Addis Ababa University

Colleague of Technology and built Environment

School of Information Technology and Engineering

Section 3 & 4

(Group 9)

Graphics

Documentation

Day/Night Cycle Earth Simulation: The Animation of Earth’s rotation and the day-night light transition in real time.

Alpha Degago(UGR/4592/15)

Ephraim Debel(UGR/0640/15)

Ruth Ambaw (UGR/8802/15)

Saron Tadesse(UGR/5471/15)

Yonas Tessema(ATR/0419/14)

Dr. Addisalem Genta

June, 2025

Table of Content

Project Overview

The **Day/Night Earth Simulation** is an interactive 3D visualization of the Earth, simulating the day and night cycle, atmospheric clouds, and celestial objects such as the Sun, Moon, and a satellite. The simulation leverages the Three.js library for real-time rendering and provides user interaction via mouse controls and object selection. It is a sophisticated web-based 3D visualization project designed to provide users with an immersive and interactive representation of the Earth and its surrounding celestial environment. The primary purpose of this project is to simulate the dynamic interplay between day and night on Earth, while also showcasing additional astronomical elements such as the Sun, Moon, and an orbiting satellite. By leveraging modern web technologies, the simulation offers a visually compelling and educational experience directly within the browser, requiring no additional software installations.

At the core of the simulation are several high-quality assets, including detailed 3D models and realistic textures. The project utilizes a GLTF-format satellite model (`Satellite.glb`) and a suite of texture images (`earth_clouds.jpg`, `earth_day.jpg`, `earth_night.jpg`, and `sun.jpg`) to accurately render the Earth's surface, atmospheric clouds, and the Sun. These assets are seamlessly integrated into the 3D scene, enhancing the realism and depth of the visualization. The careful selection and application of these resources ensure that users are presented with a scene that is both scientifically informative and visually engaging.

The main features of the simulation include a rotating Earth with distinct day and night regions, a semi-transparent cloud layer, a textured Sun, an orbiting Moon, and a satellite that traverses the planet's orbit. Interactive controls allow users to manipulate the camera view, zoom in and out, and interact with objects within the scene. For example, hovering over the satellite highlights it, and clicking on the Moon toggles its orbital motion. These features collectively create a dynamic environment that responds intuitively to user input, fostering exploration and engagement.

Technologically, the project is built upon the robust Three.js library, which provides the foundation for real-time 3D rendering in the browser via WebGL. The structure of the project is organized around three principal files: index.html (the main HTML entry point), `main.js` (the JavaScript file containing the simulation logic), and `style.css` (the stylesheet for visual presentation). The use of Three.js extensions such as OrbitControls and GLTFLoader further enhances the interactivity and asset management capabilities of the application.

Potential use cases for the Day/Night Earth Simulation are diverse, ranging from educational demonstrations in classrooms to interactive exhibits in museums or science centers. The simulation can serve as a valuable tool for teaching concepts related to planetary motion, the Earth's rotation, and the relationship between celestial bodies. Additionally, it can be adapted for public outreach, presentations, or as a visually appealing component of a larger web-based project focused on astronomy or Earth sciences.

From a user experience perspective, the simulation is designed to be intuitive and accessible. The responsive canvas ensures compatibility with various screen sizes, while the interactive controls provide a smooth and engaging way to explore the 3D environment. The integration of real-time feedback, such as object highlighting and orbit toggling, encourages users to interact with the scene and discover its features organically.

The integration of assets is handled efficiently within the JavaScript logic, with textures and models loaded asynchronously to ensure optimal performance and minimal loading times. This approach allows for a seamless transition between different visual states and supports the addition of new assets or features with minimal modification to the existing codebase. Overall, the project delivers significant educational and entertainment value by combining scientific accuracy, interactive design, and high-quality visual assets in a single, browser-based application.

Directory Structure

```
├── assets
│   ├── models
│   │   └── Satellite.glb
│   └── textures
│       ├── earth_clouds.jpg
│       ├── earth_day.jpg
│       ├── earth_night.jpg
│       └── sun.jpg
├── index.html
└── styles
    └── style.css
```

Technical Stack

Three.js:

Three.js serves as the foundational JavaScript library for this project, providing a comprehensive framework for creating, rendering, and managing interactive 3D graphics within the browser. By abstracting the complexities of WebGL, Three.js enables the development of sophisticated visualizations, such as the Earth's day and night cycle, atmospheric effects, and animated celestial objects, with a streamlined and accessible API.

WebGL:

WebGL is the underlying graphics technology leveraged by Three.js to deliver hardware-accelerated 3D rendering directly in the web browser. It allows the simulation to display complex scenes and real-time animations efficiently, without requiring any additional plugins or installations, ensuring broad compatibility and high performance across modern devices.

GLTFLoader:

GLTFLoader, an extension of Three.js, is utilized to import and display 3D models in the GLTF and GLB formats. In this project, it is specifically responsible for loading the satellite model, enabling seamless integration of detailed external assets into the scene and supporting efficient parsing and rendering of modern 3D content.

OrbitControls:

OrbitControls is a Three.js add-on that facilitates intuitive camera manipulation through mouse or touch input. It allows users to orbit, zoom, and pan around the 3D environment, greatly enhancing interactivity and user engagement by providing direct control over the viewpoint within the simulation.

HTML5:

HTML5 provides the structural markup for the application, including the canvas element that acts as the rendering target for the 3D scene. It ensures a semantic and accessible foundation for the simulation, supporting integration with modern web standards and responsive design practices.

JavaScript (ES6+):

JavaScript (ES6+) is used to implement the simulation logic, handle asset loading, manage user interactions, and drive the animation routines that bring the visualization to life. Its modular and modern syntax supports maintainable and scalable code throughout the project.

CSS3:

CSS3 is employed to style the application, ensuring that the simulation canvas and user interface are visually appealing and responsive. It contributes to a seamless user experience by adapting the layout to different screen sizes and devices.

Image Assets (JPEG/PNG):

Image assets in JPEG and PNG formats are used as high-resolution textures for the Earth, its atmosphere, and the Sun. These textures enhance the realism of the simulation by providing detailed surface features, cloud layers, and solar imagery, contributing to the overall visual fidelity of the project.

GLTF 2.0 Model:

The GLTF 2.0 model format, specifically the Satellite.glb file, provides a compact and efficient means of representing the satellite as a detailed 3D asset. This format is optimized for web delivery and real-time rendering, allowing the satellite to be animated smoothly as it orbits the Earth within the simulation.

Core Features

3D Earth Model

The simulation presents a highly realistic 3D Earth, constructed using a high-resolution sphere geometry and enhanced with multiple texture maps. The day texture (`earth_day.jpg`) and night texture (`earth_night.jpg`) from the textures directory are mapped onto the sphere to depict the Earth's surface under sunlight and city lights during nighttime, respectively. Bump mapping is applied to add surface relief, and the night texture is used as an emissive map to simulate illuminated cities. The Earth mesh is created and animated in both index.html and main.js, where it is continuously rotated to mimic the planet's natural rotation, providing a dynamic and visually compelling centerpiece for the simulation.

Cloud Layer

A semi-transparent, rotating cloud layer is rendered above the Earth to simulate atmospheric movement. This effect is achieved by creating a slightly larger sphere geometry and applying the `earth_clouds.jpg` or `earth_clouds.png` texture as a transparent material. The cloud mesh rotates independently and at a slightly different speed from the Earth, creating the illusion of dynamic weather systems. The implementation of this feature is found in both index.html and main.js, where the cloud geometry and material are defined and added to the scene.

Sun and Lighting

The simulation incorporates advanced lighting to replicate sunlight and enhance realism. Directional lighting is used to simulate the Sun's rays, casting highlights and shadows across the Earth's surface, while a point light is positioned at the Sun's location to provide additional illumination. The Sun itself is represented as a textured sphere using the `sun.jpg` asset, contributing to the authenticity of the scene. These lighting effects and the Sun mesh are implemented in the script section of index.html, where the light sources and Sun geometry are created and positioned relative to the Earth.

Moon

A 3D Moon is included in the simulation, orbiting the Earth in real time. The Moon is modeled as a smaller sphere and is animated to revolve around the Earth's position. Interactive functionality allows users to toggle the Moon's orbit by clicking on it, pausing or resuming its movement. This interactivity is handled by event listeners and animation logic in index.html, adding both visual interest and user engagement to the simulation.

Satellite

The project features a detailed satellite model, loaded from the `Satellite.glb` file in the models directory using Three.js's `GLTFLoader`. The satellite is animated to orbit the Earth, with its position and rotation updated in each animation frame. Interactive highlighting is implemented using raycasting: when users hover the mouse over the satellite, its color changes to indicate selection. This functionality is present in the main script of index.html, providing a responsive and engaging user experience.

Starfield

To enhance the sense of immersion, a procedurally generated starfield is rendered as the background. Thousands of points are randomly distributed in 3D space to simulate distant stars, using Three.js's `BufferGeometry` and `PointsMaterial`. The starfield is created and added to the scene during initialization in index.html, providing a visually rich and realistic cosmic backdrop for the simulation.

User Interaction

User interaction is a core aspect of the project, with mouse-based camera controls enabled via Three.js's `OrbitControls`. Users can orbit, zoom, and pan the camera to explore the scene from any angle. Additional interactions, such as object selection and real-time feedback (e.g., satellite highlighting and Moon orbit toggling), are implemented using event listeners and raycasting. These features are present in both the inline script of index.html and the modular code in main.js, ensuring a highly interactive and intuitive user experience.

Responsive Design

The simulation is designed to be fully responsive, with the rendering canvas automatically resizing to fit the browser window. Event listeners detect window resize events and update the camera's aspect ratio and renderer's size accordingly. This ensures that the visualization remains visually consistent and accessible across a wide range of devices and screen sizes, as implemented in both index.html and main.js.

File Descriptions

index.html

The index.html file is the main entry point for the Day/Night Earth Simulation project. It defines the web page structure, loads all required libraries, and contains the primary script that initializes and runs the interactive 3D simulation. Below is a detailed breakdown of its main sections and their roles within the project.

1. Document Declaration and `<html>` Element

```
<!DOCTYPE html>

<html lang="en">
```

Declares the document as HTML5 and sets the language to English. This ensures proper rendering and accessibility for users and search engines.

2. <head> Section

```
<head>

 <meta charset="UTF-8" />

 <title>Day/Night Earth Simulation</title>

 <meta name="viewport" content="width=device-width, initial-scale=1.0"
/>
```

```html
<link rel="stylesheet" href="styles/style.css">
 <link rel="icon" href="data:,">
</head>
```

Purpose and Function:

- **Meta Tags:**
  - `charset="UTF-8"` ensures proper character encoding.
  - `viewport` enables responsive design for different devices.
- **Title:**
  - Sets the browser tab title.
- **Link to css**
- **Favicon:**
  - Uses a data URI to avoid browser favicon errors.

Integration:

- The inline CSS provides basic styling. For more advanced or custom styles, the project can use style.css, though this is not explicitly linked here.
- The `<head>` section ensures the page is visually optimized for the simulation and ready for JavaScript-driven rendering.

3. <body> Section and Canvas

```html
<body>
 <canvas id="webglCanvas"></canvas>
      . . .
</body>
```

Purpose and Function:

- Contains a single `<canvas>` element with the ID `webglCanvas`.
- This canvas is the rendering target for the Three.js 3D scene.

Integration:

- The JavaScript code (either inline or in main.js) selects this canvas and renders all 3D graphics to it.
- The canvas is styled to fill the window, providing a seamless user experience.

4. Script Imports (Three.js and Extensions)

```
<!-- THREE.js core -->
 <script
src="https://cdn.jsdelivr.net/npm/three@0.132.2/build/three.min.js"></s
cript>
 <!-- OrbitControls -->
 <script
src="https://cdn.jsdelivr.net/npm/three@0.132.2/examples/js/controls/Or
bitControls.js"></script>
 <!-- GLTFLoader -->
 <script
src="https://cdn.jsdelivr.net/npm/three@0.132.2/examples/js/loaders/GLT
FLoader.js"></script>
```

Purpose and Function:

- Loads the Three.js core library and two essential extensions (OrbitControls and GLTFLoader) from a CDN.
- These libraries provide the 3D engine, camera controls, and model loading capabilities required for the simulation.

Integration:

- The main simulation script relies on these libraries to create and manage the 3D scene, handle user interaction, and load external 3D models (e.g., `Satellite.glb`).

5. Main Simulation Script

```
 <script> . . .  </script>
```

Purpose and Function:

- Contains the primary JavaScript logic for the simulation.
- Initializes the Three.js scene, camera, renderer, controls, and loads all required assets (textures and models).
- Creates and configures all 3D objects: Earth, clouds, Sun, Moon, satellite, and starfield.
- Sets up lighting, user interactions (mouse hover, click), and the animation loop.
- Handles window resizing to maintain responsiveness.

Integration:

- Directly manipulates the `<canvas id="webglCanvas">` element.
- Loads assets from the textures and models directories.
- Uses the loaded Three.js libraries and extensions.

- Can be modularized by moving this script to main.js for better maintainability.

Script Code

This file implements the core logic for initializing, rendering, and animating the 3D Earth simulation using Three.js. It is designed to be modular and maintainable, providing a clear separation of concerns for scene setup, asset loading, user interaction, and animation.

1. Module Imports and Scene Initialization

```
<script src="https://cdn.jsdelivr.net/npm/three@0.132.2/build/three.min.js"></script>

 <!-- OrbitControls -->

 <script src="https://cdn.jsdelivr.net/npm/three@0.132.2/examples/js/controls/OrbitControls.js"></script>

 <!-- GLTFLoader -->

 <script src="https://cdn.jsdelivr.net/npm/three@0.132.2/examples/js/loaders/GLTFLoader.js"></script>



const canvas = document.querySelector('#webglCanvas');

const scene = new THREE.Scene();
```

Explanation:

- Importing the Modules

```
<script src="https://cdn.jsdelivr.net/npm/three@0.132.2/build/three.min.js"></script>

 <!-- OrbitControls -->

 <script src="https://cdn.jsdelivr.net/npm/three@0.132.2/examples/js/controls/OrbitControls.js"></script>

 <!-- GLTFLoader -->

 <script src="https://cdn.jsdelivr.net/npm/three@0.132.2/examples/js/loaders/GLTFLoader.js"></script>
```

```
```

  - **THREE**: This import statement brings in the **entire Three.js core library** as the namespace `THREE`. It includes all foundational classes required for 3D rendering, such as geometries, materials, textures, cameras, and more.
  - **OrbitControls**: This is an ES6 module import of the **OrbitControls** utility from the examples directory of Three.js. It's an optional add-on that allows interactive camera movement (pan, zoom, rotate) via mouse or touch inputs.
- Scene and Canvas Initialization

```
const canvas = document.querySelector('#webglCanvas');

const scene = new THREE.Scene();
```

  - **canvas**: This selects the HTML `<canvas>` element with the id `webglCanvas`. This element serves as the **render target**, a low-level surface where WebGL will draw frames.
  - **scene**: Instantiates a new `THREE.Scene()` object. The scene acts as the **root container** for the 3D scene graph. All rendered objects, lights, and effects must be added to this scene for the renderer to process.

Inputs:

- **HTML Document**: Must contain an element such as:

```
 <canvas id="webglCanvas"></canvas>
```

This provides the drawing surface.

- No direct input data required from JavaScript at this point.

Outputs:

- **scene**: A fully initialized 3D scene graph, ready to accept camera, lights, meshes, and other Three.js objects.
- **canvas**: A reference to the HTML canvas element, later used by the renderer to display graphics.

Interactions:

- Relies on the `<canvas id="webglCanvas"></canvas>` element in index.html.
- `scene` will be used in combination with a `THREE.Renderer` (e.g., `THREE.WebGLRenderer`) to produce frames rendered on the `canvas`.
- `OrbitControls` will be associated with a camera and the `canvas` element to enable interactive camera movements.
- The canvas must have a valid WebGL context for rendering.

2. Camera Setup

```
const camera = new THREE.PerspectiveCamera(

 75, window.innerWidth / window.innerHeight, 0.1, 1000
);
camera.position.z = 3;
```

Explanation:

- Creates a perspective camera with
  - Field of View (FOV): The `75` parameter defines the camera's vertical field of view in degrees. This determines how "wide" the camera's frustum appears.
  - Aspect Ratio: `window.innerWidth / window.innerHeight` ensures that the camera's view maintains the correct proportions relative to the browser window dimensions, preventing distortion.
  - Near and Far Clipping Planes: Objects closer than `0.1` units or farther than `1000` units from the camera will be clipped and not rendered. These define the camera's viewing frustum.
- Positioning: By setting `camera.position.z = 3`, the camera is placed 3 units away from the origin along the Z-axis. This ensures it is not coincident with the scene's origin, allowing for a better initial view of the scene.

Inputs:

- Window dimensions: `window.innerWidth` and `window.innerHeight` are used to compute the camera's aspect ratio.
- Numeric Parameters: Field of view (FOV), near clipping, and far clipping planes.

Outputs:

- `camera`: An instance of `THREE.PerspectiveCamera` configured for a realistic, depth-based projection.

Side Effects:

- None (other than allocating an object in memory).
- The camera's position and settings do not affect the scene until rendered.

Interactions:

- The camera is used by the renderer (`THREE.WebGLRenderer`) to determine how the scene is projected onto the `canvas`.
- Will be used in tandem with OrbitControls to enable user interactions like panning, zooming, and rotating.
- Must be kept in sync with the window's dimensions (aspect ratio) for accurate rendering when resizing.

3. Renderer Setup

```
const renderer = new THREE.WebGLRenderer({ canvas });
renderer.setSize(window.innerWidth, window.innerHeight);
renderer.setPixelRatio(window.devicePixelRatio);
```

Explanation:

```
const renderer = new THREE.WebGLRenderer({ canvas });
```

- Instantiates a WebGL renderer configured to draw into an existing HTMLCanvasElement (`canvas`) rather than creating a new one.
- Enables direct access to WebGL rendering context and GPU pipelines for drawing frames.

```
renderer.setSize(window.innerWidth, window.innerHeight);
```

- Defines the internal drawing buffer size of the renderer (in pixels), matching the browser window dimensions.
- Ensures that the renderer maintains a 1:1 correspondence between its drawing surface and the viewport dimensions.

```
renderer.setPixelRatio(window.devicePixelRatio);
```

- Adjusts the renderer's pixel ratio, making it adapt to high-DPI or "Retina" displays.
- Provides crisper, more accurate rendering of lines, textures, and text by accounting for screens where one CSS pixel contains multiple device pixels.

Inputs:

- canvas: The target HTML canvas element.
- window.innerWidth / window.innerHeight: The current viewport dimensions.
- window.devicePixelRatio: The device's pixel density, used to adjust rendering resolution.

Outputs:

- `renderer`: A fully configured `THREE.WebGLRenderer` instance, ready to draw 3D scenes to the target canvas.

Side Effects:

- Modifies the canvas element's internal rendering context and its dimensions.
- Updates GPU state and the WebGL drawing buffer based on screen and device characteristics.

Interactions:

- Works in tandem with the `camera` to render frames of the `scene`.
- Will be called repeatedly in the animation loop (`renderer.render(scene, camera)`) to draw updated frames.
- Must be kept in sync with screen resizing events, making it necessary to call `setSize()` and adjust camera `aspect` accordingly when the window resizes.

4. Orbit Controls

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

Explanation:

- This Constructs an `OrbitControls` instance, binding it to the camera and the renderer's DOM element (the canvas). And enables an interactive camera system, allowing the user to rotate, pan, and zoom the camera around a target point.
- Its underlying mechanism is that it Listens for user input events (`mousedown`, `mousemove`, `wheel`, `touchstart`, `touchmove`, etc.). And translates these inputs into spherical coordinate adjustments for the camera position, updating its position and making it orbit around its target.

Inputs:

- `camera`:An instance of `THREE.PerspectiveCamera` that `OrbitControls` will manipulate.
- `renderer.domElement`: The canvas where user interactions occur (and where the controls attach event listeners).

Outputs:

- `controls`: An instance of `OrbitControls` that:
  - Exposes APIs for controlling camera behavior (e.g., `controls.target`, `controls.update()`).
  - Enables interactive navigation of the 3D scene.

Side Effects:

- Listens for and handles user events (`mousemove`, `wheel`, `touchmove`) on the `canvas`.
- Maintains internal state representing camera angles, target positions, and zoom distances.
- Modifies the camera's position and orientation every frame when `controls.update()` is called.

Interactions:

- Works in tandem with:
  - Camera: Changes its position and orientation.
  - Renderer: Enables user interaction within the rendered scene.
  - Animation Loop: Requires calling `controls.update()` every frame for smooth interactions.

5. Texture Loading

```
const loader = new THREE.TextureLoader();
const earthDayMap = loader.load('assets/textures/earth_day.jpg');
const earthNightMap = loader.load('assets/textures/earth_night.jpg');

const bumpMap = null;

const cloudMap = loader.load('assets/textures/earth_clouds.png');
```

Explanation:

- `new THREE.TextureLoader()`:
  - Instantiates a loader for 2D image files that can be used as textures in three.js.
  - Enables asynchronous downloading and GPU uploading of texture data.
- Earth Textures:
  - `earthDayMap`: A diffuse (albedo) map representing the planet surface as seen in daylight.
  - `earthNightMap`: A second map representing nighttime lighting, typically combined via shader or lighting logic.
  - `bumpMap`: Currently set to `null`, indicating that no bump or displacement map is used.
  - `cloudMap`: A semi-transparent texture representing atmospheric cloud patterns.

Inputs:

- File paths to texture assets.
  - `'assets/textures/earth_day.jpg'`
  - `'assets/textures/earth_night.jpg'`
  - `'assets/textures/earth_clouds.png'`

Outputs:

- `earthDayMap`, `earthNightMap`, `cloudMap`: Texture objects.
  - Fully configured instances of `THREE.Texture`.
  - Will be uploaded to the GPU once asynchronously loaded.

Side Effects:

- Initiates asynchronous HTTP requests for each texture.
- Upon load, `THREE.TextureLoader`:
  - Processes the images.
  - Uploads them to GPU memory.
- Will trigger a re-render if the scene uses these textures when available.

Interactions:

- **Materials**:
  - Will be used as inputs for `THREE.MeshStandardMaterial`, `THREE.MeshLambertMaterial`, or custom shader materials.
- **Geometry**:

    - Will be wrapped with these textures to produce realistic planetary surfaces and atmospheric effects.
- **Rendering**:
    - Will require consideration of UV mapping and lighting effects for best results.

6. Earth Geometry and Material

```
const earthGeometry = new THREE.SphereGeometry(1, 64, 64);

const earthMaterial = new THREE.MeshPhongMaterial({

 map: earthDayMap,

 bumpMap: bumpMap,

 bumpScale: 0.05,

 specular: new THREE.Color('grey'),

 shininess: 5,

});


// Night texture as emissive map (fake city lights)

earthMaterial.emissiveMap = earthNightMap;

earthMaterial.emissive = new THREE.Color(0xffffff);

earthMaterial.emissiveIntensity = 0.5;


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

scene.add(earthMesh);
```

Explanation:

```
const earthGeometry = new THREE.SphereGeometry(1, 64, 64);
```

- Constructs a parametric sphere mesh with:
    - Radius: 1 unit.
    - Width Segments: 64 — higher segment count creates a smooth surface ideal for planetary rendering.
    - Height Segments: 64 — similarly ensures smooth vertical tessellation.

Provides a high-resolution mesh yielding visually rounded geometry for lighting and texturing.

```
const earthMaterial = new THREE.MeshPhongMaterial({

 map: earthDayMap,

 bumpMap: bumpMap,

 bumpScale: 0.05,

 specular: new THREE.Color('grey'),

 shininess: 5,

});
```

- Base Texture (map): The earthDayMap provides the diffuse (day) surface coloring.
- Bump Mapping (bumpMap / bumpScale):
  - Enables surface micro-relief by displacing normals.
  - bumpScale: 0.05 gives a subtle terrain effect.
  - bumpMap is null in this context, so this has no effect until replaced with an actual texture.
- Specular Highlights:
  - specular: new THREE.Color('grey'): Defines the color of specular reflections.
  - shininess: 5: Provides a low specular sharpness for a matte planetary surface.

```
// Night texture as emissive map (fake city lights)

earthMaterial.emissiveMap = earthNightMap;

earthMaterial.emissive = new THREE.Color(0xffffff);

earthMaterial.emissiveIntensity = 0.5;
```

- Emissive Map (`emissiveMap`):
  - Enables self-illumination of the surface with the `earthNightMap`.
  - Simulates city lights that “glow” in low-light or shadowed areas.
- Emissive Color (`emissive`):
  - Defines the tint of the emissive map (set to neutral `white` for faithful color reproduction).
- Emissive Intensity (`emissiveIntensity`):
  - Determines how strong the emissive effect appears (0.5 = moderate brightness).

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

scene.add(earthMesh);
```

- Combines geometry and material into a **mesh**.
- Adds the mesh to the scene graph, making it part of the render pipeline.

Inputs:

- Geometry Parameters: Radius and segment counts.
- Material Textures: `earthDayMap`, `earthNightMap`, `bumpMap`.
- Material Properties: Specular color, shininess, emissive settings.

Outputs:

- `earthMesh`:
    - A renderable 3D object representing the Earth.
    - Fully configured for realistic day-night lighting.

Side Effects:

- The Earth mesh is added to the scene, making it subject to scene lighting, camera settings, and controls.

Interactions:

- Works in tandem with:
    - **Lights**: To illuminate the surface and affect the emissive map.
    - **Camera**: To position the sphere within the view frustum.
    - **Renderer**: To draw and shade the mesh each frame.
- Enables complex shading dynamics when combined with other maps (normal, specular, displacement).

7. Cloud Layer

```
const cloudGeometry = new THREE.SphereGeometry(1.01, 64, 64);

const cloudMaterial = new THREE.MeshLambertMaterial({

 map: cloudMap,

 transparent: true,

 opacity: 0.4,

});

const cloudMesh = new THREE.Mesh(cloudGeometry, cloudMaterial);

scene.add(cloudMesh);
```

Explanation:

- `const cloudGeometry = new THREE.SphereGeometry(1.01, 64, 64);`
- Constructs a slightly larger sphere with a **radius of 1.01** units (just slightly larger than the Earth's `1.0`).
- The higher segment counts (64 x 64) ensure a smooth silhouette and seamless texturing, making the cloud layer appear realistic.
- `const cloudMaterial = new THREE.MeshLambertMaterial({`
- ` map: cloudMap,`
- ` transparent: true,`
- ` opacity: 0.4,`

```
});
```

- Base Material: `MeshLambertMaterial` — ideal for semi-transparent, non-shiny surfaces that respond naturally to scene lighting.
- Texture (`map`):
  - Uses the `cloudMap` texture, providing a visual overlay of cloud patterns.
- Transparency:
  - Enables the `transparent` property, allowing partial visibility of the surface.
  - `opacity: 0.4` creates a subtle, translucent effect, making the cloud layer appear ethereal.

```
const cloudMesh = new THREE.Mesh(cloudGeometry, cloudMaterial);

scene.add(cloudMesh);
```

- Combines the geometry and material into a `THREE.Mesh`.
- Adds it to the scene, making it renderable and positionable relative to the Earth.

Inputs:

- Geometry Parameters: `radius = 1.01`, `widthSegments = 64`, `heightSegments = 64`.
- Material Properties:
  - `map`: `cloudMap` (cloud texture).
  - `transparent`: Enables blending of the cloud surface.
  - `opacity`: Determines cloud visibility and softness.

Outputs:

- `cloudMesh`: A semi-transparent mesh representing the atmospheric cloud layer.

Side Effects:

- The mesh is added to the scene, making it subject to lighting and camera interactions.
- Enables depth-based rendering due to its placement slightly offset from the surface mesh.

Interactions:

- Works closely with the Earth Mesh:
  - Provides a realistic atmospheric overlay.
- Reacts to the scene's lighting:
  - Will be shaded appropriately by any scene lights (especially relevant when using `MeshLambertMaterial`).
- Influences the scene's visual complexity:
  - The combined rendering of the Earth surface and cloud layer creates a visually rich planetary body.

8. Lighting Setup

```
const ambientLight = new THREE.AmbientLight(0x333333);
```

```
scene.add(ambientLight);


const directionalLight = new THREE.DirectionalLight(0xffffff, 1);

directionalLight.position.set(5, 3, 5);

scene.add(directionalLight);
```

Explanation:

- `const ambientLight = new THREE.AmbientLight(0x333333);`
- `scene.add(ambientLight);`
- Provides global illumination — a base level of light that affects every object in the scene equally.
- The color `0x333333` is a subdued grey, yielding a low-intensity ambient fill. This prevents completely black, unlit areas and creates subtle illumination across surfaces.
- `const directionalLight = new THREE.DirectionalLight(0xffffff, 1);`
- `directionalLight.position.set(5, 3, 5);`
- `scene.add(directionalLight);`
- Simulates a distant light source (e.g., the sun), emitting parallel rays across the scene.
- The position (`5, 3, 5`) determines the incident angle of the light, creating realistic highlights, shadows, and surface depth.
- The intensity of 1 provides a strong, crisp light source that mimics natural lighting conditions.

Inputs:

- **Light Properties**:
  - Color (`0x333333` for ambient, `0xffffff` for directional).
  - Position and intensity (for the directional light).

Outputs:

- `ambientLight`: A global light contribution, affecting all surfaces equally.
- `directionalLight`: A directional illumination yielding realistic shading, highlights, and depth across the scene.

Side Effects:

- Influences how the renderer calculates illumination across all meshes (including the Earth surface and cloud layers).
- Enables accurate Lambertian and Phong-style reflections, making materials respond realistically.

Interactions:

- The Earth Mesh and Cloud Mesh:

    - Both respond to the directional light for specular highlights and bump mapping.
    - The ambient light ensures that surfaces aren't completely black in shadowed areas.
- The Material System:
    - Works with `MeshLambertMaterial` (cloud) and `MeshPhongMaterial` (Earth) to produce realistic lighting and surface reflections.
- The Camera:
    - Together with the camera's position and light angles, defines the scene's overall illumination and mood.

9. Animation Loop

```
function animate() {
 requestAnimationFrame(animate);

 // Earth's rotation
 earthMesh.rotation.y += 0.001;
 cloudMesh.rotation.y += 0.0012;

 controls.update();
 renderer.render(scene, camera);
}

animate();
```

Explanation:

- requestAnimationFrame(animate):
    - Enables a browser-synchronized animation loop.
    - Targets a frame rate aligned with the display's refresh rate (typically 60 Hz), ensuring smooth motion and optimized performance.
    - Provides the ability for the browser to throttle or optimize animations when the page is not visible.
- Earth and Clouds Rotation:
    - `earthMesh.rotation.y += 0.001`: Gradual, continuous rotation simulates the planet's spin.
    - `cloudMesh.rotation.y += 0.0012`: Slightly different rotational speed for the cloud layer gives a parallax effect, making the scene more dynamic and realistic.

- controls.update():
  - Processes user input (drag, scroll, pinch) captured by `OrbitControls`.
  - Updates the camera position, making the scene respond to user interactions in real time.
- renderer.render(scene, camera):
  - Finalizes the frame by submitting draw commands to the GPU.
  - Renders the scene from the camera's point of view.

Inputs:

- Internal State:
  - Current rotational angles of `earthMesh` and `cloudMesh`.
  - Camera controls state (user input).
- External Inputs:
  - Mouse or touch events captured by `OrbitControls`.

Outputs:

- A continuous stream of rendered frames, creating the illusion of motion and interactivity.

Side Effects:

- Updates object positions (`rotation.y`), affecting subsequent frames.
- Invokes GPU draw commands every animation frame.

Interactions:

- Enables the scene to respond to user input via `OrbitControls`.
- Maintains smooth synchronization between rendered frames, object movement, and camera controls.
- Provides a dynamic, interactive experience that showcases lighting, texturing, and geometry working together.

10. Responsive Resize Handling

```
window.addEventListener('resize', () => {
 camera.aspect = window.innerWidth / window.innerHeight;
 camera.updateProjectionMatrix();


 renderer.setSize(window.innerWidth, window.innerHeight);
});
```

Explanation:

- Listens for the `resize` event emitted by the `window` object.
- Maintains the correct aspect ratio of the camera when the browser window is resized.
- Updates the camera's projection matrix, ensuring accurate spatial relationships and depth calculations after resizing.
- Adjusts the renderer's output surface (`canvas`) to match the new window dimensions.

Inputs:

- Window size changes:
  - `window.innerWidth`
  - `window.innerHeight`

Outputs:

- Updated camera.aspect and camera.projectionMatrix for properly scaled and framed rendering.
- Updated renderer.size so that the output occupies the available space precisely.

Side Effects:

- Forces a recalculation of the camera's internal matrices (`projectionMatrix`) every time the window resizes.
- Causes GPU draw buffers and WebGL viewport to be reallocated based on the new dimensions.

Interactions:

- Works closely with:
  - Camera: Maintains the camera's frustum ratio for accurate rendering.
  - Renderer: Adjusts the WebGL viewport and output surface.
- Enables seamless responsiveness across varying screen sizes and device orientations.

11. Sun (with texture, emissive, and point light)

```
const sunGeometry = new THREE.SphereGeometry(0.5, 32, 32);

   const sunMaterial = new THREE.MeshStandardMaterial({

     map: sunTexture,

     emissiveMap: sunTexture,

     emissive: new THREE.Color(0xffaa33),

     emissiveIntensity: 1.5

   });

   const sun = new THREE.Mesh(sunGeometry, sunMaterial);
   sun.position.set(8, 0, 0);

   scene.add(sun);


   const sunlight = new THREE.PointLight(0xffee88, 1.5, 100);

   sunlight.position.copy(sun.position);

   scene.add(sunlight);
```

Explanation:

- The sun is represented as a small sphere with a sun texture and strong emissive color, making it appear to glow.
- A point light is placed at the sun's position to illuminate the scene, simulating sunlight.

Inputs:

- Texture file: `assets/textures/sun.jpg`

Outputs:

- A glowing sun mesh and a point light source.

Side Effects:

- Adds a visible sun and realistic lighting to the scene.

Interactions:

- The sun and its light affect the appearance of all objects in the scene.

12. Moon (with orbit and click-to-toggle orbit)

```
const moonGeometry = new THREE.SphereGeometry(0.27, 32, 32);

   const moonMaterial = new THREE.MeshStandardMaterial({ color:
0xaaaaaa });

   const moon = new THREE.Mesh(moonGeometry, moonMaterial);

   moon.name = 'Moon';

   scene.add(moon);

let moonOrbitEnabled = true;

let moonAngle = 0;

if (moonOrbitEnabled) {
       moonAngle += 0.002;
     }
moon.position.set(
       earth.position.x + Math.cos(moonAngle) * 3,
       0,
       earth.position.z + Math.sin(moonAngle) * 3
     );
// Click on moon: toggle orbit
   window.addEventListener('click', (event) => {
      . . .
     if (intersects.length > 0) {
       moonOrbitEnabled = !moonOrbitEnabled;
       console.log(' Moon orbit', moonOrbitEnabled ? 'resumed' :
'paused');
     }
   });
```

Explanation:

- The moon is a smaller sphere that orbits the Earth.
- Its orbit can be paused or resumed by clicking on the moon, using raycasting to detect clicks.

Inputs:

- Mouse click events.

Outputs:

- Animated moon orbit, toggled by user interaction.

Side Effects:

- The moon's position changes dynamically; user can control its motion.

Interactions:

- The moon's orbit is visually linked to the Earth and responds to user input.

13. Satellite (GLTF model loading and orbit)

```
let satelliteModel = null;

   const gltfLoader = new THREE.GLTFLoader();

   gltfLoader.load(

     './assets/models/Satellite.glb',

     (gltf) => {

       satelliteModel = gltf.scene;

       satelliteModel.scale.set(0.03, 0.03, 0.03);

       satelliteModel.name = 'Satellite';

       scene.add(satelliteModel);

     },

     undefined,

     (error) => {

       console.error('Error loading satellite model:', error);

     }

   );

if (satelliteModel) {

       satAngle += 0.005;

       satelliteModel.position.set(

         earth.position.x + Math.sin(satAngle) * 4,

         Math.sin(satAngle) * 1,

         earth.position.z + Math.cos(satAngle) * 4

       );

       satelliteModel.rotation.y += 0.01;

     }
```

Explanation:

- Loads a 3D satellite model using GLTFLoader and adds it to the scene.
- The satellite orbits the Earth and rotates for a dynamic effect.

Inputs:

- Model file: `assets/models/Satellite.glb`

Outputs:

- Animated satellite mesh orbiting the Earth.

Side Effects:

- Adds a moving satellite to the simulation.

Interactions:

- The satellite's position and rotation are updated every frame.

14. Starfield background

```
function addStarField(count = 1000) {
     const starGeometry = new THREE.BufferGeometry();
     const starPositions = [];

     for (let i = 0; i < count; i++) {
       const x = THREE.MathUtils.randFloatSpread(200);
       const y = THREE.MathUtils.randFloatSpread(200);
       const z = THREE.MathUtils.randFloatSpread(200);
       starPositions.push(x, y, z);
     }

     starGeometry.setAttribute('position', new
THREE.Float32BufferAttribute(starPositions, 3));

     const starMaterial = new THREE.PointsMaterial({
       color: 0xffffff,
       size: 0.5,
       sizeAttenuation: true
     });
```

```
    const stars = new THREE.Points(starGeometry, starMaterial);

    scene.add(stars);

  }


  addStarField();
```

Explanation:

- Generates a field of randomly positioned stars using `THREE.Points` for a realistic space background.

Inputs:

- Star count (default 1000).

Outputs:

- Visual starfield in the background.

Side Effects:

- Enhances the realism and depth of the scene.

Interactions:

- The starfield is static and does not interact with other objects.

Satellite.glb

The Satellite.glb file is a 3D model asset in the GLTF 2.0 binary format, located in the models directory of the project. This file contains a detailed representation of a satellite, including its geometry, materials, and potentially embedded textures or animations. The GLB format is optimized for efficient loading and rendering in web applications, making it well-suited for real-time 3D environments like this simulation. By using GLTF/GLB, the project ensures that the satellite model can be easily parsed and displayed by Three.js with minimal overhead.

Within the simulation, Satellite.glb is loaded at runtime using Three.js's GLTFLoader extension, as seen in the main script of index.html. Once loaded, the satellite model is scaled and added to the scene, where it is animated to orbit the Earth. The simulation also implements interactive features: users can highlight the satellite by hovering the mouse over it, which changes its color for visual feedback. This integration of Satellite.glb not only adds visual realism and technical depth to the simulation but also demonstrates how external 3D assets can be dynamically incorporated and manipulated within a modern web-based 3D application.

textures

The textures folder, located within the assets directory, contains all the high-resolution image assets used to enhance the visual realism of the Day/Night Earth Simulation. This folder includes files such as earth_day.jpg, earth_night.jpg, earth_clouds.jpg (or .png), and sun.jpg. Each of these textures serves a specific purpose: earth_day.jpg provides the detailed daytime surface of the Earth, earth_night.jpg is used as an emissive map to simulate city lights on the night side, earth_clouds.jpg or .png adds a semi-transparent atmospheric cloud layer, and sun.jpg is mapped onto the Sun mesh to give it a realistic appearance. The use of both JPEG and PNG formats allows for a balance between image quality, file size, and support for transparency where needed.

These texture assets are loaded at runtime using Three.js's TextureLoader within the main simulation script (see index.html or main.js). Once loaded, they are applied to the corresponding 3D meshes, such as the Earth, clouds, and Sun, using appropriate material properties (e.g., map, emissiveMap, transparent). This modular approach to asset management makes it easy for developers to update or replace textures to customize the simulation's appearance. The organization of the textures folder ensures that all visual resources are centralized, maintainable, and easily referenced throughout the project's codebase. The summary of the function is

- **earth_day.jpg:** Daytime surface texture for the Earth.
- **earth_night.jpg:** Nighttime (city lights) texture for the Earth.
- **earth_clouds.jpg / earth_clouds.png:** Cloud layer texture (PNG preferred for transparency).
- **sun.jpg:** Surface texture for the Sun.

style.css

The style.css file is intended to provide custom CSS styling for the Day/Night Earth Simulation project, ensuring that the simulation canvas and any related user interface elements are visually appealing and responsive. By centralizing layout and appearance rules, this stylesheet allows for consistent control over aspects such as background color, canvas sizing, margins, and overflow behavior. While some essential styles are inlined within index.html, style.css offers a scalable location for additional or more complex styles, supporting maintainability and easy customization as the project evolves.

Usage Instructions

1. **Prerequisites:**
   - Modern web browser with WebGL support.
   - No server required; can be run locally by opening index.html.
2. **Running the Simulation:**
   - Open index.html in a web browser.
   - The simulation will initialize automatically.
3. **User Controls:**
   - **Orbit:** Click and drag to rotate the camera.
   - **Zoom:** Scroll to zoom in/out.
   - **Satellite Highlight:** Hover over the satellite to highlight it.
   - **Moon Orbit Toggle:** Click the moon to pause/resume its orbit.

Customization Guidelines

- **Textures:** Replace images in textures to update planetary or environmental visuals.
- **Models:** Replace `Satellite.glb` with another GLTF model for different satellites or objects.
- **Animation Parameters:** Adjust rotation/orbit speeds and lighting in the JavaScript code for different effects.
- **Feature Extension:** Additional celestial bodies or UI elements can be added by extending the JavaScript logic.