Addis Ababa Institute of Technology

School of Information Technology and Engineering
Department of IT/SE Eng.

Course: Graphics Project Proposal.

Title: Interactive Solar System Simulator with Three.js

Team Members

1. Abigiya Daniel	UGR/5110/15	4
2. Ashenafi Godana	UGR/7906/14	4
3. Jiru Gutema	UGR/5902/15	4
4. Blen Debebe	UGR/5297/15	4
5. Kalafat Tezazu	UGR/6048/15	4

Submitted to:Mr. Addisalem G.

Date : June 22, 2025

Table of Contents

Interactive Solar System Three.js - Technical Documentation

Project Overview

The Interactive Solar System is a web-based 3D visualization application built with Three.js that renders our solar system with accurate orbital mechanics, interactive planetary information, and real-time animation. This documentation provides comprehensive technical details for developers working with or extending this codebase.

Installation & Setup

Prerequisites

- Modern web browser with WebGL support
- Local web server (for ES6 module support)
- No build tools required - uses CDN imports

Quick Start

```
# Clone or download project
# Serve files via local server
python -m http.server 8000
# or
npx serve .
```

Dependencies

- **Three.js v0.164.1**: Loaded via CDN with import maps
- **OrbitControls**: Three.js addon for camera navigation

Architecture Documentation

Module Structure

`main.js` - Application Entry Point

Purpose: Initializes the entire application and coordinates all modules

Key Functions:

- Scene, camera, and renderer setup
- Module orchestration and initialization

- Window resize handling
- Loading screen management

Critical Code Sections:

```
// Camera configuration
const camera = new THREE.PerspectiveCamera(60, window.innerWidth/window.inne
camera.position.set(0, orbitalScales.distanceFactor*3.5, orbitalScales.dista

// Renderer setup with shadows
const renderer = new THREE.WebGLRenderer({ canvas, antialias:true, alpha:tru
renderer.shadowMap.enabled = true;
```

`data.js` - Astronomical Data Management

Purpose: Centralized storage of planetary data and scaling parameters

Data Structure:

```
export const solarSystemData = [
  {
    name: "Earth",
    radiusKm: 6371,                     // Real radius in kilometers
    visualRadius: 1 * sizeFactor * 5,   // Scaled for visualization
    orbitalRadiusAU: 1.0,              // Distance from Sun in AU
    orbitalSpeedFactor: 1.0,           // Relative orbital speed
    rotationSpeedFactor: 0.0041,       // Rotation speed factor
    // Display information
    distanceFromSunDisplay: "149.6 million km",
    orbitalPeriodDisplay: "365.25 Earth days",
    atmosphericComposition: "Nitrogen (78%), Oxygen (21%)",
    moonsDisplay: "1 (The Moon)"
  }
];
```

Scaling Configuration:

```
export const orbitalScales = {
    distanceFactor: 50,  // Orbital distance multiplier
    sizeFactor: 1,       // Planet size multiplier
    timeScale: 0.05      // Animation speed multiplier
};
```

`bodies.js` - 3D Object Creation

Purpose: Creates and configures all celestial bodies with proper materials and geometry

Key Implementation:

```
export function createBodies(scene, solarSystemData, orbitalScales, sunLight
  const celestialBodies = [];
  solarSystemData.forEach(data => {
    // Sphere geometry with high detail
    const geom = new THREE.SphereGeometry(data.visualRadius, 64, 32);

    // Material selection based on object type
    let mat;
    if (data.isStar) {
      mat = new THREE.MeshBasicMaterial({
        color: data.color,
        map: createGlowTexture(data.color)
      });
    } else {
      mat = new THREE.MeshStandardMaterial({
        color: data.color,
        roughness: 0.7,
        metalness: 0.2
      });
    }

    // Orbital path visualization
    if (!data.isStar) {
      const path = new THREE.Path().absellipse(
        0, 0,
        mesh.userData.orbitalRadiusScene,
        mesh.userData.orbitalRadiusScene,
        0, Math.PI*2, false, 0
      );
      const geoLine = new THREE.BufferGeometry().setFromPoints(path.getSpace
      const matLine = new THREE.LineBasicMaterial({
        color: 0x555555,
        transparent: true,
        opacity: 0.3
      });
    }
  });
}
```

animation.js - Physics & Motion System

Purpose: Handles orbital mechanics and planetary rotation

Animation Loop Structure:

```
export function animateLoop(renderer, scene, camera, controls, celestialBodi
  let lastTime = 0;
  function loop(now) {
    requestAnimationFrame(loop);
    now *= 0.001; // Convert to seconds
    const dt = now - lastTime;
    lastTime = now;

    // Delta time validation
    if (!isNaN(dt) && dt > 0 && dt < 0.2) {
      const baseOrb = orbitalScales.timeScale * 5;
      const baseRot = orbitalScales.timeScale * 20;

      celestialBodies.forEach(body => {
        // Planetary rotation
        if (body.userData.rotationSpeedFactor) {
          body.rotation.y += body.userData.rotationSpeedFactor * baseRot * d
        }

        // Orbital motion with Kepler's law approximation
        if (!body.userData.isStar) {
          const adj = body.userData.orbitalRadiusAU * 0.4 + 0.6;
          const eff = body.userData.orbitalSpeedFactor * baseOrb * dt / adj;
          body.userData.angle += eff;

          // Update position
          body.position.x = body.userData.orbitalRadiusScene * Math.cos(body
          body.position.z = body.userData.orbitalRadiusScene * Math.sin(body
        }
      });
    }
  }
}
```

`interaction.js` - User Interface System

Purpose: Handles mouse interaction and information display

Raycasting Implementation:

```
export function setupInteraction(canvas, scene, camera, bodies, elems) {
  const raycaster = new THREE.Raycaster();
  const mouse = new THREE.Vector2();

  canvas.addEventListener('click', e => {
    // Convert mouse coordinates to normalized device coordinates
    const r = canvas.getBoundingClientRect();
    mouse.x = ((e.clientX - r.left) / r.width) * 2 - 1;
    mouse.y = -((e.clientY - r.top) / r.height) * 2 + 1;

    // Perform raycasting
    raycaster.setFromCamera(mouse, camera);
    const hits = raycaster.intersectObjects(bodies, true);

    if (hits.length) {
      // Find the clicked celestial body
      let obj = hits[0].object;
      while (obj.parent && obj.parent !== scene && !obj.userData.name) {
        obj = obj.parent;
      }

      // Update information panel
      const d = obj.userData;
      elems.planetNameEl.textContent = d.name;
      elems.planetDiameterEl.textContent = `${(d.radiusKm*2).toLocaleString(
      // ... populate other fields
      elems.infoCard.classList.add('visible');
    }
  });
}
```

`scene.js` - 3D Environment Setup

Purpose: Creates the 3D scene and starfield background

Starfield Generation:

```
export function createStarfield(scene) {
  const geo = new THREE.BufferGeometry();
  const verts = [];

  // Generate 10,000 random star positions
  for (let i = 0; i < 10000; i++) {
    verts.push(
      THREE.MathUtils.randFloatSpread(20000),
      THREE.MathUtils.randFloatSpread(20000),
      THREE.MathUtils.randFloatSpread(20000)
    );
  }

  geo.setAttribute('position', new THREE.Float32BufferAttribute(verts, 3));
  const mat = new THREE.PointsMaterial({
    color: 0xaaaaaa,
    size: 0.7,
    transparent: true,
    opacity: 0.5
  });
  scene.add(new THREE.Points(geo, mat));
}
```

`controls.js` - Camera Navigation

Purpose: Configures orbital camera controls

```
export function setupControls(camera, rendererDom, orbitalScales) {
  const controls = new OrbitControls(camera, rendererDom);
  controls.enableDamping = true;
  controls.dampingFactor = 0.04;
  controls.minDistance = orbitalScales.distanceFactor * 0.2;
  controls.maxDistance = orbitalScales.distanceFactor * 1000;
  controls.target.set(0, 0, 0); // Focus on Sun
  return controls;
}
```

API Reference

Configuration Objects

orbitalScales

- `distanceFactor`: Multiplier for orbital distances (default: 50)
- `sizeFactor`: Multiplier for planet sizes (default: 1)
- `timeScale`: Animation speed multiplier (default: 0.05)

Planet Data Structure

- `name`: Display name
- `radiusKm`: Real radius in kilometers
- `visualRadius`: Scaled radius for 3D rendering
- `color`: Hex color value
- `orbitalRadiusAU`: Distance from Sun in Astronomical Units
- `orbitalSpeedFactor`: Relative orbital speed
- `rotationSpeedFactor`: Rotation speed (negative for retrograde)
- Display fields: `distanceFromSunDisplay`, `orbitalPeriodDisplay`, etc.

Key Functions

createBodies(scene, data, scales, light)

- Creates all celestial body meshes
- Returns array of Three.js mesh objects
- Automatically generates orbital paths

animateLoop(renderer, scene, camera, controls, bodies, scales)

- Main animation loop using requestAnimationFrame
- Handles orbital motion and rotation
- Updates camera controls

setupInteraction(canvas, scene, camera, bodies, elements)

- Configures mouse click detection
- Manages information panel display
- Handles UI event listeners

Performance Optimization

Rendering Optimizations

- **Geometry LOD**: 64-segment spheres balance quality and performance
- **Material Efficiency**: Shared materials where possible
- **Shadow Optimization**: Selective shadow casting/receiving
- **Frustum Culling**: Automatic by Three.js

Animation Optimizations

- **Delta Time**: Frame-rate independent animation
- **Bounds Checking**: Delta time validation prevents large jumps
- **Efficient Updates**: Direct position calculation vs. transform matrices

Memory Management

- **Resource Cleanup**: Proper disposal of geometries and materials
- **Event Listener Management**: Cleanup on component destruction
- **Texture Optimization**: Efficient texture loading and caching

Troubleshooting

Common Issues

Module Loading Errors

- Ensure serving via HTTP/HTTPS (not file://)
- Check import map configuration in index.html
- Verify Three.js CDN availability

Performance Issues

- Reduce sphere geometry segments for lower-end devices
- Adjust `orbitalScales.timeScale` for smoother animation
- Check browser WebGL support and hardware acceleration

Interaction Problems

- Verify raycaster setup and mouse coordinate conversion
- Check z-index and CSS positioning of UI elements
- Ensure proper event listener attachment

Debug Tools

```javascript
// Add to main.js for debugging
const stats = new Stats();
document.body.appendChild(stats.dom);

// In animation loop
stats.update();
```

Extension Guidelines

Adding New Planets/Objects

1. Add data entry to `solarSystemData` array
2. Include all required fields (name, radius, orbital parameters)
3. Test scaling and visual appearance
4. Update information panel fields if needed

Custom Materials/Textures

1. Create texture loading utilities in `utils.js`
2. Modify material creation in `bodies.js`
3. Consider performance impact of additional textures
4. Implement proper disposal for memory management

UI Enhancements

1. Extend `infoElements` object in `main.js`
2. Add corresponding HTML elements
3. Update interaction handlers in `interaction.js`
4. Style new elements in `styles.css`

This documentation provides the foundation for understanding, maintaining, and extending the Interactive Solar System application. The modular architecture ensures that modifications can be made to individual components without affecting the entire system.