Original Proposal: Utilizing webGL and Three.js for 3D Modeling

Julian A. Soto Perez
julian.soto1@upr.edu
University of Puerto Rico - Mayaguez
Mayaguez, Puerto Rico, USA

Abstract

This paper presents a comprehensive exploration of Three.js, a JavaScript library for 3D rendering, through the development of a realistic solar system model. The project demonstrates how Three.js simplifies the creation of interactive and visually accurate 3D models, requiring minimal prior experience in computer graphics. By utilizing the library's intuitive APIs, features such as planetary rotations, orbital dynamics, and textured surfaces were implemented to replicate the solar system's mechanics. The results underscore the library's accessibility and versatility, making it an ideal tool for educational and scientific visualization. Additionally, this study highlights the potential of Three.js to bridge the gap between technical complexity and creative expression in 3D modeling.

Keywords

JavaScript, webGL, 3D Rendering, Three.js, Scientific Modeling

ACM Reference Format:
Julian A. Soto Perez. 2024. Original Proposal: Utilizing webGL and Three.js for 3D Modeling . In . ACM, New York, NY, USA, 4 pages. https://doi.org/NULLXXXXXXX.XXXXXXX

1 Introduction

The visualization of complex systems, such as the solar system, has long been a compelling way to communicate scientific concepts and engage diverse audiences. Traditional tools for 3D modeling and rendering often require significant expertise, which can act as a barrier for educators, students, and developers seeking to create realistic representations of real-world objects and systems. In recent years, browser-based technologies have advanced significantly, with libraries like Three.js making 3D graphics more accessible and interactive.

Three.js is a JavaScript library that abstracts the complexities of WebGL, enabling developers to create intricate 3D scenes with relatively simple code. Its rich ecosystem of features, including predefined geometries, materials, lighting, and animation utilities, offers an approachable entry point for creating dynamic and realistic visualizations. This paper explores the capabilities of Three.js through the development of a solar system model, showcasing its ease of use and the quality of output it can achieve.

The solar system render serves as a case study to illustrate how Three.js can be utilized for educational and scientific purposes. The model incorporates realistic planetary textures, orbital paths, and lighting effects to simulate the solar system's dynamics. By examining the development process, this paper aims to highlight the potential of Three.js as a tool for creating accessible and engaging 3D representations of real-life systems. The findings suggest that Three.js can significantly lower the barrier to entry for individuals interested in 3D modeling, opening new opportunities for innovation in various fields.

2 Methodology

This methodology section outlines the necessary steps and tools for creating a Three.js-based solar system rendering project. It begins with the required hardware and software setup, followed by a step-by-step guide to project initialization, including file organization, dependency management, and scene configuration. Additionally, specific portions of the source code are analyzed to demonstrate the simplicity and efficiency of using Three.js for realistic 3D modeling.

2.1 Necessary Equipment

To implement the solar system model using Three.js, a modern computer with sufficient computational capabilities is essential for smooth performance. The development environment for this project was set up on a 2022 MacBook Air with an M2 chip (8-core) and 16 GB of RAM. This hardware provided adequate processing power for rendering complex 3D scenes and handling interactive animations without noticeable performance issues.

The Three.js library requires a browser that supports WebGL, such as Google Chrome, Mozilla Firefox, or Microsoft Edge. Additionally, to facilitate a streamlined development workflow, a local server environment was established using Vite, a modern build tool optimized for frontend development. Vite enables rapid prototyping by providing hot module replacement (HMR) and a lightweight server for serving files locally.

The software setup for this project included:

- `Node.js` : Installed to manage dependencies and run the Vite server.
- `Vite` : Used to serve the development environment locally.
- `Three.js`: Installed via npm for modular and efficient use within the project.
- `Code Editor`: Visual Studio Code was utilized for writing and managing the project's source code.

This setup ensures compatibility, performance, and ease of development, enabling efficient creation and testing of 3D visualizations in real time.

2.2 Project Setup

To create a basic Three.js rendering project, the following steps were followed:

(1) Create a Project Directory:
   - Begin by setting up a folder to organize the project files, including the HTML structure, JavaScript scripts, and any additional assets like textures.

(2) Initialize the HTML File:
   - Design an HTML file to act as the foundation of the project. This file establishes the structure of the webpage and specifies where the Three.js scene will be rendered, such as through a dedicated canvas element.

(3) Set Up the JavaScript File:
   - Develop a primary JavaScript file to serve as the entry point for the Three.js code. This script is responsible for initializing the scene, camera, and renderer, and for defining the behavior of the 3D objects within the scene.

(4) Install Dependencies:
   - Use a package manager to install the necessary software libraries and tools, including Three.js and a local development server utility.

(5) Configure the Development Server:
   - Set up a local server for running and testing the project. This server enables real-time updates to the code and facilitates debugging during development.

(6) Write the code to configure the Three.js environment, which includes:
   - Setting up a rendering engine to display the 3D graphics.
   - Configuring a camera to view the scene from a specific perspective.
   - Creating and positioning 3D objects such as spheres for planets.
   - Applying textures and materials to enhance realism.
   - Adding light sources to simulate natural lighting effects.

(7) Test and Iterate:
   - Launch the project using the local development server to test functionality. Adjust and refine the code to achieve the desired level of realism and interactivity in the 3D model.

2.3 Code Implementation

The following section highlights portions of the source code and explains their functionality in the context of the solar system render:

2.3.1 ***HTML file setup example:*** An HTML file to act as the foundation of the project. This file establishes the structure of the webpage and specifies where the Three.js scene will be rendered, such as through a dedicated canvas element.

```
/////////////////////////////////////////////////////

<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Solar System</title>

  </head>
  <body>
    <div id="app"></div>

    <canvas id="workspace"></canvas>

    <script type="module" src="/src/main.js"></script>

  </body>
</html>

/////////////////////////////////////////////////////
```

2.3.2 ***Scene, Camera, and Renderer Setup:*** The scene, camera, and renderer are foundational elements of any Three.js project. In this implementation:

- A scene was created to act as a container for all 3D objects
- A perspective camera was configured to provide a realistic viewing angle, with parameters optimized for the solar system's scale and proportions.
- A WebGL renderer was initialized and linked to a canvas element in the HTML file to enable rendering of the 3D graphics.

Portion of script:

```
/////////////////////////////////////////////////////

const scene = new THREE.Scene();

const camera = new THREE.PerspectiveCamera(90,
window.innerWidth / window.innerHeight, 0.1, 3000);

const renderer = new THREE.WebGLRenderer({
canvas: document.querySelector('#workspace'),});

renderer.setPixelRatio(window.devicePixelRatio);
renderer.setSize(window.innerWidth,
window.innerHeight);
camera.position.setZ(200);
camera.position.setX(0);

renderer.render(scene, camera);

/////////////////////////////////////////////////////
```

2.3.3 ***Adding Objects with Built-In Meshes and Materials:*** Three.js simplifies the process of adding 3D objects to the scene by providing predefined meshes and materials. Unlike base OpenGL or WebGL, which require manually defining vertex and fragment shaders, Three.js abstracts these complexities. For instance:

- A sphere geometry can be created using the built-in SphereGeometry class to represent planets.
- Materials, such as MeshStandardMaterial, allow for realistic texturing and lighting effects without the need to write custom shaders.

- The object can then be added to the scene with a single function call, streamlining the workflow for developers.

Portion of script:

```
/////////////////////////////////////////////////////

const sunGeometry = new THREE.SphereGeometry(47, 32, 32);
const sunMaterial = new THREE.MeshBasicMaterial({
color: 0xffff00, emissive: 0xffff00 });
const sun = new THREE.Mesh(sunGeometry, sunMaterial);
sun.position.set(0,0,0);

scene.add(sun);

/////////////////////////////////////////////////////
```

2.3.4 *Combining Three.js Classes and Methods to Achieve Realism:*

To create a realistic 3D environment, Three.js provides tools for incorporating lighting, advanced material types, and movement. For instance:

- **Lighting and Materials:** Adding light objects, such as PointLight or DirectionalLight, can simulate natural illumination, creating depth and enhancing the scene's realism. Materials like MeshPhysicalMaterial or MeshLambertMaterial interact with these light sources to produce realistic shading and reflections.
- **Movement and Rotation:** Built-in functions, such as object rotation and position updates, facilitate the simulation of dynamic movements. For example, rotation can be applied to planetary objects to mimic their axial spin, while position updates can replicate orbital motion around a central star.

Portion of script:

```
//Earth

const earthOrbitGroup = new THREE.Group();
scene.add(earthOrbitGroup);

const textureLoader = new THREE.TextureLoader();

const earthTexture = textureLoader.load(
'./src/earth_dif.jpg');
const earthDepth = textureLoader.load(
'./src/earth_depth.jpg');
const earthSpecular = textureLoader.load(
'./src/earth_spec.jpg');

const earthGeometry = new THREE.SphereGeometry(
0.4084, 32, 32);

const earthMaterial = new THREE.MeshPhongMaterial({
map: earthTexture,
bumpMap: earthDepth,specular: earthSpecular});

const earth = new THREE.Mesh(
earthGeometry, earthMaterial);

earth.position.set(147, 0, 0); // Set the
// planet's initial position

earthOrbitGroup.add(earth);

//MOON

// Create the Moon's orbit group
const moonOrbitGroup = new THREE.Group();

earthOrbitGroup.add(moonOrbitGroup); // Attach the
//Moon's orbit group to Earth's orbit group

// Create the Moon
// Smaller sphere for the Moon
const moonGeometry = new THREE.SphereGeometry(0.1,
32, 32);


const moonMaterial = new THREE.MeshStandardMaterial
({ color: 0xffffff });
const moon = new THREE.Mesh(moonGeometry,
moonMaterial);

// Position the Moon relative to Earth
// Set the Moon's initial position
moon.position.set(148, 1, 0);
moonOrbitGroup.add(moon);

// Attach the Moon's orbit group to Earth's
earthOrbitGroup.add(moonOrbitGroup);

//adding sun light to the scene

//bright white light
const sunLight = new THREE.PointLight(0xffffff,
5, 20);

// Position at the center of the Sun
sunLight.position.set(145, 0, 0);

// Enable shadow casting if needed
sunLight.castShadow = false;

// Add the light to the scene
scene.add(sunLight);

earthOrbitGroup.add(sunLight);
```

3 Renders Preview:

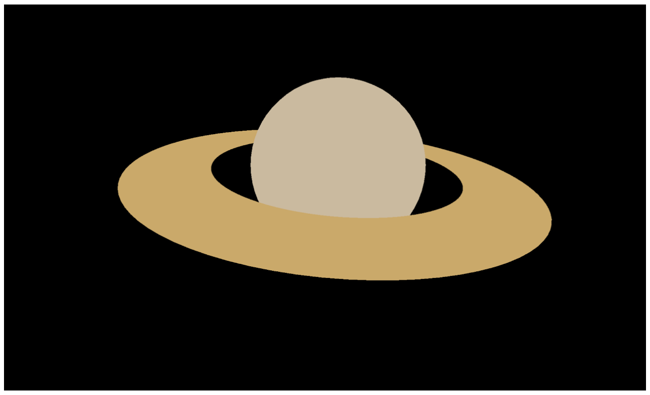

Figure 1: Basic Planet Example

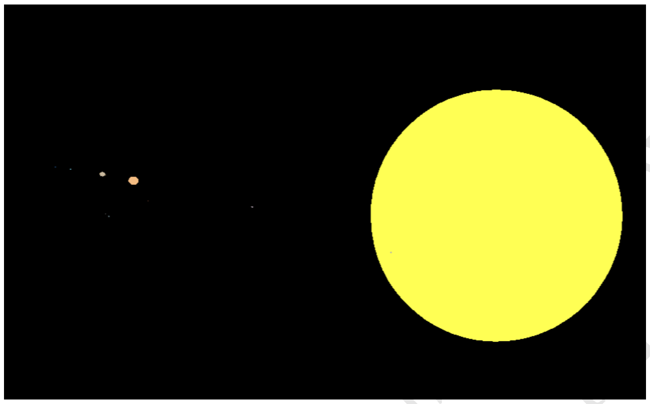

Figure 2: Basic Solar System Model Scaled

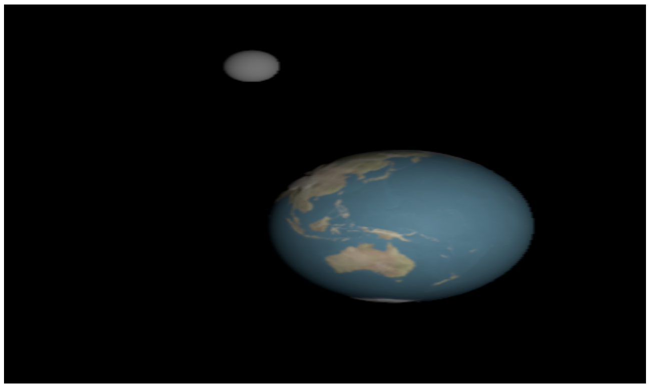

Figure 3: Planet containing additional lighting and texture effects

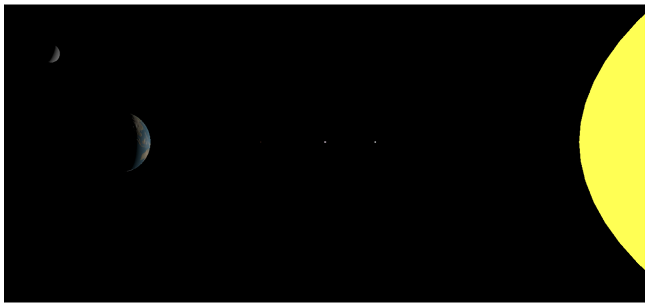

Figure 4: Solar System With Realistic Lighting

4 Conclusion

This paper has explored the capabilities and ease of use of Three.js for 3D modeling and rendering through the development of a solar system visualization. By leveraging its intuitive APIs and built-in functionalities, Three.js provides an accessible pathway for creating highly realistic and interactive models, even for developers with minimal prior experience in 3D graphics programming.

The project demonstrated how essential elements such as geometries, materials, and lighting can be combined to achieve visual fidelity, while built-in animation and transformation tools facilitate dynamic interactions. The solar system render served as a compelling example of how Three.js simplifies complex rendering processes, such as implementing realistic lighting effects, incorporating textures, and animating movements, which traditionally require advanced knowledge in low-level graphics libraries like OpenGL or WebGL.

Through this work, it is evident that Three.js lowers the barrier for creating sophisticated visual representations, making it an ideal tool for educational, scientific, and creative projects. Its versatility and robust feature set enable developers to focus on creative exploration and problem-solving without being hindered by the technical complexities of 3D graphics programming.

Future work could expand upon this foundational model by integrating additional interactive features, such as user-driven exploration or real-time data overlays, to further enhance educational and visualization experiences.

5 References:

`Source code for renders available at:`
`https://github.com/JSP1130/solar_system`

Received 12 December 2024