namespace `THREE`. Code further ahead will use this namespace.
We get the same output, and we get no errors, which means that `THREE.js` has been imported successfully.

[3] https://threejs.org

3.2.1 Setting up the scene

[4] The goal of this section is to set up the scene for our globe page, as a brief example, I will create a spinning sphere as the basis for our graphical web page. Firstly, let's actually *create* the scene:

```
const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera(75, window.
    innerWidth / window.innerHeight, 0.1, 1000);

const renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);
```

The `scene` variable, contains information in regards to the actual *scene*. Since we are looking at our globe from a perspective, we initialize the `THREE` class `PerspectiveCamera`.
The first parameter is the *field of view*, the extent of the scene which can be seen on display at any moment. I set it to 75°.
The second parameter is the *aspect ratio*, which determines how wide the pixels are on the screen. For mostly all monitors, the aspect ratio is defined:

$$Aspect\ Ratio = \frac{Window\ Width}{Window\ Height}$$

The fact that `window.innerWidth` or `window.innerHeight` are variable, means that as the window is resized, so will the scene.
The next two parameters are the `near` and `far` clipping plane. Which creates a interval at which objects won't be rendered. For example, objects further than `far` will not render, and objects closer than `near` will also not render. For now, I don't exactly know how big the globe will be, and so I used arbitrary values 0.1 and 1000 for `near` and `far` respectively.

Now, we need to define a renderer to render the scene. `THREE.js` comes with many renderers, however I chose to use the most conventional renderer, the `WebGLRenderer` class. It uses WebGL, a web version of OpenGL, to render objects. I want the renderer to fill the entire screen, and so I chose for it to render at maximum width and height.
Finally, we want to add the renderer to our HTML document. Behind the scenes, `THREE.js` uses a `<canvas>` tag to render everything.

[4] https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene

3.2.2 Populating the scene

[5] Now that we have created a scene, let's define some objects to populate the scene. I am going to populate the scene with a spinning sphere. `THREE.js` makes this task very simple:

```
const geometry = new THREE.SphereGeometry(5);
const material = new THREE.MeshBasicMaterial({color: 0
    xff0000});
const sphere = new THREE.Mesh(geometry, material);
scene.add(sphere);

camera.position.z = 10;
```

The `geometry` variable contains all the vertices and faces of the sphere. `THREE.js` has many simple shape geometries that can easily be created, and for this case we need a sphere, so we use the `SphereGeometry` class with initial radius of 5.
The `material` variable apply properties to a certain geometry, and allows us to colour the faces of our sphere with any colour of our liking, as well as allow for more complex materials such as images. `MeshBasicMaterial` is a basic interface class which allows us to specify basic properties. I want the sphere to be red, and so we pass an empty table to `MeshBasicMaterial` with an attribute `color` that contains a hex colour `0xff0000`. Hex colours work by dividing a 24 bit number into 3 parts, the red, the green, and the blue parts. Each part can have a range from 0-255 e.g `0xff = 255`. For our sphere to render in red, we want to only colour the red part of the hex colour. If we split up our hex colour it would look like this:

FF	00	00
R	G	B

Now we can actually create the object. Objects in `THREE.js` are called 'Meshes'. The `sphere` variable contains a mesh taking our `geometry` variable and our `material` variable respectively. The reason we define the geometry and material variables is so that we can modify them later when we want to animate our sphere.
Finally, we add the sphere to the scene. However, since the camera is placed at `(0, 0, 0)`, we want to shift the camera backwards so that we can actually *see* the sphere. I moved it to the point `(0, 0, 10)` by changing `camera.position.z` to 10.
However, we are still not done yet, since we have to render and animate our scene.

[5] https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene

3.2.3 Rendering and animating the scene

[6] Now, we can finally render our scene. `THREE.js` makes this task simple and quick. Conventionally, we define a function `animate` which will contain all the code we wish to repeat in the program loop. This is how we do this:

```
function animate() {
  requestAnimationFrame(animate);
  renderer.render(scene, camera);
}
animate();
```

The `animate` function creates an infinite loop which redraws the scene every time the *screen* is refreshed, usually 60 times per second. The `requestAnimationFrame` function takes in the `animate` function and actually creates this loop, which is a nice abstraction.
Then the renderer renders our scene with our camera, and finally, we call the `animate` function to actually run this snippet of code.

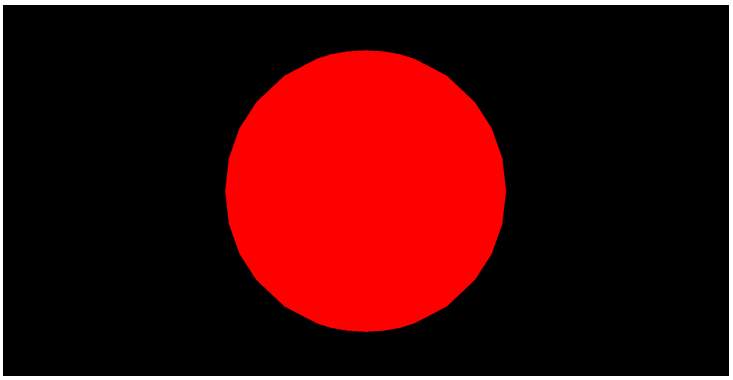


Figure 3.2:

Finally, our sphere can be seen on the screen! However, we have created no way for it to rotate.
Note that our `animate` function gets called 60 times per second. This is where we want to place our rotation code.

[6] https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene

Here's our updated `animate` function:

```
function animate() {
  sphere.rotateY(0.01);

  requestAnimationFrame(animate);
  renderer.render(scene, camera);
}
animate();
```

`sphere.rotateY(0.01)` rotates the sphere in the y axis by 0.01 radians every $\frac{1}{60}$th of a second. `THREE.js` rotations look like this: [7]

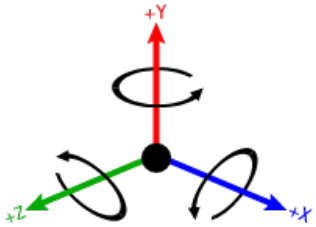


Figure 3.3:

It's quite hard to see on the screen due to the fact the sphere is one solid colour, however we can set the `wireframe` attribute in the `MeshBasicMaterial` class to true like so. It allows us to see only the outlines of the sphere:

```
const material = new THREE.MeshBasicMaterial({
  color: 0xff0000,
  wireframe: true
});
```

Our sphere now rotates anti-clockwise on the y axis, and looks like this:

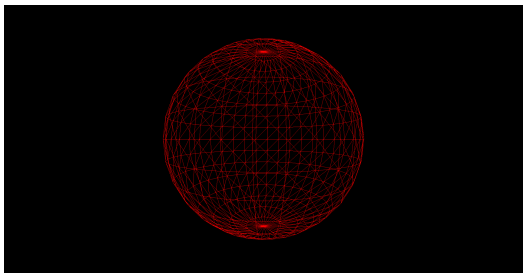


Figure 3.4:

[7] https://cdn.kastatic.org/ka-perseus-images/d24dd08a0ea7aaeeaa90d84f642e12998df3ffe7.svg

3.2.4 Overview

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Globe</title>
    <style>
      body { margin: 0; }
    </style>
  </head>
  <body>
    <script type="module" src="js/main.js"></script>
  </body>
</html>
```

js/main.js

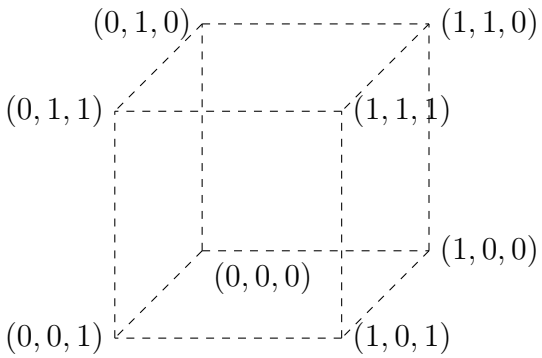
```
import * as THREE from './three.js';

const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera(75, window.
    innerWidth / window.innerHeight, 0.1, 1000);

const renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

const geometry = new THREE.SphereGeometry(5);
const material = new THREE.MeshBasicMaterial({color: 0
    xff0000, wireframe: true});
const sphere = new THREE.Mesh(geometry, material);
scene.add(sphere);

camera.position.z = 10;

function animate() {
  sphere.rotateY(0.01);

  requestAnimationFrame(animate);
  renderer.render(scene, camera);
}

animate();
```

3.3 WebGL Shaders and GLSL

WebGL is the way `THREE.js` interacts with graphics. We enabled WebGL rendering earlier by setting our `renderer` to a `THREE.WebGLRenderer`. It allows for graphics to be ran on the GPU instead of the CPU, allowing for huge performance boosts over CPU drawing. In essence, WebGL is a javascript API to allow users to draw graphics on the screen with maximum performance. Performance is important because if we want to draw a complex scene 60 times per second, we need to be performant.
The reason we need to worry about this, is that for us users to create beautiful graphical effects efficiently is to communicate with the WebGL *shader pipeline*.

3.3.1 Introduction to shaders

[8] Now that we have created our sphere, I think it's time to turn it into a globe. To do this, we need to introduce shaders.
Shaders are little programs that **rest on the GPU**. They belong in a *shader pipeline*. [9].
The shader pipeline is a sequence of steps that WebGL takes when rendering objects. It consists of many steps, but to keep it simple, we will focus on two important steps which we can manipulate: vertex processing and fragment processing.
The way we render positions in WebGL is by taking many vertices and processing them, for example, a cube has 8 vertices, each with a position. A 1x1 cube may be defined like this in WebGL:



A vertex shader, therefore, performs manipulations on these coordinates.

[8] https://learnopengl.com/Getting-started/Shaders
[9] https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

A fragment shader, acts not on the coordinates, but on `fragments`[10]. A `fragment` is a collection of values, each representing a segment of an area of pixels. You can think of this as segments of the faces of the sphere. Therefore, a fragment shader is a little program to perform manipulations on the pixels themselves. This allows us to, for example, change the colour of the pixels themselves depending on the vertex of the object.
These are combined into a *shader pipeline*, where the vertex shader program is ran, then its output is *piped* into the fragment shader program.
GLSL[11] is the programming language in which we will write our shaders in. It's a C-style language, and is specifically designed for vertex/fragment shaders.

3.3.2 Shading

Let's convert our sphere from a `MeshBasicMaterial` to a `ShaderMaterial`. A `ShaderMaterial` allows us to add a shader to our object.
Firstly, I changed our `material` variable to:

```
const material = new THREE.ShaderMaterial({
  uniforms: {
    time: clock.getElapsedTime()
  },

  vertexShader: sphereShader.vertex,
  fragmentShader: sphereShader.fragment,
});
```

Let's take a moment to understand these changes, the `uniforms` attribute lets us pass in values from our javascript program to our shader files. That's what a uniform is, a sort of 'parameter' into the shader.
Now, we pass our user defined shaders into the `vertexShader` and `fragmentShader` attributes. We actually haven't defined `sphereShader`. Let's define this in a new file `shader/sphereShader.js`.

[10] https://www.khronos.org/opengl/wiki/Fragment
[11] https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language

shader/sphereShader.js

```
export const vertex = ``;

export const fragment = ``;
```

We will write our vertex shader in the `vertex` variable, and the fragment shader in the `fragment` variable. Now, we have some shaders. Let's try to open the webpage.

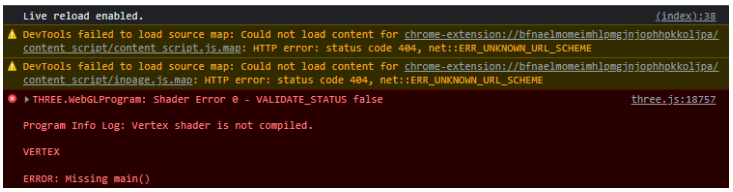


Figure 3.5:

We've ran into a problem. WebGL is telling us that our vertex shader has no `main()` function. The way GLSL works is that when a shader gets executed, WebGL calls the `main()` function, where all the calculations and manipulations are made. Let's define it for both shaders.

```
export const vertex = `void main(void) { }`;

export const fragment = `void main(void) { }`;
```

We define functions in GLSL like `type function_name(types...) { ... }`. The type of `main(void)` is `void`, which means the function does not return anything. If we were to define a function like `int foo(...) { return x; }`, x must be of an int type.
Let's open our page again.

Figure 3.6:

We get another error! The error describes us as not having any shader outputs. This is because both shaders, vertex and fragment must output something. In WebGL, the output variable is called `gl_Position` for vertex shaders and `gl_FragColor` for fragment shaders. With this information, let's try and draw our sphere again.

Let's begin with the vertex shader. Recall that the vertex shader takes many vertices and performs calculations on these vertices so that they 'project' onto our screen. After some testing, this is what I came up with:

```
varying vec3 vUv;
void main(void) {
  vUv = position;

  vec4 modelViewPosition = modelViewMatrix * vec4(position,
      1.0);
  gl_Position = projectionMatrix * modelViewPosition;
}
```

Let's break this down. Firstly, we define a global `varying vec3 vUv`. The `varying` tags the `vec3 vUv` as being part of the pipeline. This means that this variable `vUv` will be passed onto the next shader (the fragment shader). `vUv` is shorthand for the *coordinate of the vertex*, it will be useful later when we use it in the fragment shader.
After the `main(void)` definition, we set `vUv` as the `position` global, where `position` is the position of the vertex.
`modelViewMatrix` is a global 4D matrix (`mat4`) which contains all transformations, rotations and scaling required to put a vertex in its position. [12]
With this information, we know that we have to translate the vertex by the translation matrix. This is what an example `modelViewMatrix` looks like:

$$modelViewMatrix = \begin{bmatrix} ... & ... & ... & v_x \\ ... & ... & ... & v_y \\ ... & ... & ... & v_z \\ ... & ... & ... & 1.0 \end{bmatrix}$$

Where v_n is the position of the vertex in **relation to the model**. Because this matrix is relative, we need to turn it into a real world position on the screen. Therefore, we need to translate our existing `position` by the `modelViewMatrix`:

$$modelViewPosition = modelViewMatrix * \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

Where p_n is the coordinate of the 3D vector WebGL `position`.

[12] http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

Finally, we further translate our position by the `projectionMatrix`. Since we view the program through a 'camera', we need to further translate the real world position into a position that correlates to the screen. In other words, a vertex that happens to have $x = 0$ and $y = 0$ should be rendered at the center of the screen. However, it's not as simple as the x and y coordinates, since we are rendering 3D, the z coordinate also counts. For two vertices with $v1_x = v2_x$ and $v1_y = v2_y$, the vertex with the biggest z coordinate will be **more on the center of the screen than another**, just like how we *see* in perspective. To do this, we just transform it again:

$$finalPosition = projectionMatrix * modelViewPosition$$

Here is a way to visualize it. We have blue objects in the scene, and the red shape represents the frustum of the camera - *the part of the scene the camera can actually see.*

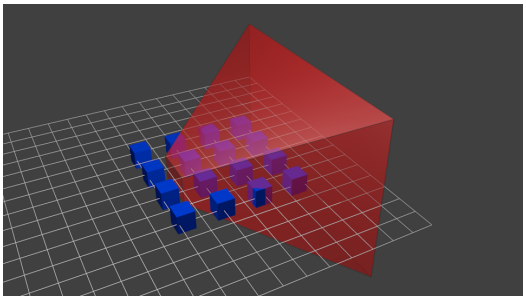


Figure 3.7:

When we multiply **everything** by the projection matrix (i.e everything is the `modelViewPosition`) it has the following effect:

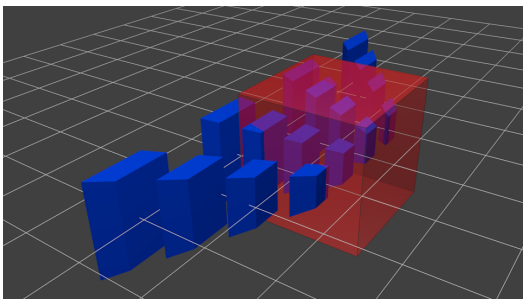


Figure 3.8:

Now the projection looks like a cube, and from 'behind' the camera frustum it looks like this:

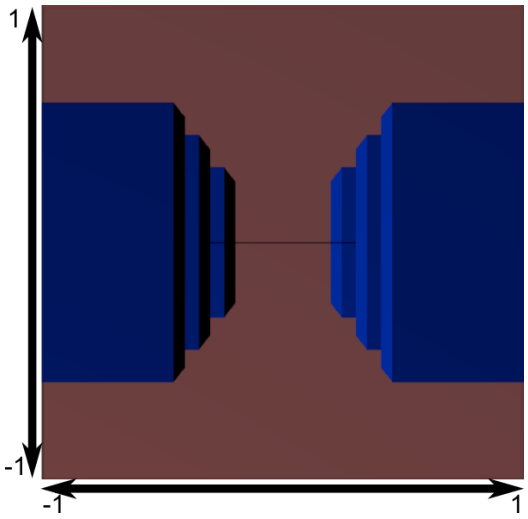


Figure 3.9:

Images taken from http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

The reason that WebGL uses matrices for translations, rotations and scaling is that matrices are a singular structure. This means that we can move around all this information in just a single structure.

3.3.3 Mapping the globe texture

Instead of using a `ShaderMaterial` for our sphere, I will switch back to a `MeshBasicMaterial`. `THREE.js` allows us to easily map a texture using a `TextureLoader`. I saved the texture in a new folder `texture`.[13]

```
const texture = new THREE.TextureLoader().load('texture/8
    k_earth_daymap.jpg')
const material = new THREE.MeshBasicMaterial({
    map: texture
})
```

Now that our material has been changed, updating our project, our globe looks like this:

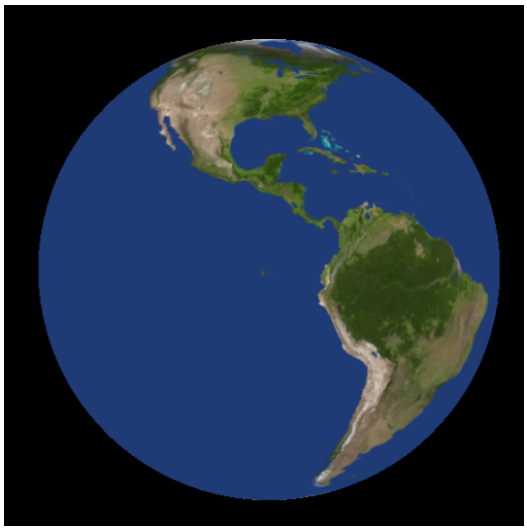


Figure 3.10:

This is quite an ugly looking planet however, and so I decided to create an atmosphere.

[13]Texture: https://www.solarsystemscope.com/textures/

3.3.4 Creating an atmosphere

An atmosphere can be created by adding another, slightly larger, sphere to the scene. Since our planet has a radius of 5, I chose to let the atmosphere's radius to be 5.02. The atmosphere will be a good example of usage of `GLSL`. Firstly, we need to create the atmosphere sphere, this is exactly the same as the globe, however, instead of the `MeshBasicMaterial`, we will use a `ShaderMaterial`:

```
import * as sphereShader from "./shader/sphereShader.js";

const atmosphere = new THREE.Mesh(
    new THREE.SphereGeometry(5.02, 64, 64),
    new THREE.ShaderMaterial({
        transparent: true,
        vertexShader: sphereShader.vertex,
        fragmentShader: sphereShader.fragment,
    })
);
```

Now, we need to define our atmosphere shader `sphereShader`. This is defined in a different file located in "shader/sphereShader.js". Looking back at our atmosphere algorithm:

```
Intensity = DotProduct(Normal, Vector(0, 0, 1))
Atmosphere = Vector(0, 1.2, 2.0) * Intensity
Pixel Colour = Vector(Atmosphere.XYZ, 0.5)
```

We need to translate this pseudocode into actual GLSL code:

Vertex Shader

```
varying vec3 vUv;
varying vec3 vNormal;
void main(void) {
  vUv = position;

  vec4 modelViewPosition = modelViewMatrix * vec4(position,
      1.0);
  gl_Position = projectionMatrix * modelViewPosition;
  vNormal = normalize(normalMatrix * normal);
}
```

Fragment Shader

```
varying vec3 vUv;
varying vec3 vNormal;

void main(void) {
  float intensity = 1.25 - dot(vNormal, vec3(0.0, 0.0, 1.0)
     );
  vec3 atmosphere = vec3(1.0, 1.0, 1.0) * pow(intensity *
     1.5, 3.0);
  gl_FragColor = vec4(vec3(0.0, 0.7, 2.0) * atmosphere,
     0.5);
}
```

After tweaking some parameters, our `GLSL` code is not quite the same as our pseudocode. Firstly, I clamped the `Intensity` to be at **least** 1.25, otherwise our atmosphere will be too thin. After that, I modified the `atmosphere` vector to follow this formula:

$$\text{Atmosphere} = \begin{bmatrix}1\\1\\1\end{bmatrix} \cdot (1.5\text{Intensity})^2$$

The singleton vector of 1 essentially turns our intensity variable into a `vec3`, and I further tweaked the intensity so that the further the pixel is from the center, the less opaque it is. Finally, I multiplied the finalized atmosphere vector with a cyan-ish colour, and toned down the final opacity of the globe to 0.5. After adding the `atmosphere` mesh to the scene (`scene.add(atmosphere)`) our globe looks like this:

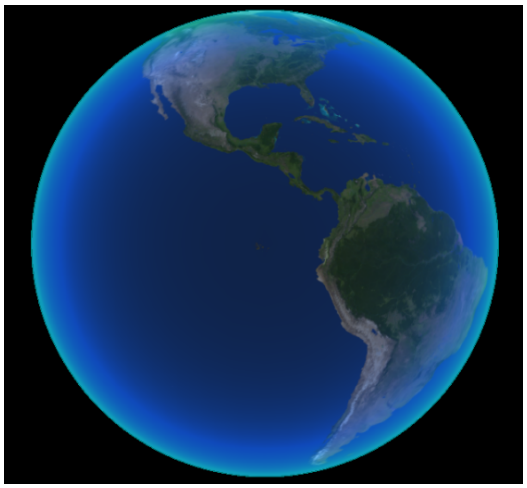


Figure 3.11:

I think it still looks a bit lacklustre, and so I've decided to add some stars to the background.

3.3.5 Stars

I want a lot of stars in the background, however, I think that adding each star as a `SphereGeometry` would slow down the project a lot. I want thousands of stars, and I don't think each star needs to be a sphere, instead we can dumb down each star to represent a single pixel. In `THREE.js` we have access to a data structure called a `BufferGeometry`. A `BufferGeometry` stores a constant amount of 'vertices', very efficiently. Since I only need stars to be pixels, I can just represent them as points, thus `BufferGeometry` is a perfect fit for this, as it only stores coordinates (vertices).

Firstly, lets create an array containing a large amount of x,y,z coordinates.

```
const star_vertices = [];
for (let i = 0; i < 10000; i++) {
    const x = THREE.MathUtils.randFloatSpread(2000);
    const y = THREE.MathUtils.randFloatSpread(2000);
    const z = THREE.MathUtils.randFloatSpread(2000);

    star_vertices.push(x,y,z);
}
```

Here I create 10000 random x,y,z coordinates with each coordinate spread in a range of 0 to 2000 using the `randFloatSpread` function.

Now, we convert this array into a `BufferGeometry` as such:

```
const star_geometry = new THREE.BufferGeometry();
star_geometry.setAttribute('position', new THREE.
    Float32BufferAttribute(star_vertices, 3));
```

Let's convert this into a mesh, using a `ShaderMaterial` as the material, and `Points` as the mesh. The `Points` mesh just holds vertices, just like the `BufferGeometry`.

```
import * as starShader from "./shader/starShader.js";

const star_material = new THREE.ShaderMaterial({
    uniforms: star_uniforms,
    vertexShader: starShader.vertex,
    fragmentShader: starShader.fragment,
});
const stars = new THREE.Points(star_geometry, star_material
    );
```

Let's also create a `GLSL` shader pipeline for the stars `starShader`, we will leave the vertex shader as normal without any vertex manipulation, since we will only manipulate the fragment shader.

Fragment Shader

```
varying vec3 vUv;
varying vec3 vNormal;

uniform float delta;

void main(void) {
    vec3 vN = normalize(vUv);
    vec3 blink = vec3(1.0, 1.0, 1.0) * sin(delta * 0.05 +
       vUv.x);
    gl_FragColor = vec4(blink, 1.0);
}
```

The fragment shader simply adds a flicker to each star, over time (`delta`). Firstly, I normalized the vertex position (`vUv`) to reduce the large position range into a decimal number from 0 to 1. Secondly, I added a flicker to the stars (`blink`) by multiplying the color white `vec3(1.0, 1.0, 1.0)` with a slowly changing sine wave `sin(delta * 0.05 + vUv.x)`.

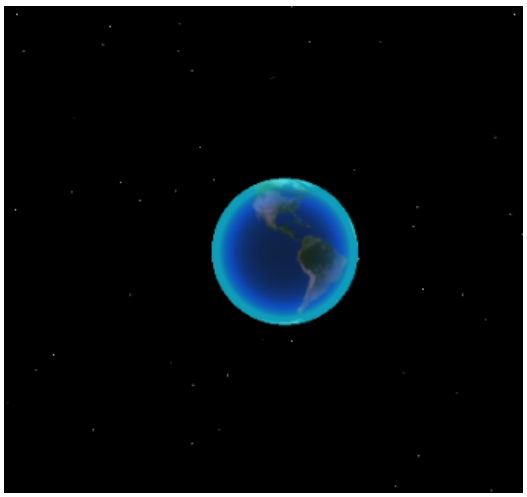


Figure 3.12:

3.3.6 Overview

No changes made to `index.html`.

Changes made to `main.js`:

```
import * as sphereShader from "./shader/sphereShader.js";
import * as starShader from "./shader/starShader.js";

// Create globe object
const geometry = new THREE.SphereGeometry(5, 64, 64);
const texture = new THREE.TextureLoader().load('texture/8
    k_earth_daymap.jpg');

const material = new THREE.MeshBasicMaterial({
  map: texture
});
const sphere = new THREE.Mesh(geometry, material);
scene.add(sphere);

// Create atmosphere object
const atmosphere = new THREE.Mesh(
  new THREE.SphereGeometry(5.02, 64, 64),
  new THREE.ShaderMaterial({
    transparent: true,
    vertexShader: sphereShader.vertex,
    fragmentShader: sphereShader.fragment,
  })
);

scene.add(atmosphere);

// Create stars
const star_vertices = [];
for (let i = 0; i < 10000; i++) {
    const x = THREE.MathUtils.randFloatSpread(2000);
    const y = THREE.MathUtils.randFloatSpread(2000);
    const z = THREE.MathUtils.randFloatSpread(2000);

    star_vertices.push(x,y,z);
}

const star_geometry = new THREE.BufferGeometry();
star_geometry.setAttribute('position', new THREE.
```

```
    Float32BufferAttribute(star_vertices, 3));

// The delta uniform is modified every animation tick
// We can use the delta uniform in the star shader to cause
    blinking of the stars
let star_uniforms = {
    delta: { value: 1.0 },
};

const star_material = new THREE.ShaderMaterial({
    uniforms: star_uniforms,
    vertexShader: starShader.vertex,
    fragmentShader: starShader.fragment,
});
const stars = new THREE.Points(star_geometry, star_material
    );
scene.add(stars);

function animate() {
  // Here I am updating the delta uniform so it increases
     by 0.5 every animation tick
  star_uniforms.delta.value += 0.5;

  requestAnimationFrame(animate);
  renderer.render(scene, camera);
}

animate();
```

shader/sphereShader.js This is the shader code for the globe's atmosphere.

```
export const vertex = `
varying vec3 vUv;
varying vec3 vNormal;
void main(void) {
  vUv = position;

  vec4 modelViewPosition = modelViewMatrix * vec4(position,
      1.0);
  gl_Position = projectionMatrix * modelViewPosition;
  vNormal = normalize(normalMatrix * normal);
}
`;
```

```
export const fragment = `
varying vec3 vUv;
varying vec3 vNormal;

void main(void) {
  float intensity = 1.25 - dot(vNormal, vec3(0.0, 0.0, 1.0)
     );
  vec3 atmosphere = vec3(1.0, 1.0, 1.0) * pow(intensity *
     1.5, 3.0);
  gl_FragColor = vec4(vec3(0.0, 0.7, 2.0) * atmosphere,
     0.5);
}
`;
```

shader/starShader.js This is the shader code for the stars.

```
export const vertex = `
varying vec3 vUv;
varying vec3 vNormal;
void main(void) {
  vUv = position;

  vec4 modelViewPosition = modelViewMatrix * vec4(position,
      1.0);
  gl_Position = projectionMatrix * modelViewPosition;
  vNormal = normalize(normalMatrix * normal);
}
`;

export const fragment = `
varying vec3 vUv;
varying vec3 vNormal;

// Import delta uniform from star_uniforms
uniform float delta;

void main(void) {
    vec3 vN = normalize(vUv);
    vec3 blink = vec3(1.0, 1.0, 1.0) * sin(delta * 0.05 +
       vUv.x);
    gl_FragColor = vec4(blink, 1.0);
}
`;
```

3.4 User Input

3.4.1 Orbital Controls

Through my analysis, I have decided to control and rotate the globe using an orbital control system. `THREE.js` supplies its own orbital control system. Since orbital control systems are quite complex, I have decided to go with `THREE.js`'s version. My reasoning for this is that `THREE.js`'s version is fast, small, and allows for a lot of functionality.
Firstly, I need to import the orbital control system, and attach it to our web app:

```
import { OrbitControls } from './OrbitControls.js';
// Attach the controls to our camera and allow it to have
    access to the domElement for events
const controls = new OrbitControls(camera, renderer.
    domElement);
```

I can finally see the back of the globe. And even zoom out.

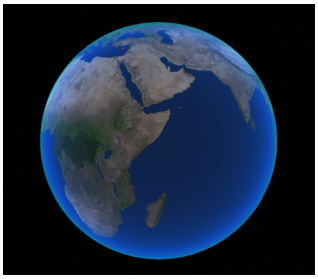


Figure 3.13:

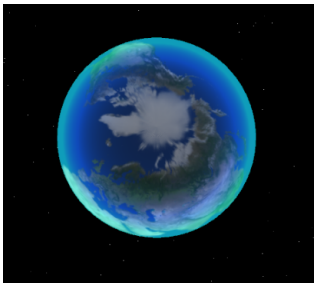


Figure 3.14:

3.4.2 Country Selection

Now I have to handle the mouse click, and correctly select a country. I am going to go with the nearest neighbour algorithm for selecting a country.
I need to add a listener for the 'mousedown' event, to execute an event on click. I have decided to structure the mouse as a 2D `THREE.js` vector.

```
document.addEventListener('mousedown', onDocumentMouseDown,
    false);
let mouse = new THREE.Vector2();

function onDocumentMouseDown(event) {
    console.log("Clicked!");
}
```

Now, whenever I click the webpage, it responds with "Clicked!"

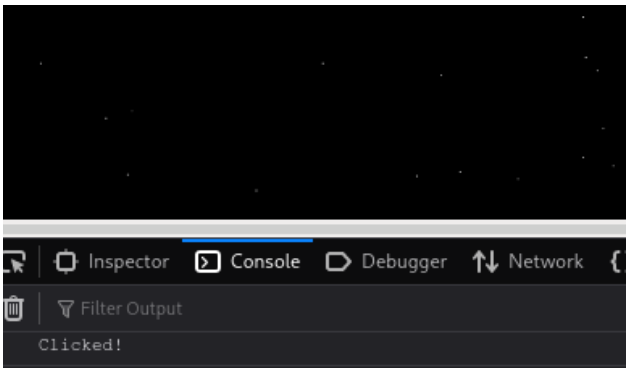


Figure 3.15:

I can also read the mouse coordinates:

```
document.addEventListener('mousedown', onDocumentMouseDown,
    false);
let mouse = new THREE.Vector2();

function onDocumentMouseDown(event) {
    mouse.x = (event.clientX / window.innerWidth) * 2 - 1;
    mouse.y = -(event.clientY / window.innerHeight) * 2 +
        1;
    console.log({x: mouse.x, y: mouse.y});
}
```

Now, whenever the user clicks on the web page, it responds with the **normalized** mouse coordinates on the screen. These coordinates range from -1 to 1, and are easier to work with than window size based coordinates.

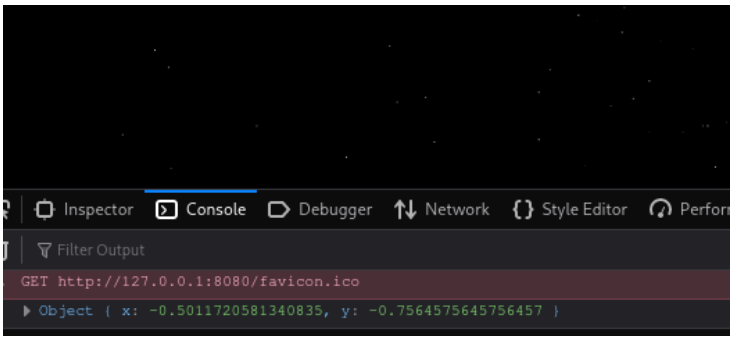


Figure 3.16:

Now that we have access to the normalized mouse coordinates, we need somehow detect that the user has clicked on the sphere, and where on the sphere the user clicked. This involves usage of a `THREE.js` class called the `Raycaster`. The raycaster sends a 'ray' in which whatever object it intersects, it records its intersection position.

```
const raycaster = new THREE.Raycaster();

function onDocumentMouseDown(event) {
    mouse.x = (event.clientX / window.innerWidth) * 2 - 1;
    mouse.y = -(event.clientY / window.innerHeight) * 2 +
        1;

    // Position the raycaster so that it is in the same
       position as the camera
    raycaster.setFromCamera(mouse, camera);

    // Send a ray from the raycaster and check if it
       intersects with any 'sphere's, in this case, it
       should only intersect the globe.
    let globe = raycaster.intersectObject(sphere);

    // Iterate through all the intersections, there should
       only be one intersection anyways, but just in case
       there were no intersections, the for loop is skipped
       .
    for (let i = 0; i < globe.length; i++) {
        // 'point' represents the intersection coordinates
        let point = globe[i].point;

        // 'norm' represents the normalized intersection
```

```
            coordinates, to be converted into latitude and
               longitude, since its algorithm requires it.
            let norm = new THREE.Vector3(point.x, point.y,
               point.z);

            // normalize 'norm'
            norm.normalize();
        }
}
```

After this change, when the user clicks, the correct, normalized coordinates of the exact position of the click in relation to the sphere is recorded. I can finally convert this into latitude and longitude, and begin to select a country:

```
let latitude = Math.asin(norm.y) * (180.0 / Math.PI);
let longitude = Math.atan2(norm.z, norm.x) * (-180.0 / Math
    .PI);
console.log({latitude, longitude});
```

When the user clicks on the globe now, the console responds:

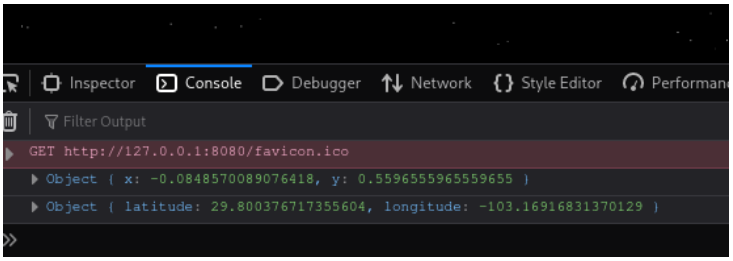


Figure 3.17:

After these changes, we can begin to detect and select countries. Firstly, we need to retrieve some sort of file which contains a list of countries, with their respective name and latitude and longitude coordinates.
I got my file from this MIT Licensed github repository: https://github.com/eesur/country-codes-lat-long
Each data entry is formatted as such:

```
{
  "country" : "Albania",
  "alpha2" : "AL",
  "alpha3" : "ALB",
  "numeric" : 8,
  "latitude" : 41,
```

```
  "longitude" : 20
},
```

Since we now have each piece of the puzzle, we can begin connecting each piece. Firstly, we need to apply our nearest neighbour algorithm to select the country which is nearest to the latitude and longitude location. I turned this json file into a js file since json is embeddable in js, and imported it into my main.js file:

```
import { countries } from './countries.js';
```

Now I can apply the nearest neighbour algorithm:

```
// Calculate the distance between two objects { longitude:
    ..., latitude: ... }
function distance(a, b) {
  // Pythagorean rule
  let dx = a.longitude - b.longitude;
  let dy = a.latitude  - b.latitude;
  return Math.sqrt(dx*dx + dy*dy)
}

let lng_lat = { longitude, latitude }

let closest =
    countries["ref_country_codes"] // our json file
       contains this as its initial key
    .reduce((a, b) => distance(a, lng_lat) < distance(b,
       lng_lat) ? a : b)
```

I have condensed the algorithm into an argument to the javascript `reduce` function. I have some experience with javascript, and I think the `reduce` function can save a lot of space, and still make the code readable.
The `reduce` function simply compares the distance between the last, closest country `b` with `a`. Whichever is closest to the clicked position, is the next closest country. It is a functional pattern. I used the ternary operator `?` as a more compact `if` statement.

```
console.log(closest)
```

Now outputs: Most importantly:

Figure 3.18:

```
Object { country: "United States", alpha2: "US", alpha3: "
   USA", numeric: 840, latitude: 38, longitude: -97 }
alpha2: "US"
alpha3: "USA"
country: "United States"
latitude: 38
longitude: -97
numeric: 840
```

Comparing the values with our clicked value:

```
Clicked: { latitude: 30.129368529775785, longitude:
   -93.34851220545634 }
```

They are quite close, which dictates that the algorithm does in fact work.

Overview - main.js

```
import { countries } from './countries.js';

const raycaster = new THREE.Raycaster();

document.addEventListener('mousedown', onDocumentMouseDown,
    false);
let mouse = new THREE.Vector2();

// Calculate the distance between two objects { longitude:
   ..., latitude: ... }
function distance(a, b) {
  // Pythagorean rule
  let dx = a.longitude - b.longitude;
  let dy = a.latitude  - b.latitude;
  return Math.sqrt(dx*dx + dy*dy)
}


function onDocumentMouseDown(event) {
    mouse.x = (event.clientX / window.innerWidth) * 2 - 1;
    mouse.y = -(event.clientY / window.innerHeight) * 2 +
       1;

    // Position the raycaster so that it is in the same
       position as the camera
    raycaster.setFromCamera(mouse, camera);

    // Send a ray from the raycaster and check if it
       intersects with any 'sphere's, in this case, it
       should only intersect the globe.
    let globe = raycaster.intersectObject(sphere);

    // Iterate through all the intersections, there should
       only be one intersection anyways, but just in case
       there were no intersections, the for loop is skipped
       .
    for (let i = 0; i < globe.length; i++) {
        // 'point' represents the intersection coordinates
        let point = globe[i].point;

        // 'norm' represents the normalized intersection
           coordinates, to be converted into latitude and
```

```
            longitude, since its algorithm requires it.
        let norm = new THREE.Vector3(point.x, point.y,
            point.z);

        // normalize 'norm'
        norm.normalize();

        let latitude = Math.asin(norm.y) * (180.0 / Math.PI
            );
        let longitude = Math.atan2(norm.z, norm.x) *
            (-180.0 / Math.PI);
        console.log({latitude, longitude});
        let lng_lat = { longitude, latitude }

        let closest =
            countries["ref_country_codes"] // our json file
                contains this as its initial key
            .reduce((a, b) => distance(a, lng_lat) <
                distance(b, lng_lat) ? a : b)
    }
}
```

3.5 API Requests & User Interface

I now have the alpha-2 and alpha-3 ISO codes for the closest country selected through the mouse click, and the country name. I can start developing the user interface, so I need to switch back to HTML & CSS for now.

index.html

```
<html>
  <head>
    <meta charset="utf-8">
    <title>Globe</title>
    <style>
      body { margin: 0; }

            .data-container {
                position: absolute;
                display: grid;
                height: 100%;
                width: 20vw;
                right: 0;
```

```
                background: #000000;
                border-color: #FFFFFF;
                border-width: 2px;
                border-left-style: solid;
            }

            .data-title {
                font-size: 3vh;
            }

            .data-header {
                font-size: 2.5vh;
            }

            .data-text {
                font-family: Arial, Helvetica, sans-serif;
                color: white;
                text-align: center;
                margin-top: 1vh;
                margin-bottom: 1vh;
            }

            .data-flag {
                display: block;
                margin-left: auto;
                margin-right: auto;
                height: 10vh;
                width:  10vw;
                object-fit: contain;
            }
    </style>
  </head>
  <body>
        <div class="data-container" id="container">

        </div>
        <script type="module" src="js/main.js"></script>
  </body>
</html>
```

I have modified the HTML & CSS code to create a black `div` that sticks to the right of the screen, with a white left border to indicate the separation. Inside the `div` classed `data-container`, I can start to add the data points,

and they will be layed out vertically on top of eachother due to the `grid` layout used.

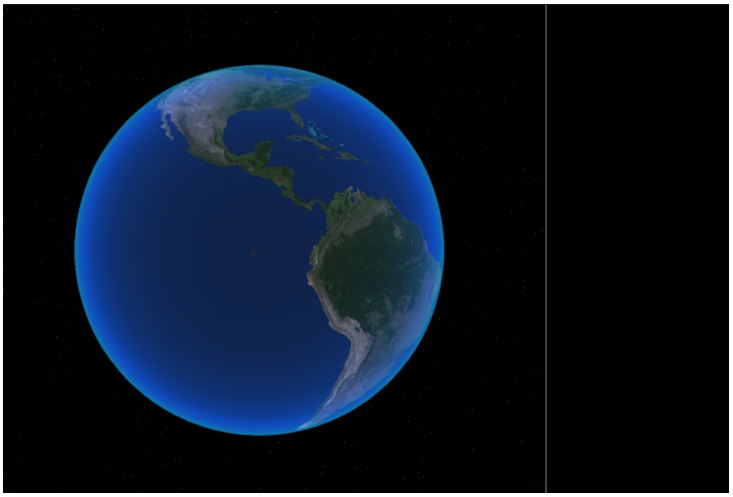


Figure 3.19: Visible white separator between space and user interface

After creating a simple user interface, I will create a basic POST query to 'countriesnow.space' to retrieve the country flag, and display the country name once clicked on the globe and applying the nearest neighbour algorithm.

index.html

```
...
<div class="data-container" id="container">
    <div>
        <p class="data-title data-text" id="title"></p>
        <img class="data-flag" style="visibility: hidden"
            id="flag"></img>
    </div>
</div>
...
```

main.js

```
function postJSON(url, data, f) {
    // POST Request a url, with the data (body) of the
        request, and a final function when the request
        responds with the json data.
    return fetch(url, {
        method: "POST",
        // The response must be in json
        headers: { "Content-Type": "application/json" },
```

```
        body: JSON.stringify(data),
    }).then(jdata => jdata.json().then(f));
}

// Set the text content of the HTML Element with id `id`.
function setContent(id, content) {
    document.getElementById(id).textContent = content;
}

function queryCountry(country) {
    setContent("title", country.country);

    setContent("population-header", "Population");
    postJSON("https://countriesnow.space/api/v0.1/countries
        /population", {
         "iso3": country.alpha3
    }, data => {
         // Find the most recent population count
         let latest = data.data.populationCounts
                                .reduce((a, b) => (a.year < b
                                    .year) ? b : a, { year: 0
                                    })
         setContent("population", latest.value);
    })
}
```

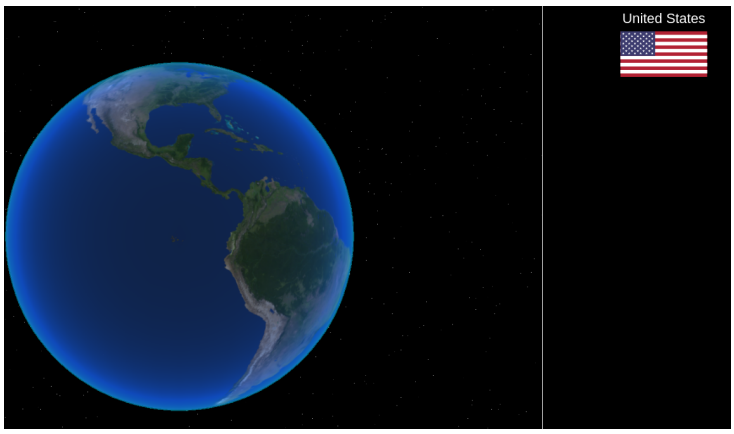


Figure 3.20:

After these changes, all I have to do, is broaden the API requests to different kinds of data.

3.5.1 Overview

main.js

```
import * as sphereShader from "./shader/sphereShader.js";
import * as starShader from "./shader/starShader.js";
import * as THREE from './three.js';
import { OrbitControls } from './OrbitControls.js';
import { countries } from './countries.js';

const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera(75, window.
    innerWidth / window.innerHeight, 0.1, 1000);

const renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

const controls = new OrbitControls(camera, renderer.
    domElement);

const raycaster = new THREE.Raycaster();

const clock = new THREE.Clock();

const geometry = new THREE.SphereGeometry(5, 64, 64);
const texture = new THREE.TextureLoader().load('texture/8
    k_earth_daymap.jpg');

const material = new THREE.MeshBasicMaterial({
  map: texture
});
const sphere = new THREE.Mesh(geometry, material);
scene.add(sphere);

const atmosphere = new THREE.Mesh(
  new THREE.SphereGeometry(5.02, 64, 64),
  new THREE.ShaderMaterial({
    transparent: true,
    vertexShader: sphereShader.vertex,
    fragmentShader: sphereShader.fragment,
  })
);
```

```
scene.add(atmosphere);

const star_vertices = [];
for (let i = 0; i < 10000; i++) {
    const x = THREE.MathUtils.randFloatSpread(2000);
    const y = THREE.MathUtils.randFloatSpread(2000);
    const z = THREE.MathUtils.randFloatSpread(2000);

    star_vertices.push(x,y,z);
}

const star_geometry = new THREE.BufferGeometry();
star_geometry.setAttribute('position', new THREE.
   Float32BufferAttribute(star_vertices, 3));

let star_uniforms = {
    delta: { value: 1.0 },
};

const star_material = new THREE.ShaderMaterial({
    uniforms: star_uniforms,
    vertexShader: starShader.vertex,
    fragmentShader: starShader.fragment,
});
const stars = new THREE.Points(star_geometry, star_material
   );
scene.add(stars);

camera.position.z = 10;

document.addEventListener('mousedown', onDocumentMouseDown,
    false);
let mouse = new THREE.Vector2();

function distance(a, b) {
  let dx = a.longitude - b.longitude;
  let dy = a.latitude  - b.latitude;
  return Math.sqrt(dx*dx + dy*dy)
}

function ltoxyz(r, lng, lat) {
  return {
    x: r * Math.cos(lat) * Math.cos(lng),
```

```
    y: r * Math.cos(lat) * Math.sin(lng),
    z: r * Math.sin(lat)
  };
}

function postJSON(url, data, f) {
    return fetch(url, {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(data),
    }).then(jdata => jdata.json().then(f));
}

function getJSON(url, f) {
    return fetch(url).then(jdata => jdata.json().then(f));
}

function setContent(id, content) {
    document.getElementById(id).textContent = content;
}

function queryCountry(country) {
    setContent("title", country.country);
    setContent("lat-long-position",
            "~"
            + Math.abs(country.latitude) + (country.
               latitude > 0 ? "N" : "S")
            + " "
            + Math.abs(country.longitude) + (country.
               longitude > 0 ? "E" : "W"))

    setContent("population-header", "Population");
    postJSON("https://countriesnow.space/api/v0.1/countries
       /population", {
        "iso3": country.alpha3
    }, data => {
        let latest = data.data.populationCounts
                              .reduce((a, b) => (a.year < b
                                 .year) ? b : a, { year: 0
                                 })
        setContent("population", latest.value);
    })
```

```
postJSON("https://countriesnow.space/api/v0.1/countries
    /flag/images", {
     "iso2": country.alpha2
}, data => {
     let flag = document.getElementById("flag");
     flag.style.visibility = "visible";
     flag.src = data.data.flag;
})

setContent("cities-header", "Cities");
postJSON("https://countriesnow.space/api/v0.1/countries
    /cities", {
     "iso2": country.alpha2
}, data => {
     setContent("cities-count", data.data.length + "
         cities");
})

postJSON("https://countriesnow.space/api/v0.1/countries
    /capital", {
     "iso2": country.alpha2
}, data => {
     setContent("cities-capital", "Capital: " + data.
         data.capital);
})

postJSON("https://countriesnow.space/api/v0.1/countries
    /currency", {
     "iso2": country.alpha2
}, data => {
     setContent("currency", data.data.currency + "$")
})

// WorldBank WDI Data

// Get GDP with most recent value (~2020) of country
setContent("wdi-header", "World Development Indicators
    ");
getJSON(`http://api.worldbank.org/v2/country/${country.
    alpha2}/indicator/NY.GDP.PCAP.PP.CD?mrv=1&format=
    json`, data => {
     if (data.length <= 1) {
          setContent("wdi-gdp", "N/A");
```

```
            return;
        }
        setContent("wdi-gdp", "GDP: " + data[1][0].value.
            toFixed(2) + "$");
    })

    getJSON(`http://api.worldbank.org/v2/country/${country.
        alpha2}/indicator/SL.UEM.TOTL.ZS?mrv=1&format=json`,
        data => {
        if (data.length <= 1) {
            setContent("wdi-unemployment", "N/A");
            return;
        }
        setContent("wdi-unemployment", "Unemployment: " +
            data[1][0].value.toFixed(2) + "%");
    })

    // Population growth (annual %)
    getJSON(`http://api.worldbank.org/v2/country/${country.
        alpha2}/indicator/SP.POP.GROW?mrv=1&format=json`,
        data => {
        if (data.length <= 1) {
            setContent("wdi-popgrow", "N/A");
            return;
        }
        setContent("wdi-popgrow", "Population Growth: " +
            data[1][0].value.toFixed(2) + "% p.a");
    })

    // Life expectancy in years
    getJSON(`http://api.worldbank.org/v2/country/${country.
        alpha2}/indicator/SP.DYN.LE00.IN?mrv=1&format=json`,
        data => {
        if (data.length <= 1) {
            setContent("wdi-lifeexpectancy", "N/A");
            return;
        }
        setContent("wdi-lifeexpectancy", "Life Expectancy:
            " + data[1][0].value.toFixed(2) + " years");
    })

    // Environmental data
    setContent("environment-header", "Environment");
```

```
    // CO2 Emissions
    getJSON(`http://api.worldbank.org/v2/country/${country.
        alpha2}/indicator/EN.ATM.CO2E.PC?mrv=1&format=json`,
        data => {
        if (data.length <= 1) {
            setContent("env-co2", "N/A");
            return;
        }
        setContent("env-co2", "CO2 Emissions: " + data
            [1][0].value.toFixed(2) + "mt/capita");
    })

    // Forest area
    getJSON(`http://api.worldbank.org/v2/country/${country.
        alpha2}/indicator/AG.LND.FRST.ZS?mrv=1&format=json`,
        data => {
        if (data.length <= 1) {
            setContent("env-forestarea", "N/A");
            return;
        }
        setContent("env-forestarea", "Forest Area: " + data
            [1][0].value.toFixed(2) + "%");
    })

    // Access to electricity (% of population)
    getJSON(`http://api.worldbank.org/v2/country/${country.
        alpha2}/indicator/EG.ELC.ACCS.ZS?mrv=1&format=json`,
        data => {
        if (data.length <= 1) {
            setContent("env-electricity", "N/A");
            return;
        }
        setContent("env-electricity", "Access to
            electricity: " + data[1][0].value.toFixed(2) +
            "% of pop.");
    })
}

function onDocumentMouseDown(event) {
  mouse.x = (event.clientX / window.innerWidth) * 2 - 1;
  mouse.y = -(event.clientY / window.innerHeight) * 2 + 1;
```

```
  raycaster.setFromCamera(mouse, camera);
  let globe = raycaster.intersectObject(sphere);

  for (let i = 0; i < globe.length; i++) {
    let point = globe[i].point;
    let norm  = new THREE.Vector3(point.x, point.y, point.z
       );
    norm.normalize()

    // let p = new THREE.Mesh(
    //   new THREE.SphereGeometry(0.1, 64, 64),
    //   new THREE.MeshBasicMaterial({ color: 0xFF0000 })
    // );

    let latitude  = Math.asin(norm.y) * (180.0 / Math.PI);
    let longitude = Math.atan2(norm.z, norm.x) * (-180.0 /
       Math.PI);
    console.log({latitude, longitude});


    let lng_lat = { longitude, latitude }
    let closest =
        countries["ref_country_codes"]
        .reduce((a, b) => distance(a, lng_lat) < distance(b
           , lng_lat) ? a : b)

    let closest_xyz = ltoxyz(5.0, closest.longitude,
       closest.latitude);

    queryCountry(closest);
    // console.log(closest_xyz);
    // console.log(point);

    // p.translateX(closest_xyz.x);
    // p.translateY(closest_xyz.y);
    // p.translateZ(closest_xyz.z);

    // scene.add(p);
  }
}

function animate() {
  star_uniforms.delta.value += 0.5;
```

```
  // sphere.rotateY(0.005);

  requestAnimationFrame(animate);
  renderer.render(scene, camera);
}

animate();
```

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Globe</title>
    <style>
      body { margin: 0; }

            .data-container {
                position: absolute;
                display: grid;
                height: 100%;
                width: 20vw;
                right: 0;

                background: #000000;
                border-color: #FFFFFF;
                border-width: 2px;
                border-left-style: solid;
            }

            .data-title {
                font-size: 3vh;
            }

            .data-header {
                font-size: 2.5vh;
            }

            .data-text {
                font-family: Arial, Helvetica, sans-serif;
                color: white;
                text-align: center;
```

```
                margin-top: 1vh;
                margin-bottom: 1vh;
            }

            .data-flag {
                display: block;
                margin-left: auto;
                margin-right: auto;
                height: 10vh;
                width:  10vw;
                object-fit: contain;
            }
    </style>
</head>
<body>
        <div class="data-container" id="container">
            <div>
                <p class="data-title data-text" id="title
                    "></p>
                <p class="data-text" id="lat-long-position
                    "></p>
                <p class="data-text" id="currency"></p>
                <img class="data-flag" style="visibility:
                    hidden" id="flag"></img>
            </div>

            <div>
                <p class="data-header data-text" id="
                    population-header"></p>
                <p class="data-text" id="population"></p>
            </div>

            <div>
                <p class="data-header data-text" id="cities
                    -header"></p>
                <p class="data-text" id="cities-capital"></
                    p>
                <p class="data-text" id="cities-count"></p>
            </div>

            <div>
                <p class="data-header data-text" id="wdi-
                    header"></p>
```

```
                <p class="data-text" id="wdi-gdp"></p>
                <p class="data-text" id="wdi-unemployment
                    "></p>
                <p class="data-text" id="wdi-popgrow"></p>
                <p class="data-text" id="wdi-lifeexpectancy
                    "></p>
            </div>

            <div>
                <p class="data-header data-text" id="
                    environment-header"></p>
                <p class="data-text" id="env-co2"></p>
                <p class="data-text" id="env-forestarea"></
                    p>
                <p class="data-text" id="env-electricity
                    "></p>
            </div>
        </div>
        <script type="module" src="js/main.js"></script>
    </body>
</html>
```

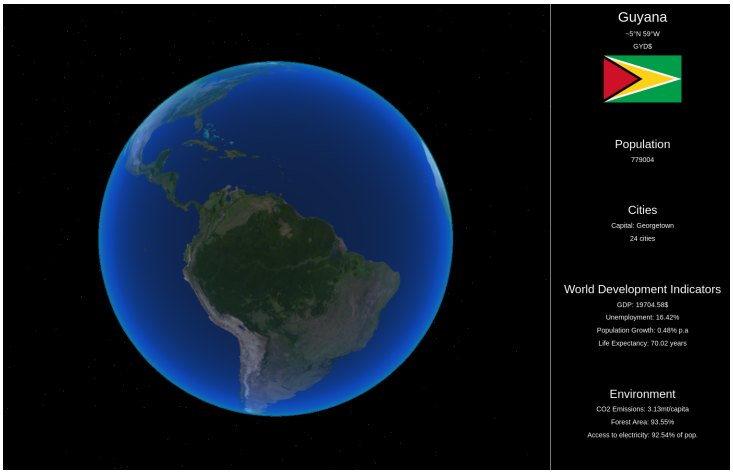


Figure 3.21: Final user interface

Chapter 4

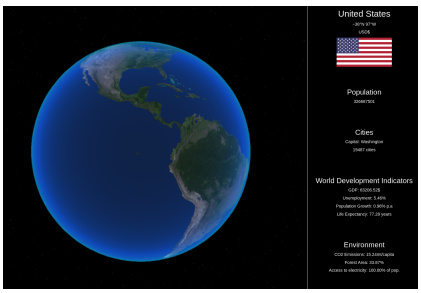
Testing

4.1 Test Plan

To test software, a plan must be set out to efficiently capture all bugs and to test the accuracy and precision of the said software. The goal of this section is to devise an efficient testing plan.
My project is not very edge-case intensive, each component in my project works with one another and if one breaks, then the whole project stops making sense. Since this is the case, catching a bug is critical, however, bugs are rare since they are so easy to catch. My plan is to pick different tests, and to apply those tests in different locations of the globe, multiple times. That way, I can efficiently find any bugs, glitches or unexpected behaviour. I will also be testing the *accuracy* of my project, by comparing values with real, trusted data banks.

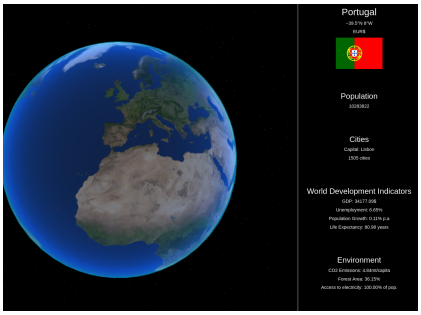
4.2 Accuracy Testing

1. Objective: Select 'United States'
   Test Result: **Passed**

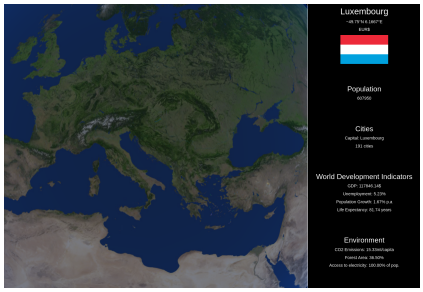


2. Objective: Select 'Spain'
   Test Result: **Passed**

3. Objective: Select 'Portugal'
   Test Result: **Passed**

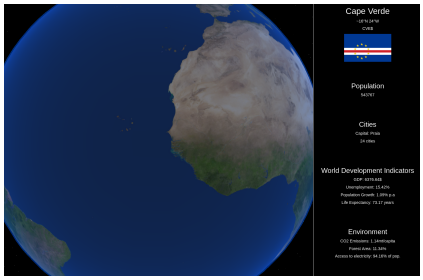


4. Objective: Select 'Luxembourg'
   Test Result: **Passed**
   Note: Needed to zoom in to accurately select it.

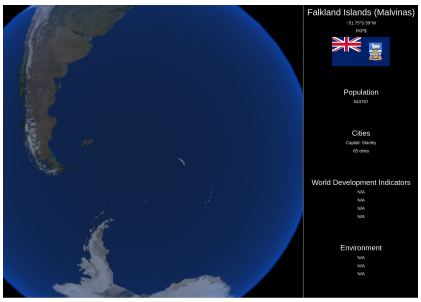


5. Objective: Select 'Italy'
   Test Result: **Failed**
   Note: Kept selecting San Marino & Vatican City due to their extreme proximity to italy.

6. Objective: Select 'Cape Verde'
   Test Result: **Passed**



7. Objective: Select 'Falkland Islands'
   Test Result: **Passed**
   Note: Inconclusive world bank indicator API results.



4.2.1 Test Conclusion

In conclusion, I think that the tests were quite successful. It can select even small islands, and most of europe passes the test. However, countries like Italy failed due to it's enclaves (Vatican City, San Marino microstate). This could be solved by using a different model for country selection such as the complex country border mapping model, which involves mapping borders onto the globe, and calculating selection that way. I'd say that my model is actually quite accurate, and for most cases, it can get the country right the very first attempt.

4.3 API Testing

I will only be testing the population variable and the GDP variable to save time. Population & GDP is compared with CIA estimate.[1]

1. Objective: Test 'Portugal'
   Comparison:
   - **GDP per capita (PPP)**
     Result: 34,177$
     Real: 32,000$
   - **Population**
     Result: 10,283,822
     Real: 10,142,236
2. Objective: Test 'Iceland'
   Comparison:
   - **GDP per capita (PPP)**
     Result: 53,616$
     Real: 52,300$
   - **Population**
     Result: 352,721
     Real: 357,603
3. Objective: Test 'Madagascar'
   Comparison:
   - **GDP per capita (PPP)**
     Result: 1,544$
     Real: 1,500$
   - **Population**
     Result: 26,262,368
     Real: 28,172,462

[1] https://www.cia.gov/the-world-factbook/field/real-gdp-per-capita/country-comparison

4. Objective: Test 'Falkland Islands'
   Comparison:

   - **GDP per capita (PPP)**
     Result: N/A$
     Real: 70,800$
   - **Population**
     Result: 1,265,303
     Real: 3,198

4.3.1 Test Conclusion

I'd say that the majority of the tests were correct, with a small error, probably due to differences in time taken of data or estimate differences between the CIA database and the World Bank database. However, the last test, was not so successful. Firstly, the CIA database contained the GDP per capita (PPP) of the Falklands Islands, unlike the World Bank's database. Secondly, the population was much, much higher than expected. I think that this is not the API's fault, and that it is the fact that there was actually no value for the population at all, and that the last population value was not updated in the user interface.

Chapter 5

Evaluation

5.1 Objective Completeness and Improvements

1. **Rendering the globe**

   - **Completeness** In the final program, the user is able to see a globe on the screen, with a well rendered atmosphere covering the globe. The globe is wrapped with a texture of the earth.
   - **Improvements** In the final program, I could add clouds, to add some more realism.

2. **User Input**
   The user must be able to rotate the globe.

   - **Completeness** In the final program, the user is able to rotate the globe, and zoom in and out away or towards the globe using only the mouse.
   - **Improvements** I don't think this objective requires any extra improvements.

   The user must be able to click and select a country.

   - **Completeness** In the final program, the user is able to click and select a country, provided that they roughly click inside the countries borders.
   - **Improvements** I think that, the model which I used (closest neighbour) could be improved upon by switching to a method which involves actual country borders. This would resolve any issues such as clicking the water resulting in selecting the nearest country, and would also improve the accuracy of the selection.

3. **API Requests**
   The user must be able to read the selected countries name, and see the countries flag.

   - **Completeness** This objective is complete.

   The user should be able to read the population of the country, and its native currency.

   - **Completeness** This objective is complete.
   - **Improvements** In some edge-cases, such as very small islands or barely habited countries, the population is not valid. This causes no update in the user interface (no switch from the previous result to N/A) and thus can cause confusion. This bug needs fixing.

   The user could be able to see extra data points, such as GDP, or environmental factors.

   - **Completeness** This objective is complete.
   - **Improvements** In some edge-cases, such as very small islands or barely habited countries, the World Bank database may not contain entries for these countries. A potential solution to this is to extract data from many databases, and pick the valid result.

5.2 Conclusion

In conclusion, I think that my project is quite well built. It's quite accurate, and it checks all of the main objectives which I was required to complete. Most of the features work out of the box, and there is very little user requirement or thought to use the app.

If I were to continue the development of this web application, I would probably polish the globe, and make it look really realistic, I think that would look great. I think a nice update to the appearance of the HTML & CSS user interface would be also a great addition.