Global AIS Data Visualization on a 3D Earth

by
Hongyan Yi

A MASTER’S PROJECT REPORT

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science in Computer Science

Presented May 31, 2017
Commencement June 2017
AN ABSTRACT OF THE PROJECT OF

Hongyan Yi for the degree of Master of Science in Computer Science presented on May 31, 2017.

Title: Global AIS Data Visualization on a 3D Earth

Abstract approved:

Prof. Mike Bailey

The constant increase in marine traffic requires a strategy to manage safety. The automatic identification system (AIS) was developed as a navigation safety device for ships in the 1990s. AIS is intended, primarily, to allow ships to view marine traffic in their area and to be seen by that traffic. However, the huge dimension of the AIS data to process requires the adoption of careful strategies for the data visualization. In this project, our work focused on visualizing the global AIS data on a 3D Earth. It was necessary to have a global view to see the ship distribution throughout the world, for marine research and business. Three.js was used to create and display animated 3D computer graphics for this Web Application.
Master of Science Project Report of Hongyan Yi presented on May 31, 2017

APPROVED:

[Signature]
Major Professor, representing Major Name

[Signature]
Head (Chair or Director) of the Department (School, College) of (department name)

[Signature]
Dean of the Graduate School

I understand that my project report will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my project report to any reader upon request.

[Signature]
Hongyan Yi, Author
ACKNOWLEDGEMENTS

First and foremost, I would thank my major advisor Prof. Mike Bailey for his continuous help throughout my study and research in OSU. I've taken all his graphic classes, Scientific Visualization, Parallel Programming, Intro to Computer Graphics and Computer Graphics Shaders. He is pretty patient and helpful for me both in and after class or during office hours. These make me feel less shy or upset to ask questions and get help from him. He is very knowledgeable, every time I ask questions, he could explain them in very simple way which are easier to understand. As a major advisor, he always think more for students, give useful help to students in practice. For example, give creative hints or guide for research, share effective information in research group, help students mock defend or interview, lend students valuable books. At least, every two weeks could have a one-to-one meeting, he also open to meet and help when his door is open. I appreciate to have such excellent and helpful advisor.

I would also thank Prof. Alex Groce, he is my first advisor in OSU, although he left OSU after my second year study. Thank you for choosing me as a PhD student after I submitted application in China, without your choice I could not directly come to OSU. Thank you also for giving me freedom to learn diverse coursework, which inspired my interest in Computer Graphics and drove me to switch to Prof. Bailey after you leave. Thank you for giving the opportunity of being your Graduate Teaching Assistant.

I would also thank my committee member, Prof. Bo Zhao and Prof. Raffaele de Amicis. Bo gave me lots of hints when I choosing my Master project topic. He also supplied me the raw AIS data and gave me guidance for my Master project in a GIS view. Raffaele was nice and always open to help if I needed.
Then, I truly appreciate my whole big family. Their endless supports save me lots of time and energy, so that I could do study-life balance. Their trust gives me a lot of motivation to get MS degree and find a job in US.

Finally, I want to thank friends that I've had along the way who help me to be a better person: Amin Alipour, Laxmi Ganesan, Yuanli Pei, Arpit Moses Christi and so on.

The Source code for this project is in:

https://github.com/AngieYi/Ship_Movements_On_3D_Earth

Feel free to correct my mistakes and discuss with me through email:
hongyan.angie.yi@gmail.com.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. AIS Data</td>
<td>2</td>
</tr>
<tr>
<td>2.1 AIS Raw Data</td>
<td>2</td>
</tr>
<tr>
<td>2.2 AIS Filtered Data</td>
<td>6</td>
</tr>
<tr>
<td>3. Web Development Library: Three.js</td>
<td>7</td>
</tr>
<tr>
<td>4. Earth, Clouds and Stars</td>
<td>8</td>
</tr>
<tr>
<td>4.1 Scene, Camera and Renderer</td>
<td>8</td>
</tr>
<tr>
<td>4.1.1 Scene</td>
<td>8</td>
</tr>
<tr>
<td>4.1.2 Camera</td>
<td>8</td>
</tr>
<tr>
<td>4.1.3 Renderer</td>
<td>10</td>
</tr>
<tr>
<td>4.2 Lighting</td>
<td>11</td>
</tr>
<tr>
<td>4.2.1 Ambient Light</td>
<td>11</td>
</tr>
<tr>
<td>4.2.2 Directional Light</td>
<td>12</td>
</tr>
<tr>
<td>4.3 Creating Earth</td>
<td>13</td>
</tr>
<tr>
<td>4.3.1 Creating Sphere</td>
<td>13</td>
</tr>
<tr>
<td>4.3.2 Texture Mapping</td>
<td>13</td>
</tr>
<tr>
<td>4.3.3 Clear Earth without Clouds</td>
<td>14</td>
</tr>
<tr>
<td>4.3.4 Bump Mapping</td>
<td>16</td>
</tr>
<tr>
<td>4.3.5 Reflective Ocean and Lakes</td>
<td>21</td>
</tr>
<tr>
<td>4.4 Adding Clouds</td>
<td>24</td>
</tr>
<tr>
<td>4.4.1 Separate Clouds and Earth</td>
<td>24</td>
</tr>
<tr>
<td>4.4.2 Clouds Implementation</td>
<td>26</td>
</tr>
<tr>
<td>4.5 Adding Stars</td>
<td>27</td>
</tr>
<tr>
<td>4.6 Rotate, Zoom and Translate</td>
<td>29</td>
</tr>
<tr>
<td>5. Ship Visualization</td>
<td>31</td>
</tr>
<tr>
<td>5.1 Ship Point Cloud</td>
<td>31</td>
</tr>
<tr>
<td>5.1.1 Point Cloud Geometry</td>
<td>31</td>
</tr>
<tr>
<td>5.1.2 Point Cloud Material</td>
<td>32</td>
</tr>
</tbody>
</table>
5.2 Ship Spline Curve ................................................................................................. 36
  5.2.1 Spline Curve and Control Points ................................................................. 36
  5.2.2 Spherical Linear Interpolation ...................................................................... 37
  5.2.3 Control Points Position ................................................................................ 37
  5.2.4 Central Angle in Slerp .................................................................................. 38
  5.2.5 Latitude Longitude to XYZ .......................................................................... 39
  5.2.6 Latitude and Longitude of Control Points .................................................. 41
  5.2.7 Spline Curve with Control Points ............................................................... 43
5.3 Ship Route ........................................................................................................... 44
5.4 Ship Animation .................................................................................................... 46
  5.4.1 Ship Movements ........................................................................................... 46
  5.4.2 Scene Animation .......................................................................................... 48
6. GUI Design and Implementation ........................................................................... 50
  6.1 Changing Ship Point Size ................................................................................ 50
  6.2 Changing Ship Speed ....................................................................................... 51
  6.3 Changing the Opacity of Ship Route ............................................................... 52
  6.4 About Window .................................................................................................. 53
7. Resizing Window .................................................................................................... 55
8. Stats Counter .......................................................................................................... 57
9. Loading Overlay ..................................................................................................... 58
10. Conclusions and Future Work ............................................................................ 60
References .................................................................................................................. 61
1. Introduction

The constant increase in marine traffic requires a strategy to manage safety. The adoption of e-Navigation is a possible solution for improving safety and security at sea by integrating maritime information on board and ashore [1]. The automatic identification system (AIS) was developed as a navigation safety device for ships in the 1990s. AIS is intended, primarily, to allow ships to view marine traffic in their area and to be seen by that traffic. After the terrorist attacks on Sept. 11, 2001, maritime safety authorities all over the world realized that ships could be used by terrorists either as targets or as weapon delivery systems. AIS provided a ready to implement system to track ships, as well as augment navigation safety. AIS was soon required in the United States and internationally on most merchant ships [2]. However, the huge dimension of the AIS data to process requires the adoption of careful strategies for the data visualization.

For the large amount of AIS data, it's necessary to have a global view to see the ship distribution throughout the world. In this project, my work focused on visualizing the AIS data on a 3D Earth, which was obvious for the user to see all the ships in one application. For the sake of easier user access, this project was designed to be a Web application. The remainder of this paper was organized as follows. Chapter 2 introduced AIS raw data and explained how to filter the data. Chapter 3 introduced a web data visualization library Three.js. Chapter 4 illustrated how to draw the Earth, clouds and stars. Chapter 5 was the most difficult and important part, it presented how to locate the ship, animate the ship and visualize the ship routes. Chapter 6 illustrated how to use dat.GUI to implement GUI for this specific project. Chapter 7 explained how to deal with the camera when changing the window size. Chapter 8 explained the stats counter for this project. Chapter 9 illustrated how I designed and implemented the loading overlay. Chapter 10 illustrated my conclusions and future work for this project.
2. AIS Data

I collected the raw AIS data from College of Earth, Ocean, and Atmospheric Sciences (CEOAS) of Oregon State University (OSU), since they have connection with AISHub (http://data.aishub.net/), which shares data with CEOAS.

2.1 AIS Raw Data

AIS data contains information about the vessel identity, position, speed, and course with Vessel Traffic Services (VTS) stations as well as with other ships. In particular, depending on a vessel’s speed, AIS transmitters send data every 2 to 10 seconds while underway and every 3 minutes while at anchor (Perez et al., 2009) [1]. I collected 1,196,199 lines of raw AIS data, which were all the AIS data that AISHub had on May.01.2015. I listed the raw data fields and added some notes for each field. The notes were collected from a wide source, but catb.org [3] has most of the information. I also added two real examples, though some fields did not have real data. See the AIS raw data field, notes and examples in Table 2.1.
<table>
<thead>
<tr>
<th>NO.</th>
<th>Fields</th>
<th>Notes</th>
<th>Example1</th>
<th>Example2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>type</td>
<td>Type of ship/cargo, details about ship type in Table 2.2.</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>mmsi</td>
<td>Maritime Mobile Service Identity - a series of nine digits uniquely identifying ship stations.</td>
<td>273812710</td>
<td>266208000</td>
</tr>
<tr>
<td>3</td>
<td>imo</td>
<td>International Maritime Organization - ship identification number - a seven digit number that remains unchanged upon transfer of the ship's registration to another Country.</td>
<td>8610277</td>
<td>N/A</td>
</tr>
<tr>
<td>4</td>
<td>shipname</td>
<td>20 characters to represent the name of the vessel.</td>
<td>VSEVOLOD SIBIRTSEV</td>
<td>N/A</td>
</tr>
<tr>
<td>5</td>
<td>shiptype</td>
<td>Type of ship/cargo, details about ship type in Table 2.2.</td>
<td>30</td>
<td>N/A</td>
</tr>
<tr>
<td>6</td>
<td>shiptype_text</td>
<td>Text notes for the ship type.</td>
<td>Fishing</td>
<td>N/A</td>
</tr>
<tr>
<td>7</td>
<td>callsign</td>
<td>International radio call sign – up to seven characters, assigned to the vessel by its Country of registry.</td>
<td>UDVB</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Description</td>
<td>Value</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-----</td>
<td>-------------------------------------------------------------------------------</td>
<td>---------------</td>
<td>-----</td>
</tr>
<tr>
<td>8</td>
<td>timestamp</td>
<td>Coordinated Universal Time (UTC) time accurate to nearest second when this data was generated. Moreover, every 6 minutes the AIS transmitter sends additional fields.</td>
<td>2015-05-01 21:30:31 UTC</td>
<td>2015-05-01 06:46:45 UTC</td>
</tr>
<tr>
<td>9</td>
<td>lon</td>
<td>Longitude degrees.</td>
<td>N/A</td>
<td>11.16450024</td>
</tr>
<tr>
<td>10</td>
<td>lat</td>
<td>Latitude degrees.</td>
<td>N/A</td>
<td>58.80083466</td>
</tr>
<tr>
<td>11</td>
<td>speed</td>
<td>Speed over ground – 0 to 102 knots with 0.1 knot resolution.</td>
<td>N/A</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>course</td>
<td>Course over ground – relative to true north to 0.1 degree.</td>
<td>N/A</td>
<td>286</td>
</tr>
<tr>
<td>13</td>
<td>heading</td>
<td>True Heading – 0 to 359 degrees from gyro compass.</td>
<td>N/A</td>
<td>314</td>
</tr>
<tr>
<td>14</td>
<td>tagblock_station</td>
<td>Name of signal station.</td>
<td>rORBCOMM 009</td>
<td>rORBCOMM M000</td>
</tr>
<tr>
<td>15</td>
<td>seg_id</td>
<td>Segment identifier.</td>
<td>273812710-014-08-10T00:39:37.000000Z</td>
<td>266208000-2015-04-26T09:07:38.000000Z</td>
</tr>
<tr>
<td>16</td>
<td>draught</td>
<td>Vertical distance</td>
<td>7.900000095</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Table 2.1: AIS raw data

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>eta</td>
<td>Estimated time of arrival (ETA) at destination – UTC date hour: minute.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2015-04-17 23:23:00 UTC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>18</td>
<td>to_port</td>
<td>Dimension to port.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>19</td>
<td>destination</td>
<td>max 20 characters.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VLADIVOSTOK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>20</td>
<td>type_and_cargo</td>
<td>Cargo classification.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>21</td>
<td>to_bow</td>
<td>Dimension to bow.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>22</td>
<td>to_stern</td>
<td>Dimension to stern.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>149</td>
</tr>
<tr>
<td></td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>23</td>
<td>to_starboard</td>
<td>Dimension to starboard.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>24</td>
<td>status</td>
<td>Status of signal.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

Table 2.2: AIS ship type

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - Reserved for future use</td>
<td>0 - All ships of this type</td>
<td>-</td>
</tr>
<tr>
<td>2 - WIG</td>
<td>1 - Carrying DG, HS, or MP, IMO hazard or pollutant category X</td>
<td>-</td>
</tr>
<tr>
<td>3 - See right column</td>
<td>2 - Carrying DG, HS, or MP, IMO hazard or pollutant category Y</td>
<td>3 - Vessel</td>
</tr>
<tr>
<td>4 - HSC</td>
<td>3 - Carrying DG, HS, or MP, IMO hazard or pollutant category Z</td>
<td>-</td>
</tr>
<tr>
<td>5 - See above</td>
<td>4 - Carrying DG, HS, or MP, IMO hazard or pollutant category OS</td>
<td>-</td>
</tr>
<tr>
<td>6 - Passenger ships</td>
<td>6 - Reserved for future use</td>
<td>-</td>
</tr>
<tr>
<td>7 - Cargo ships</td>
<td>7 - Reserved for future use</td>
<td>-</td>
</tr>
<tr>
<td>8 - Tanker(8)</td>
<td>8 - Reserved for future use</td>
<td>-</td>
</tr>
<tr>
<td>9 - Other types of ship</td>
<td>8 - No additional information</td>
<td>-</td>
</tr>
</tbody>
</table>

For each specific ship, the raw AIS data contains several lines of records, each record contains all its information at a specific time. For example, the ship (mmsi = 273812710) has 71 lines of records on May.01.2015. The earliest record is at 2015-05-01 00:24:50 UTC, and the last record is at 2015-05-01 23:24:37 UTC. As I
wanted to visualize the global ship movements, it was too much data load, if I visualized every record of every ship. In addition, in a global view, the effect was not so obvious to use lots of points when the route was very short. Therefore, for each specific ship, I only picked its start and end point from the raw data, saved them into a new file. Finally, I only visualized the new file.

2.2 AIS Filtered Data

To filter the AIS raw data, the process of the algorithm is listed as follows:

1) Read the AIS raw data file, get all lines of data for one specific ship based on mmsi, order them by timestamp.

2) Get the Longitude and Latitude from the data which has the earliest timestamp, set them to slng and slat.

3) Get the Longitude and Latitude from the data which has the last timestamp, set them to elng and elat.

4) Save the five fields for each ship: mmsi, slng(start longitude), slat(start latitude), elng(end longitude) and elat(end latitude) to a new file, see field notes in Table 2.3. At the end, there are 15,670 lines of filtered data, that is to say, there are 15,670 ships on May.01.2015 that have data in AISHub system.

<table>
<thead>
<tr>
<th>NO.</th>
<th>Field</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mmsi</td>
<td>Maritime Mobile Service Identity - a series of nine digits uniquely identifying ship stations</td>
</tr>
<tr>
<td>2</td>
<td>slat</td>
<td>Start latitude</td>
</tr>
<tr>
<td>3</td>
<td>slng</td>
<td>Start longitude</td>
</tr>
<tr>
<td>4</td>
<td>elat</td>
<td>End latitude</td>
</tr>
<tr>
<td>5</td>
<td>elng</td>
<td>End longitude</td>
</tr>
</tbody>
</table>

Table 2.3: AIS filtered data
3. Web Development Library: Three.js

In this project, I made use of a popular Web data visualization library called Three.js to do the computer graphics. Three.js [4] is a cross-browser JavaScript library/API used to create and display animated 3D computer graphics in a web browser. WebGL is a JavaScript API for rendering interactive 3D graphics in modern web browsers without the use of plug-ins. Three.js is built on top of WebGL and it is a lightweight 3D library that hides a lot of the WebGL complexities and makes it very simple to get started with 3D programming on the web.

The following list shows some of the things that Three.js makes easy:

• Creating simple and complex 3D geometries
• Animating and moving objects through a 3D scene
• Applying textures and materials to your objects
• Loading objects from 3D modeling software
• Creating 2D sprite-based graphics

Three.js allows the creation of GPU-accelerated 3D animations using the JavaScript language as part of a website without relying on proprietary browser plugins. This is possible thanks to the advent of WebGL. High-level libraries such as Three.js or GLGE, SceneJS, PhiloGL or a number of other libraries make it possible to author complex 3D computer animations that display in the browser without the effort required for a traditional standalone application or a plugin.
4. Earth, Clouds and Stars

4.1 Scene, Camera and Renderer

To be able to display something with Three.js, we need three things: a scene, a camera and a renderer.

4.1.1 Scene

The scene is the container used to store and keep track of the objects (Earth and stars) we want to render.

4.1.2 Camera

The camera determines what we’ll see when we render the scene. To make a better (ordinary) perspective, I used the perspective camera, because it was more like the real world. As we can see in Figure 4.1, if we use the Orthographic camera, all the elements are of the same size and distance plays no role.

![Perspective VS Orthographic camera]

Figure 4.1: Perspective VS Orthographic camera[5]

It's necessary to totally understand each parameters of the perspective camera (Table 4.1), and adjust them to have a good view. See Figure 4.2 for my implementation of value.
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fov</strong></td>
<td>Field of view – this is part of scene that can be seen from the position of the camera. Humans have an almost 180-degree field of view, while some birds might even have a complete 360-degree field of view. However, for computers, we usually use the field of view between 60 and 90 degrees.</td>
</tr>
<tr>
<td><strong>Aspect</strong></td>
<td>The aspect ratio is the ratio between the horizontal and vertical size of the area where we render the output. As we usually use the entire window, we will just use that ratio. The ordinary value is <code>window.innerWidth / window.innerHeight</code></td>
</tr>
<tr>
<td><strong>Near</strong></td>
<td>This property defines a minimum distance from the camera the Three.js renders the scene. Usually, this is a very small value, e.g. 0.1</td>
</tr>
<tr>
<td><strong>Far</strong></td>
<td>This property defines a maximum distance we see the scene from the position of the camera. If we set this as too low, a part of our scene might not be rendered; if we set it as too high, in some cases, it might affect the rendering performance. A normal value is between 500 and 2000</td>
</tr>
</tbody>
</table>

Table 4.1: Perspective camera parameter [5]
4.1.3 Renderer

The renderer is responsible for rendering the scene in the browsers. Three.js supports different renderers like WebGL, Canvas, SVG and CSS 3D. Here I used the WebGL Render, because it's easier to handle after learning OpenGL. I used window width and height to allow our Earth to fill the browser window. See Figure 4.2 for how to use Three.js for scene, camera and renderer.

```javascript
scene = new THREE.Scene();
camera = new THREE.PerspectiveCamera(45, window.innerWidth / window.innerHeight, 0.01, 100);
camera.position.z = 1.5;
renderer = new THREE.WebGLRenderer();
renderer clearColor(0x000000, 1.0);
renderer setPixelRatio(window.devicePixelRatio);
renderer setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);
```

Figure 4.2: Three.js for scene, camera and renderer
4.2 Lighting

Three.js supports different light sources that have specific behavior and uses. In this project, I used ambient and directional light. See the lighting difference in Figure 4.3 Ambient VS Directional light.

![Figure 4.3: Ambient VS Directional light](image)

4.2.1 Ambient Light

Ambient light adds a constant color to the entire scene. It's the basic light which is applied globally. The dimmed ambient light shows areas away from the sun. Ambient lights only have one property: the color. See Figure 4.4 for the implementation of ambient light.

```javascript
scene.add(new THREE.AmbientLight(0x777777));
```

![Figure 4.4: Implementation of ambient light](image)
4.2.2 Directional Light

Directional lights are a common form of lighting when creating an outdoor scene, but can be used in any scene. Here I used it to mimic the sun. It’s a light that is very far away and shines in one direction. Just like the sun, because it’s so far away, all of the rays run parallel to each other. Also, because this light acts as if it’s far away, the position of the light does not matter, only the angle of the light.

See Figure 4.5, in the first line, I created the light with two properties: the first is the color of the light and the second is the intensity or brightness of the light. The second line changes the position of the light, and the final line adds it to our scene.

```
var directionalLight = new THREE.DirectionalLight(0xffffff, 0.15);
directionalLight.position.set(5, 3, 5);
scene.add(directionalLight);
```

**Figure 4.5: Implementation of directional light**
4.3 Creating Earth

4.3.1 Creating Sphere

The sphere is created using THREE.SphereGeometry. The first parameter is the radius. The second and third parameters are the number of width and height segments. See Figure 4.6. The sphere is drawn as a polygon mesh.

\[
\text{new THREE.SphereGeometry}(\text{radius}, \text{segments}, \text{segments}),
\]

Figure 4.6: create sphere geometry in Three.js

When creating sphere geometry, the more segments, the more vertices, edges and faces it needs to draw, will take more time to render. In this project, I chose 64 width/height segments. See the difference in Figure 4.7.

Figure 4.7: A sphere rendered with 8, 16 and 32 width/height segments

4.3.2 Texture Mapping

Texture mapping is a method for adding detail to a 3D object by applying an image to one or more of the faces of that object. This allows us to add fine detail without the need to model those details into our 3D object, thus keeping the polygon count down. This can be a huge performance booster when rendering our models [6].
4.3.3 Clear Earth without Clouds

![Clear Earth without Clouds](image)

**Figure 4.8: Clear Earth without cloud**

I'm using map data from shadedrelief.com. This is a great collection of shaded relief maps by cartographer Tom Patterson. Natural Earth III [7] is collection of raster map data tailored towards visualizing Earth from space. Compared to photographs of Earth, Natural Earth III offers brighter colors, fewer clouds and distinct environmental zones. The maps are very pleasant to look at.

There are two choices of Earth without cloud in Natural Earth III, 16,200 x 8,100 JPEG (30 MB) and 8,192 x 4,096 JPEG (9.1 MB). "Proper" discrete desktop GPUs will probably support 8,192 x 8,192 pixels (up to maximum of 16,384 x 16,384). If you want to be safe, it seems 2,048 x 2,048 is the minimum supported across a wide variety of systems (including integrated Intel GPUs and old discrete GPUs): For mobile devices, from above information it seems for iOS the maximum is 2,048 x 2,048, and for Android it's 4,096 x 4,096 pixels [8]. I reduced the image size to 4096 x 2048 pixels (Figure 4.8), which was the maximum texture size for the GPU of my computer.

The Projection information for the original 8,192 x 4,096 JPEG is listed as following:
Projection: Plate Carree aka Geographic or "LatLong"
Earth ellipsoid: Sphere, radius 6370997 m
Datum: WGS84
Extent: 180 West to 180 East, 90 North to 90 South
Size: 8,192 height samples wide x 4,096 high

The effect of texture mapping with clear Earth image is shown in Figure 4.9. I used THREE.MeshPhongMaterial to wrap map data around the sphere. See Figure 4.10 for implementation.

![Figure 4.9: Texture mapping with clear Earth image](image)

```javascript
function createEarth(radius, segments)
{
    return new THREE.Mesh(
        new THREE.SphereGeometry(radius, segments, segments),
        new THREE.MeshPhongMaterial({
            map: THREE.ImageUtils.loadTexture('images/no_cloud_surface.jpg'),
            bumpMap: THREE.ImageUtils.loadTexture('images/bump_surface.jpg'),
            bumpScale: 0.002,
            specularMap: THREE.ImageUtils.loadTexture('images/water.png'),
            specular: new THREE.Color('white')
        })
    );
}
```

![Figure 4.10: Implementation of creating clear Earth](image)
4.3.4 Bump Mapping

1) What is bump mapping?
Displacement mapping uses a texture or height map to cause an effect where the actual geometric position of points over the textured surface are displaced. Bump-mapping [9] is the process of creating the illusion of 3D depth by using a manipulated surface normal in the lighting, rather than actually creating the extra surface detail. See Figure 4.11 for the difference between displacement map and bump map.

![Figure 4.11: Displacement map VS Bump map](image)

The Most Straightforward Type of Bump-Mapping is Height Fields [9]. See Figure 4.12 for using bump map for a height field.
2) How to use bump mapping?

In this project, bump mapping was designed to simulate bumps and wrinkles on the surface of the Earth. Figure 4.13 is the bump mapping image used in this project. The result is an apparently bumpy surface rather than a smooth surface although the surface of the underlying object is not actually changed. See the difference with and without bump mapping in Figure 4.14.
In Figure 4.14, since the original image (left) contains shaded relief, so the bump map effect (right) is limited on this texture. It's necessary to know the details about shaded relief to help understand its effect. See the difference without and with shaded relief in Figure 4.15.
Shaded relief [10], or hill-shading, simulates the cast shadow thrown upon a raised relief map, or more abstractly upon the planetary surface represented. The shadows normally follow the English convention of top-left lighting in which the light source is placed near the upper-left corner of the map. If the map is oriented with north at the top, the result is that the light appears to come from the north-west. Many people have pointed out that this is unrealistic for maps of the northern hemisphere, because the sun does not shine from that direction, and they have proposed using southern lighting. However, the normal convention is followed to avoid multi-stable perception illusions (i.e. crater/hill confusion).
3) How to adjust bump effect?

We could adjust the bump effect (how much the map affects lighting) with the bumpScale parameter. BumpScale means how much the bump map affects the material. Typical ranges are 0-1. Default is 1. You could see how bumpScale affects the Earth in Figure 4.16. Implementation of bump mapping is shown in Figure 4.17.

```
function createEarth(radius, segments)
{
    return new THREE.Mesh(  
        new THREE.SphereGeometry(radius, segments, segments),  
        new THREE.MeshPhongMaterial({  
            map: THREE.ImageUtils.loadTexture('images/no_cloud_surface.jpg'),  
            bumpMap: THREE.ImageUtils.loadTexture('images/bump_surface.jpg'),  
            bumpScale: 0.002,  
            specularMap: THREE.ImageUtils.loadTexture('images/water.png'),  
            specular: new THREE.Color('white')  
        })
    );
}
```

Figure 4.16: How bump Scale affects the Earth

Figure 4.17: Implementation of Bump Mapping
4.3.5 Reflective Ocean and Lakes

1) What are specular maps?
Specular maps [11] are the maps you use to define a surface's shininess and highlight color. The higher the value of a pixel (from black to white), the shinier the surface will appear. Therefore, surfaces such as dry stone or cotton fabric would tend to have a very dark specular map, while surfaces like polished chrome or plastic would tend to have lighter specular maps.

2) How to use a specular map?
In this project, I wanted to make the ocean and lakes reflective by applying a land/water mask. See Figure 4.18. The color of ocean and lakes are white, which have higher value of pixels than the land color black, so the ocean and lakes will have shiner surfaces. See the different effect in Figure 4.19.

Figure 4.18: Land/water mask
3) **How to adjust Specular color?**

In addition, we can control the specular color with specular parameter. In this project, I chose white instead of yellow, which is closer to the real sunshine. See color difference in Figure 4.20. Implementation of Specular Map is shown in Figure 4.21.
function createEarth(radius, segments)
{
    return new THREE.Mesh(
        new THREE.SphereGeometry(radius, segments, segments),
        new THREE.MeshPhongMaterial(
            map: THREE.ImageUtils.loadTexture('images/no_cloud_surface.jpg'),
            bumpMap: THREE.ImageUtils.loadTexture('images/bump_surface.jpg'),
            bumpScale: 0.002,
            specularMap: THREE.ImageUtils.loadTexture('images/water.png'),
            specular: new THREE.Color('white'))
    );
}
4.4 Adding Clouds

4.4.1 Separate Clouds and Earth

There are two methods to have the cloud effect, either directly texture mapping an image that has clouds in it (see Figure 4.22) or texture mapping the clear Earth (see Figure 4.8) and pure clouds (see Figure 4.23) separately. See the different effects in Figure 4.24. In this project, I chose to texture mapping them separately. In this way, I could control the clouds independently. For example, when the cloud rotates automatically at a different speed than the Earth, you could see different views of Earth at different times from space. It's closer to reality since the cloud is moving separately from the Earth.

![Figure 4.22: Earth with clouds](image)

Figure 4.22: Earth with clouds
Figure 4.23: Cloud with transparent background

Texture mapping an Earth with cloud  
Texture mapping Earth and cloud separately

Figure 4.24: Different effect of texture mapping cloud
4.4.2 Clouds Implementation

According to the layers of atmosphere (see Figure 4.25), the Earth radius is 6371km and the clouds are 16km above the Earth, which is still in the Troposphere. Therefore, I set the cloud sphere geometry radius equal to Earth radius plus 0.0025. The transparent value set to be true, otherwise we could only see the clouds since it was around the Earth. See implementation in Figure 4.26.

![Figure 4.25: Layers of Atmosphere](image)

```javascript
function createClouds(radius, segments) {
  return new THREE.Mesh(
    new THREE.SphereGeometry(radius + 0.0025, segments, segments),
    new THREE.MeshPhongMaterial({
      map: THREE.ImageUtils.loadTexture('images/clouds.png'),
      transparent: true
    })
  );
}
```

![Figure 4.26: Implementation of adding cloud](image)
4.5 Adding Stars

The starfield is created by adding a much larger sphere around the Earth and projecting the star texture on the inner side. Similar to adding clouds, texture mapping a galaxy image (Figure 4.27) to a much bigger sphere. I set the sphere radius to be 90, comparing to the Earth radius of 0.5. This way, the Earth looks like it is inside the galaxy (Figure 4.28). For implementation (Figure 4.29), the side property defines which side of faces will be rendered - front, back or both. The default is THREE.FrontSide, other options are THREE.BackSide and THREE.DoubleSide.

Figure 4.27: Galaxy Image

Figure 4.28: Earth in the galaxy
function createStars(radius, segments) {
    return new THREE.Mesh(
        new THREE.SphereGeometry(radius, segments, segments),
        new THREE.MeshBasicMaterial({
            map: THREE.ImageUtils.loadTexture('images/stars.png'),
            side: THREE.BackSide
        })
    );
}

Figure 4.29: Implementation of galaxy
4.6 Rotate, Zoom and Translate

I added the trackball controls plugin by Eberhard Graether, which allowed to rotate, zoom and translate the scene. TrackballControls.js [12] is in the js sub-directory of the examples directory. It is part of the examples -- not the library. You must include it explicitly in your project. You are free to modify it to your liking. You may also want to consider OrbitControls, which is appropriate if your scene has a natural "up" direction.

For implementation, the minDistance is the minimum the zoom level, the maxDistance is the maximum zoom level. If the zoom level (minDistance) is too small, the view might have framebuffer problem. I tried a minDistance value of 0.75, which was the shortest distance that close to the Earth in this project. See Figure 4.30. It's necessary to understand the meaning of zoom level. Know its relationship with other common attributes, you could refer to Table 4.2.

```javascript
controls = new THREE.TrackballControls(camera, renderer.domElement);
controls.rotateSpeed = 0.1;
controls.noZoom = false;
controls.noPan = true;
controls.staticMoving = false;
controls.minDistance = 0.75;
controls.maxDistance = 3.0;
```

Figure 4.30: Implementation of Rotate, Zoom and Translate
In Table 4.2, the 'degree' column gives the map width in degrees, for a map at that zoom level which is 256 pixels wide. Values listed in the column "m / pixels" gives the number of meters per pixel at that zoom level. These values for "m / pixel" are calculated with an Earth radius of 6372.7982 km at the equator. For other latitudes the values must be multiplied by the cosine of the latitude. "Scale" (map scale) is only an approximate size comparison and refers to distances on the equator. In addition, the map scale will be dependent on the monitor. These values are for a monitor with a 0.3 mm / pixel (85.2 pixels per inch or PPI). The '# Tiles' column indicates the number of tiles needed to show the entire world. This is useful when calculating storage requirements for pre-generated tiles [13].
5. Ship Visualization

5.1 Ship Point Cloud

There are thousands of ships to visualize. Each ship is a point. The collection of these points is a point cloud. Therefore, I use THREE.PointCloud to visualize them. The constructor of THREE.PointCloud takes two properties: a geometry and a material. See Figure 5.1.

```javascript
return new THREE.PointCloud(pointBufGeo, shaderMaterial);
```

Figure 5.1: Use THREE.PointCloud

5.1.1 Point Cloud Geometry

The geometry defines where the individual particles are positioned. Each vertex and each point used to define the geometry is shown as a particle. For the geometry part, the collection of points, I used THREE.BufferGeometry to save all the information. See Figure 5.2.

THREE.BufferGeometry [13] is an efficient method to Geometry, because it stores all data, including vertex positions, face indices, normals, colors, UVs, and custom attributes within buffers. This reduces the cost of passing all this data to the GPU. This also makes BufferGeometry harder to work with than Geometry. Rather than accessing position data as Vector3 objects, color data as Color objects, and so on, you have to access the raw data from the appropriate attribute buffer. This makes BufferGeometry best-suited for static objects where you don't need to manipulate the geometry much after instantiating it.

```javascript
pointBufGeo = new THREE.BufferGeometry();
pointBufGeo.addAttribute('position', new THREE.BufferAttribute(positions, 3));
pointBufGeo.addAttribute('customColor', new THREE.BufferAttribute(colors, 3));
pointBufGeo.addAttribute('size', new THREE.BufferAttribute(sizes, 1));
```

Figure 5.2: Using THREE.BufferGeometry
5.1.2 Point Cloud Material

1) Shader material object

The material is used to color and texture the particles. I wanted to implement an effect not included with any of the built-in materials and combine many objects into BufferGeometry in order to improve performance. THREE.ShaderMaterial is a good choice. See Figure 5.3.

```javascript
var shaderMaterial = new THREE.ShaderMaterial({
  uniforms: uniforms,
  attributes: attributes,
  vertexShader: document.getElementById('vertexshader').textContent,
  fragmentShader: document.getElementById('fragmentshader').textContent,
  blending: THREE.AdditiveBlending,
  depthWrite: false,
  transparent: true
});
```

Figure 5.3: Using THREE.ShaderMaterial

THREE.ShaderMaterial is a material rendered with custom shaders. A ShaderMaterial will only be rendered properly by WebGLRenderer, since the GLSL code in the vertexShader and fragmentShader properties must be compiled and run on the GPU using WebGL [14].
2) Vertex and Fragment shader

A shader is a small program written in GLSL that runs on the GPU. In this project, I used two kind of shaders, Vertex shaders and Fragment shaders. See the relationship between them in the shader-style pipeline (Figure 5.4).

The Basic Computer Graphics Pipeline, Shader-style

As you can see in Figure 5.4, the vertex shader runs first, it receives attributes, calculates / manipulates the position of each individual vertex, and passes additional data (varyings) to the fragment shader; The fragment (or pixel) shader runs second, it sets the color of each individual ”fragment” (pixel) rendered to the screen.

In this project, the vertex shader is implemented as Figure 5.5 and fragment shader is implemented as Figure 5.6.
There are three types of variables in shaders[15]: uniforms, attributes, and varyings:

**Uniforms** are variables that have the same value for all vertices - lighting, fog, and shadow maps are examples of data that would be stored in uniforms. Uniforms can be accessed by both the vertex shader and the fragment shader.

**Attributes** are variables associated with each vertex---for instance, the vertex position, face normal, and vertex color are all examples of data that would be stored in attributes. Attributes can only be accessed within the vertex shader.

**Varyings** are variables that are passed from the vertex shader to the fragment shader. For each fragment, the value of each varying will be smoothly interpolated from the values of adjacent vertices.
Note that within the shader itself, uniforms and attributes act like constants; you can only modify their values by passing different values to the buffers from your JavaScript code. For example, in this project, uniform variable color and texture, the value were modified from JavaScript code as Figure 5.7.

```javascript
var uniforms = {
    color: {
        type: "c",
        value: new THREE.Color(0xffffffff)
    },
    texture: {
        type: "t",
        value: THREE.ImageUtils.loadTexture("images/point.png")
    }
};
```

**Figure 5.7: Uniform variables modified in JavaScript**
5.2 Ship Spline Curve

Now I need to define a smooth curve passing through the start and end points that I’ve defined above. I chose to add some control points between these two points, and then generate a spline curve passing through all these points.

5.2.1 Spline Curve and Control Points

A spline curve [16] is a mathematical representation for which it is easy to build an interface that will allow a user to design and control the shape of complex curves and surfaces. The general approach is that the user enters a sequence of points, and a curve is constructed whose shape closely follows this sequence. The points are called control points. A curve that actually passes through each control point is called an interpolating curve; a curve that passes near to the control points but not necessarily through them is called an approximating curve. See Figure 5.8. In this project, I chose interpolating curves, I used spherical interpolation not linear interpolation, since the latter one might pass though the Earth.

![Figure 5.8: Two kinds of Spline curve](image-url)
5.2.2 Spherical Linear Interpolation

In computer graphics, Slerp is shorthand for spherical linear interpolation, introduced by Ken Shoemake in the context of quaternion interpolation for the purpose of animating 3D rotation. Using Spherical interpolation could avoid points passing through the sphere. Then how do we get the control points positioned in spherical interpolation?

5.2.3 Control Points Position

![Figure 5.9: Spherical Interpolation](image)

In Figure 5.9, let \( p_0 \) and \( p_1 \) be the start and end points of the arc, and let \( t \) be the parameter, \( 0 \leq t \leq 1 \). Compute \( \Omega \) as the angle subtended by the arc, so that \( \cos \Omega = p_0 \cdot p_1 \), the n-dimensional dot product of the unit vectors from the origin to the ends. The geometric formula [17] is then:

\[
\text{Slerp}(p_0, p_1; t) = \frac{\sin[(1 - t)\Omega]}{\sin \Omega} p_0 + \frac{\sin[t\Omega]}{\sin \Omega} p_1
\]

The symmetry can be seen in the fact that \( \text{Slerp}(p_0, p_1; t) = \text{Slerp}(p_1, p_0; 1 - t) \). In the limit as \( \Omega \to 0 \), this formula reduces to the corresponding symmetric formula for linear interpolation.

\[
\text{Lerp}(p_0, p_1; t) = (1 - t)p_0 + tp_1
\]

In this way, the control points positions could be calculated based on the start and end points, but how to calculate the central angle \( \Omega \) in this Slerp formula?
5.2.4 Central Angle in Slerp

The formula for calculations is based on a spherical Earth (ignoring ellipsoidal effects), which is accurate enough for most purposes. See Figure 5.10, a perspective view of the Earth showing how latitude (\(\varphi\)) and longitude (\(\lambda\)) are defined on a spherical model. In fact, the Earth is very slightly ellipsoidal, but using a spherical model gives errors typically only up to 0.3\% [18]. This error is acceptable for this project.

![Perspective view of spherical model](image)

Figure 5.10: Perspective view of spherical model

Calculate the central angle \(\Omega\) between two points. This comes from Haversine formula [19].

\[
\Omega = 2\arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_1 - \varphi_0}{2}\right) + \cos(\varphi_0) \cos(\varphi_1) \sin^2\left(\frac{\lambda_1 - \lambda_0}{2}\right)}\right)
\]

In this formula, \(\Omega\) is the central angle, \(\varphi_0\) is the start point latitude, \(\varphi_1\) is the end point latitude, \(\lambda_0\) is the start point longitude, and \(\lambda_1\) is the end point longitude.
5.2.5 Latitude Longitude to XYZ

For further calculation, we need to know more details about the latitude and longitude. Latitude and longitude [20] are imaginary lines drawn on maps to easily locate places on the Earth. Latitude is the distance north or south of the equator (an imaginary circle around the Earth halfway between the North Pole and the South Pole) and longitude is the distance east or west of the prime meridian (an imaginary line running from north to south through Greenwich, England). Both are measured in terms of the 360 degrees (symbolized by °) of a circle.

The Equator is the line of 0° latitude, the starting point for measuring latitude. The latitude of the North Pole is 90° N, and that of the South Pole is 90° S. The latitude of every point in between must be some degree north or south, from 0° to 90°. One degree of latitude covers about 69 miles (111 kilometers).

Longitude is measured in degrees east or west of the prime meridian. This means one half of the world is measured in degrees of east longitude up to 180°, and the other half in degrees of west longitude up to 180°. See Figure 5.11, to understand latitudes and longitudes.

Figure 5.11: Latitude and longitude
If you are given latitude and longitude (in degrees) for some place on Earth. Latitude is between -90 and 90, and longitude is between -180 and 180. Suppose that the model of the Earth you're using has radius R. Then the following formulae produce x y z coordinates for that place:

\[
\begin{align*}
\text{LAT} &= \text{latitude} \times \pi/180 \\
\text{LON} &= \text{longitude} \times \pi/180 \\
x &= R \times \cos(\text{LAT}) \times \cos(\text{LON}) \\
y &= R \times \sin(\text{LAT}) \\
z &= -R \times \cos(\text{LAT}) \times \sin(\text{LON})
\end{align*}
\]

In this project, the implementation of transferring latitude and longitude to x, y, z is showed in Figure 5.12.

```javascript
function latLngToXYZ(lat, lng, radius) {
    var LAT = lat * Math.PI / 180;
    var LNG = lng * Math.PI / 180;

    return {
        x: radius * Math.cos(LAT) * Math.cos(LNG),
        y: radius * Math.sin(LAT),
        z: -radius * Math.cos(LAT) * Math.sin(LNG)
    };
}
```

Figure 5.12: Transfer latitude and longitude to x, y, z
5.2.6 Latitude and Longitude of Control Points

First get its x, y, z based on spherical interpolation formulas and rules of transferring latitude and longitude to x, y, z, and finally transfer x, y, z to latitude and longitude. Pseudocode shows as following:

\[
\begin{align*}
\text{let } A &= \frac{\sin[(1 - t)\Omega]}{\sin \Omega} \\
\text{let } B &= \frac{\sin[t\Omega]}{\sin \Omega} \\
\end{align*}
\]

\[
\begin{align*}
x &= A \cdot \cos(LAT_1) \cdot \cos(LON_1) + B \cdot \cos(LAT_2) \cdot \cos(LON_2) \\
y &= A \cdot \sin(LAT_1) + B \cdot \sin(LAT_2) \\
z &= -1 \cdot (A \cdot \cos(LAT_1) \cdot \sin(LON_1) + B \cdot \cos(LAT_2) \cdot \sin(LON_2)) \\
\end{align*}
\]

\[
\begin{align*}
\text{lat} &= \text{atan2}(y, \sqrt{x^2 + z^2}) \cdot 180 / \pi \\
\text{lng} &= (-1) \cdot \text{atan2}(z, x) \cdot 180 / \pi \\
\end{align*}
\]

The implementation is shown in Figure 5.13.
function InnerPointLatLng(lat1, lng1, lat2, lng2, t) {

    var LAT1 = lat1 * Math.PI / 180.0;
    var LNG1 = lng1 * Math.PI / 180.0;
    var LAT2 = lat2 * Math.PI / 180.0;
    var LNG2 = lng2 * Math.PI / 180.0;

    var A = Math.sin((1 - t) * omega) / Math.sin(omega);
    var B = Math.sin(t * omega) / Math.sin(omega);

    var x = A * Math.cos(LAT1) * Math.cos(LNG1) + B * Math.cos(LAT2) * Math.cos(LNG2);
    var y = A * Math.sin(LAT1) + B * Math.sin(LAT2);

    var lat = Math.atan2(y, Math.sqrt(Math.pow(x, 2) + Math.pow(z, 2))) * 180 / Math.PI;
    var lng = (-1) * Math.atan2(z, x) * 180 / Math.PI;

    return {
        lat: lat,
        lng: lng
    };
}

Figure 5.13: Get latitude and longitude of control points
5.2.7 Spline Curve with Control Points

Each ship’s trail is a spline curve, consisting of 8 control points. See Figure 5.14. Here I used a global variable array `path_splines` to save all of the spline curve.

```javascript
function generateControlPoints(radius) {
    for (var i = 0; i < length; ++i) {
        var s_lat = vessels[i]['s_lat'];
        var s_lng = vessels[i]['s_lng'];
        var e_lat = vessels[i]['e_lat'];
        var e_lng = vessels[i]['e_lng'];
        var points = [];
        var num_pnts = 8;
        for (var j = 0; j < num_pnts + 1; j++) {
            var t = j / num_pnts;
            var latlng = InnerPointLatLng(s_lat, s_lng, e_lat, e_lng, t);
            var arc_radius = radius + 0.005;
            var posXYZ = latlngToXYZ(latlng.lat, latlng.lng, arc_radius);
            points.push(new THREE.Vector3(posXYZ.x, posXYZ.y, posXYZ.z));
        }
        var spline = new THREE.SplineCurve3(points);
        path_splines.push(spline);
        var arc_length = spline.getLength();
        distance.push(arc_length);
        setShipTimes(i);
    }
}
```

Figure 5.14: Spline Curve with control points
5.3 Ship Route

After getting the spline curve, I could use SplineCurve3.getPoint(t) function to get points from this spline curve. Then I could use these points to draw the ship route. I used THREE.Line to draw the ship route, which is rendered using gl.LINE_STRIP instead of gl.LINES. For the geometry, I used THREE.BufferGeometry. For line material, I used THREE.LineBasicMaterial. See Figure 5.15 for implementation.

```javascript
var lineBufGeom = new THREE.BufferGeometry();

lineBufGeom.addAttribute('position', new THREE.BufferAttribute(vertices, 3));
lineBufGeom.addAttribute('color', new THREE.BufferAttribute(colors, 3));
lineBufGeom.computeBoundingSphere();

var line_material = new THREE.LineBasicMaterial({
  color: 0xffffff,
  vertexColors: THREE.VertexColors,
  transparent: true,
  opacity: lineOpacity,
  depthTest: true,
  depthWrite: false,
  linewidth: 0.003
});

return new THREE.Line(lineBufGeom, line_material, THREE.LinePieces);
```

Figure 5.15: Draw spline curve

The position attribute of the BufferGeometry, is set by an array called vertices. The vertices save all the position information of the points. The more points set, the more smooth the line will be. See Figure 5.16. These vertices position are obtained from SplineCurve3.getPoint(t), this function returns a vector for point t of the curve where t is between 0 and 1. The SplineCurve was generated in 5.2.7.

Similar to the position attribute, the color attribute of the BufferGeometry is also set by an array called colors. The colors stores the red, green, and blue channels of vertex color of each vertex in this geometry. So the vertices and colors variables are set at the same time within the same loop.
```javascript
var ctrl_pnts = 32;
var vertices = new Float32Array(length * 6 * ctrl_pnts);
var colors = new Float32Array(length * 6 * ctrl_pnts);
for (var i = 0; i < length; ++i)
{
    for (var j = 0; j < ctrl_pnts - 1; ++j)
    {
        var s_pos = path_splines[i].getPoint(j / (ctrl_pnts - 1));
        var e_pos = path_splines[i].getPoint((j + 1) / (ctrl_pnts - 1));
        vertices[(i * ctrl_pnts + j) * 6 + 0] = s_pos.x;
        vertices[(i * ctrl_pnts + j) * 6 + 1] = s_pos.y;
        vertices[(i * ctrl_pnts + j) * 6 + 2] = s_pos.z;
        vertices[(i * ctrl_pnts + j) * 6 + 3] = e_pos.x;
        vertices[(i * ctrl_pnts + j) * 6 + 4] = e_pos.y;
        vertices[(i * ctrl_pnts + j) * 6 + 5] = e_pos.z;
        colors[(i * ctrl_pnts + j) * 6 + 0] = 0.0;
        colors[(i * ctrl_pnts + j) * 6 + 1] = 1.0;
        colors[(i * ctrl_pnts + j) * 6 + 2] = 0.5;
        colors[(i * ctrl_pnts + j) * 6 + 3] = 0.0;
        colors[(i * ctrl_pnts + j) * 6 + 4] = 1.0;
        colors[(i * ctrl_pnts + j) * 6 + 5] = 0.5;
    }
}
```

Figure 5.16: Set BufferGeometry attribute at the same time
5.4 Ship Animation

5.4.1 Ship Movements

How to mimic ship movement effect like accelerate the first halfway and decelerate the second halfway?

The position of the ship is always updated by the update_ships function in animate function. Find a mathematical method that could generate a value between a range, meanwhile, the value is increasing at the beginning, after reaching the middle point then decreasing. Finally, I decided to use quadratic easing in/out [21], which accelerates the first halfway, then deceleration the second half.

In this project, the return value of easeInOutQuad function (see Figure 5.17), increases from 0 to 1 (first half accelerates and second half decelerates) then decrease from 1 to negative (first half accelerates and second half decelerates), when it goes to a negative value, I set it to 0.

```plaintext
function easeInOutQuad(t, b, c, d)
{
    t /= d / 2;
    if (t < 1)
        return c / 2 * t * t + b;
    --t;
    return -c / 2 * (t * (t - 2) - 1) + b;
}
```

- t: current time, b: start value, c: change in value, d: duration
- t and d can be frames or seconds/milliseconds

**Figure 5.17: Implementation of Quadratic easing in/out**
I used this ease_val as a parameter to get points from the spline curve (see 5.2.7), and then set this point position as the current active point position. See Figure 5.20 for implementation. This way could mimic the ship departing from the start point, accelerates in the first half and decelerates the second half. See Figure 5.21. After arriving the destination, then returning back to the departure position, it also accelerates in the first and decelerates the second half.

```javascript
function update_ships()
{
    pointBuffGeo.attributes.position.needsUpdate = true;

    for (var i = 0; i < length; ++i)
    {
        if (Date.now() > startTime[i])
        {
            var ease_val = easeInOutQuad(Date.now() - startTime[i], 0, 1, endTime[i] - startTime[i]);

            if (ease_val < 0 || speed_changed)
            {
                ease_val = 0;
                setShipTimes(i);
            }

            var pos = path_splines[i].getPoint(ease_val);
            positions[3 * i + 0] = pos.x;
            positions[3 * i + 1] = pos.y;
            positions[3 * i + 2] = pos.z;
        }
    }
}
```

**Figure 5.20: Implementation for getting ship position**

![Graph showing speed over time](image)

**Figure 5.21 Accelerates 1st half and decelerates 2nd half**
5.4.2 Scene Animation

The Earth, clouds, ships and ship routes are all rotating around its axis by increasing the y parameter. Clouds rotates faster than others, since normally clouds are moving faster than the Earth. The animate function is called at specific intervals defined by the browser using requestAnimationFrame [22] function. Therefore, I only need to call the animate() function once, when we've done initializing the scene. See Figure 5.17.

```javascript
function start_app()
{
    init();
    animate();
}
```

**Figure 5.17: Call animate only one time**

In the animate() function itself, we use requestAnimationFrame to schedule the next animate. See Figure 5.18. This way the browser will make sure the animate() function is called at the correct interval (usually around 60 times a second). This would call the specified function once every set interval. Before requestAnimationFrame was added to the browsers, usually setInterval(function, interval) or setTimeout(function, interval) was used. The problem with this approach is that it doesn't take into account what else is going on. Even if your animation isn't shown or is in a hidden tab, it is still called and it uses resources. Another issue is that these functions update the screen whenever they are called, not when it is the best time for the browser. This, once again, means higher CPU usage. With requestAnimationFrame, we don't tell the browser when it needs to update the screen. We ask the browser to run the supplied function when it's most opportune. Usually this results in a frame rate of about 60 fps. With requestAnimationFrame, the animations will run more smoothly and will be more CPU and GPU resource friendly and you don't have to worry about timing issues yourself.
function animate(time) {
    requestAnimationFrame(animate);
    if (!is_loading) {
        controls.update();
        update_ships();
    }
    stats.update();
    clouds.rotation.y += 0.001;
    earth.rotation.y += 0.0005;
    ships.rotation.y += 0.0005;
    path_lines.rotation.y += 0.0005;
    renderer.render(scene, camera);
}

Figure 5.18: Scene Animation
6. GUI Design and Implementation

I used a third party library called dat.GUI to do the GUI. dat.GUI [23] is a lightweight graphical user interface for changing variables in JavaScript.

6.1 Changing Ship Point Size

Using this GUI, the user could change the ship point size. See Figure 6.1. For implementation, I set GUI size value to each ship, which is represented as a global variable size. See Figure 6.2.

```javascript
var gui = new dat.GUI();

gui.add(this, 'pointSize', 0.01, 0.2).name("Size").onChange(function(value)
{
    pointBufGeo.attributes.size.needsUpdate = true;
    for (var i = 0; i < length; ++i)
    {
        sizes[i] = pointSize;
    }
});
```

Figure 6.1: GUI control ship size

Figure 6.2: Changing ship point size
6.2 Changing Ship Speed

Using this GUI, the user could change the ship movement speed. The user could directly modify the speedScale value (Figure 6.3), this value will affect the duration time for each ship. The bigger the speedScale value is, the less duration time the ship will take to finish its route. See setShipTimes function (Figure 6.4). After the ship returns to its departing place (ease_val<0), it will update its duration time (setShipTimes will be called, refer Figure 5.20).

```javascript
  gui.add(this, 'speedScale', minScale, maxScale).name("Speed").onFinishChange(function(value)
  {
      speed_changed = true;
      update_ships();
      speed_changed = false;
  });
```

**Figure 6.3:** Change ship speed from GUI

```javascript
function setShipTimes(index)
{
    var start_time = Date.now() + 5000;
    startTime[index] = start_time;
    var scale = (speedScale - minScale) / (maxScale - minScale);
    var duration = (1-scale) * distance[index] * 80000;
    endTime[index] = start_time + duration;
}
```

**Figure 6.4:** SpeedScale value affects duration of each ship
6.3 Changing the Opacity of Ship Route

Using this GUI, the user could change the opacity of the ship route. See effect in Figure 6.5. path_lines is a global variable. It is an object of THREE.Line. Its material has an opacity attribute. Therefore, setting the GUI value to this opacity attribute could change the opacity of the ship route. See Figure 6.6 for implementation.

![GUI change ship path opacity](image1.png)

**Figure 6.5: GUI change ship path opacity**

```javascript
gui.add(this, 'lineOpacity', 0, 1.0).name("Track Opacity").onChange(function(value) {
    path_lines.material.opacity = value;
});
```

**Figure 6.6: Change ship route opacity**
6.4 About Window

From the GUI, the user could click my name Hongyan Yi, then a new window will open to show some information about this project and author. See Figure 6.7. For implementation, I added an handle_About event (Figure 6.8), when the user clicks my name, this function will be triggered. Then it will call a show_about function (Figure 6.9). This function will decide whether to show the extra background, and the About window. The content of the About is in index.html file. See Figure 6.10. The styles of background and About window are set in style.css.

```
gui.add(this, "handle_About").name(“Hongyan Yi | Credits”);
```

Figure 6.7: About window

Figure 6.8: GUI show About window
function show_about(visible) {
    if (visible) {
        document.getElementById("about_box_bkg").className = "show";
        document.getElementById("about_box").className = "show";
        document.getElementById("about_box").style.pointerEvents = "all";
    } else {
        document.getElementById("about_box_bkg").className = "hide";
        document.getElementById("about_box").className = "hide";
        document.getElementById("about_box").style.pointerEvents = "none";
    }
}

Figure 6.9: Implementation of show About window and extra background

Figure 6.10: Html of About window and extra background
7. Resizing Window

The goal of this feature is that if I resize the window after it is loaded, the whole camera should also be adaptable to the window. See the effect in Figure 7.1. This requires the camera's aspect ratio, projection matrix and the renderer's size, to be updated if the window is resized. We can specify a function to be run by adding an event listener to the window's resize event, like this:

```javascript
window.addEventListener('resize', onWindowResize, false);
```

The `onWindowResize` function is implemented as Figure 7.2.

![Camera adaptable to the window size](image1)

**Figure 7.1: Camera adaptable to the window size**
```javascript
function onWindowResize()
{
    camera.aspect = window.innerWidth / window.innerHeight;
    camera.updateProjectionMatrix();
    renderer.setSize(window.innerWidth, window.innerHeight);
}
```

Figure 7.2: Implementation of onWindowResize event
8. Stats Counter

A number of Three.js applications examples have a neat little stats counter showing the number of frames per second, and displaying a graph of the frame rate over the last few seconds. This information is invaluable when testing the performance of a 3D application [24].

The JavaScript library responsible for this stats counter is not actually part of Three.js. It is a separate library.

If you click on the stats counter, you will switch between two views (Figure 8.1):
1) FPS Frames rendered in the last second. The higher the number the better.
2) MS Milliseconds needed to render a frame. The lower the number the better [25].

![Figure 8.1: Two views of stats counter](image)

With the stats.js library loaded, now create the stats counter and add it to the HTML page in JavaScript. The following lines are added to the end of the init() function.

Note that I'm appending the stats element directly to the body of the HTML page. Because I specified absolute positioning that is applied to the stats element, it will not interfere with the position of the renderer.

```javascript
stats = new Stats();
stats.domElement.style.position = 'absolute';
stats.domElement.style.top = '0px';
document.body.appendChild( stats.domElement );
```

The final step is to update the stats counter every time a frame is rendered. This is done with the following code added to the end of the animate() function.

```javascript
stats.update();
```
9. Loading Overlay

I created an overlay while the application is rendering the scene (Figure 9.1), at this time the overlay element could handle mouse events. After finish rendering, the overlay will be hidden, and will not handle a mouse event. At this time, the DOM will be controlled by THREE.TrackballControls.

![Figure 9.1: Show overlay while loading](image)

First add loading_overlay element into HTML as a div, which has an image, hide/show class, as following:

```
<div id="loading_overlay" class="hide"><img id="loading_overlay_img" src="images/loading.gif"></div>
```

For handling mouse events, I used CSS property pointer-events [26]. It allows authors to control under what circumstances a particular graphic element can become the target of mouse events. When the pointer-events value equals all, the element can only be the target of a mouse event when the pointer is over the interior (i.e., fill) or the perimeter (i.e., stroke) of the element. When the pointer-events value equals none, the element is never the target of mouse events; however, mouse events may target their descendant elements if those descendants have pointer-events set to some other
value. In these circumstances, mouse events will trigger event listeners on this parent element as appropriate on their way to/from the descendant during the event capture/bubble phases.

The following Figure 9.2 is the implementation of how loading_overlay element is controlled by show_loading function. The show_loading function is called before rendering the scene with show_loading(true), and after rendering the scene with show_loading(false).

```javascript
function show_loading(visible) {
    if (visible) {
        is_loading = true;
        document.getElementById("loading_overlay").className = "show";
        document.getElementById("loading_overlay").style.pointerEvents = "all";
    } else {
        is_loading = false;
        document.getElementById("loading_overlay").className = "hide";
        document.getElementById("loading_overlay").style.pointerEvents = "none";
    }
}
```

**Figure 9.2: Implementation of show_loading function**
10. Conclusions and Future Work

To summarize, I successfully reached my original goal to visualize the AIS data on a 3D Earth. One of the significant findings was that currently there was no free project to animate global AIS data. Even a commercial website such as fleetmon was focused more on data visualization but less on animation. This made the project progress much rougher, but more valuable and interesting. The data was real AIS data that I collected from the OSU’s CEOAS. Although it was only one day data, the 1,196,199 lines of raw data was pretty large. The most challenging part was calculating the control points position of the ship on the 3D Earth, which needed to figure out considerate amount of mathematic information. I used these control points to get spline curve, used spline curve to get points, and finally used these points to draw the route and visualize the movements. To mimic ship animation for accelerating and decelerating was also very interesting and challenging. The whole project implementation was time consuming. This was my first time to deal with a Web graphics application. The process to learn Html, JavaScript, CSS, Three.js was interesting. It was good that I already had a Computer Graphics background. This made drawing the Earth, clouds and stars parts easier. In addition, handling the Shader part was easier to pick up after taking OSU’s Shader class.

After digging into this project, I found that there were lots of options to extend this project in the future. For example, adding different Earth textures to let users choose whichever they like; Adding more days of data to track the ship route in a longer time; Visualizing the AIS data in group by type, country, speed and so on; Visualizing the AIS data in 2D maps with different map projections; Visualizing in different precision based on different zoom levels. For instance, when zoom level is very low, use more real ship positions to visual the route. This way the route is closer to reality. When zoom level is very high, use less real ship positions to visualize to decrease the GPU loading, since the difference is not obvious in this situation.
References


[16] https://people.cs.clemson.edu/~dhouse/courses/405/notes/splines.pdf


[23] https://github.com/dataarts/dat.gui


[25] https://github.com/mrdoob/stats.js/