CS 405 - Project Report 3 Emre Soylu 26582

This project focuses on implementing a solar system simulation by completing three distinct tasks, each requiring modifications to specific files. The provided files serve distinct purposes and are foundational to the project's implementation:

meshDrawer.js : Handles the rendering of meshes, with Task 2 involving modifications to its fragment shader to implement diffuse and specular lighting.

sceneNode.js : implements the nodes in the scene graph, with Task 1 focusing on implementing its `draw` method to propagate transformations from parent to child nodes.

project3.html : serves as the main entry point for the program, with Task 3 requiring additions to the scene graph to include Mars as a child node of the Sun.

Throughout the project, guidance and references were utilized. For Task 2, the diffuse and specular lighting implementation was inspired by a similar application in Project 2. Task 3 relied on structured assistance from ChatGPT to integrate Mars into the solar system while adhering to the scene graph structure.

This report will detail the step-by-step execution of each task, highlighting the modifications made, the challenges faced, and the solutions implemented.

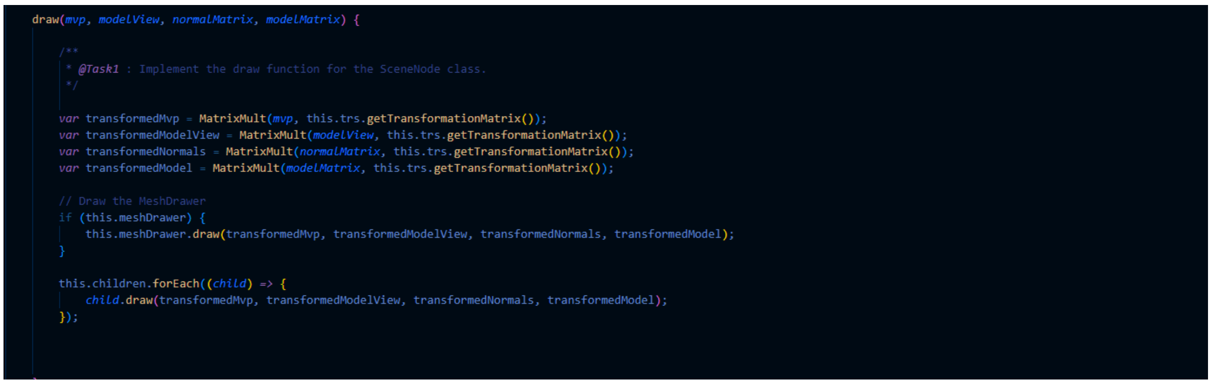
TASK 1

Task 1: Implementing the Draw Function for the Scene Graph

The objective of Task 1 was to complete the `draw` function for the `SceneNode` class in the `sceneNode.js` file. The key challenge was to ensure that any transformations applied to a parent node were correctly propagated to its child nodes within the scene graph. This hierarchical transformation is essential for accurately rendering objects in a structured environment, such as a solar system.

I implemented the `draw` function as follows:

The Code:



```
draw(mvp, modelView, normalMatrix, modelMatrix) {

    /**
     * @Task1 : Implement the draw function for the SceneNode class.
     */

    var transformedMvp = MatrixMult(mvp, this.trs.getTransformationMatrix());
    var transformedModelView = MatrixMult(modelView, this.trs.getTransformationMatrix());
    var transformedNormals = MatrixMult(normalMatrix, this.trs.getTransformationMatrix());
    var transformedModel = MatrixMult(modelMatrix, this.trs.getTransformationMatrix());

    // Draw the MeshDrawer
    if (this.meshDrawer) {
        this.meshDrawer.draw(transformedMvp, transformedModelView, transformedNormals, transformedModel);
    }

    this.children.forEach((child) => {
        child.draw(transformedMvp, transformedModelView, transformedNormals, transformedModel);
    });
```

Step-by-Step Explanation:

1. Getting the Current Node's Transformation:

```
this.trs.getTransformationMatrix()
```

- This method retrieves the transformation matrix for the current node, which includes the node's translation, rotation, and scaling. Every object in the scene has its own transformations, but they are influenced by their parent's transformations.
- **Why?** This ensures that the node's position and orientation are calculated relative to its parent, maintaining the hierarchical structure.

2. Updating the Model-View-Projection (MVP) Matrix:

```
var transformedMvp = MatrixMult(mvp,
this.trs.getTransformationMatrix());
```

- The MVP matrix determines where an object appears on the screen, combining its position, view, and perspective transformations.
- **Why?** By multiplying the parent's MVP matrix with the current node's transformation matrix, the object's final position on the screen includes both its own transformations and those of its parent.

3. Updating the Model-View Matrix:

```
var transformedModelView = MatrixMult(modelView,
this.trs.getTransformationMatrix());
```

- The Model-View matrix is used to position the object relative to the camera.
- **Why?** Without updating this matrix, the object might not appear correctly in the scene, as its position would not account for the camera's view.

4. Updating the Normal Matrix:

```
var transformedNormals = MatrixMult(normalMatrix,
this.trs.getTransformationMatrix());
```

- The Normal matrix transforms the surface normals of an object. These normals are critical for calculating how light interacts with the surface.
- **Why?** If the normals aren't transformed properly, the lighting (such as shadows or highlights) will not align with the object's orientation.

5. Updating the Model Matrix:

```
var transformedModel = MatrixMult(modelMatrix,
this.trs.getTransformationMatrix());
```

- The Model matrix holds the object's local transformations (scaling, rotation, and translation).
- **Why?** Keeping the Model matrix updated ensures the object retains its individual transformations relative to its parent.

---

6. Drawing the Current Node:

```
if (this.meshDrawer) {

    this.meshDrawer.draw(transformedMvp, transformedModelView,
transformedNormals, transformedModel);

}
```

- If the current node has a `meshDrawer` (a class responsible for rendering objects), it uses the updated matrices to draw the object.
- **Why?** This ensures the object appears in the correct position and orientation on the screen, with accurate lighting effects.

---

7. Drawing the Child Nodes:

javascript

Kodu kopyala

```
this.children.forEach((child) => {

    child.draw(transformedMvp, transformedModelView,
transformedNormals, transformedModel);

});
```

- The function loops through all child nodes of the current node and calls their `draw` method, passing the updated matrices.
- **Why?** This recursive approach ensures that all child nodes inherit the transformations of their parent. For example, if the Sun moves, the Earth and Moon (as child nodes) will move with it.

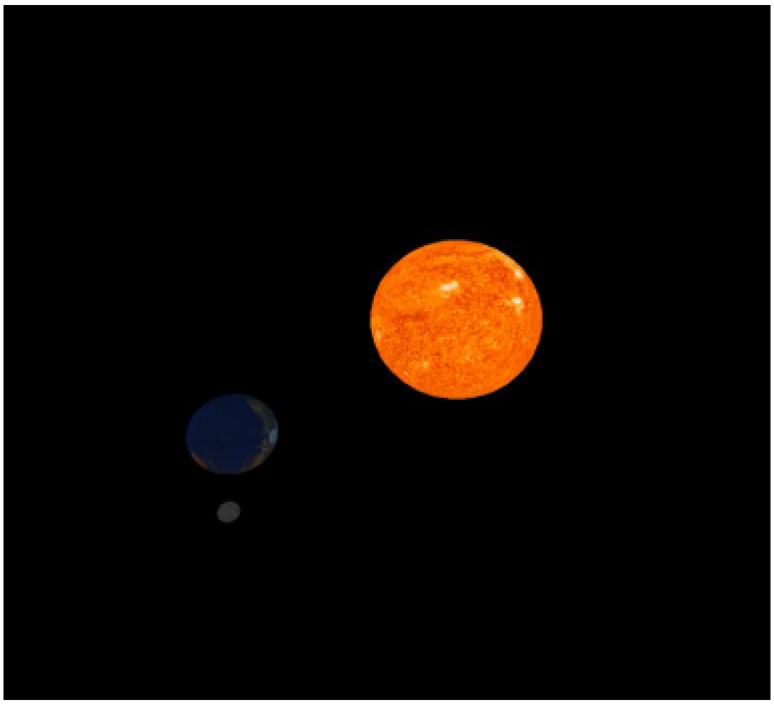
Why This Approach Works:

The hierarchical structure of the scene graph means that transformations applied to a parent affect all its children. For example, if the Sun rotates, its child nodes (Earth, Mars, and Moon) should rotate around it. By calculating and updating the transformation matrices at each level, the `draw` function ensures that every node appears in the correct position with the proper lighting.

Outcome:

This implementation was successful. The draw function allowed transformations to propagate through the scene graph, ensuring that parent and child relationships were respected. Objects like the Earth and Moon moved naturally with the Sun while maintaining their individual transformations, resulting in a dynamic and realistic solar system simulation.

Through this process, I gained a deeper understanding of how scene graphs work and how transformations can be passed down hierarchically to create complex and interactive graphics.



TASK 2 : Adding Diffuse and Specular Lighting to the Fragment Shader

In Task 2, the goal was to enhance the existing fragment shader in `meshDrawer.js` to include proper calculations for **diffuse** and **specular** lighting. The current implementation only supported ambient lighting, which resulted in flat shading. By implementing these lighting techniques, the rendered objects could display realistic lighting effects based on their orientation to the light source and the viewer's perspective.

Fragment Shader Code Update:

```
///////////////////////////////////////////////////////////////////////////////
// PLEASE DO NOT CHANGE ANYTHING ABOVE !!!
// Calculate the diffuse and specular lighting below.

// Diffuse Lighting: Lambert's cosine law
diff = max(dot(normal, lightdir), 0.0);

// Specular Lighting: Phong reflection model
vec3 viewDir = normalize(vPosition); // Direction to the camera (viewer)
vec3 reflectDir = reflect(-lightdir, normal); // Reflection vector
spec = pow(max(dot(viewDir, reflectDir), 0.0), phongExp);


// PLEASE DO NOT CHANGE ANYTHING BELOW !!!
///////////////////////////////////////////////////////////////////////////////
```

Explanation of the Code:
Diffuse Lighting (Lambert's Cosine Law):

diff = max(dot(normal, lightdir), 0.0);

1.
   - **Purpose**: This calculates the diffuse lighting, which simulates the way light scatters across a rough surface.
   - **dot(normal, lightdir)**:
     - This computes the cosine of the angle between the surface normal (`normal`) and the light direction (`lightdir`).
     - A value closer to 1 means the surface is facing the light source directly, resulting in stronger illumination.
   - **max(..., 0.0)**:
     - Ensures the value is non-negative. If the angle between the normal and light direction exceeds 90 degrees, the surface is in shadow, and the diffuse lighting is zero.
   - **Visual Effect**: Diffuse lighting makes the object's surface appear brighter when facing the light source and darker when angled away.

Specular Lighting (Phong Reflection Model):

vec3 viewDir = normalize(vPosition); // Direction to the camera (viewer)
vec3 reflectDir = reflect(lightdir, normal); // Reflection vector
spec = pow(max(dot(viewDir, reflectDir), 0.0), phongExp);

2.
   - **Purpose**: This calculates the specular lighting, which simulates the shiny highlights on a smooth surface caused by direct reflection of the light source.
   - **viewDir**:
     - Represents the direction from the fragment to the viewer (camera). This is normalized to ensure proper calculations.

    - **reflectDir**:
        - The reflection vector is calculated using the `reflect()` function, which computes the direction in which light reflects off the surface.
    - **dot(viewDir, reflectDir)**:
        - This calculates the cosine of the angle between the viewer's direction and the reflection direction. The closer this value is to 1, the shinier the surface will appear in that region.
    - **pow(..., phongExp)**:
        - Applies the Phong exponent (`phongExp`) to control the sharpness of the highlight. A higher value results in a smaller, more focused highlight, simulating shinier surfaces.
    - **Visual Effect**: Specular lighting creates shiny highlights that vary based on the viewer's position relative to the light source, adding realism to the object.

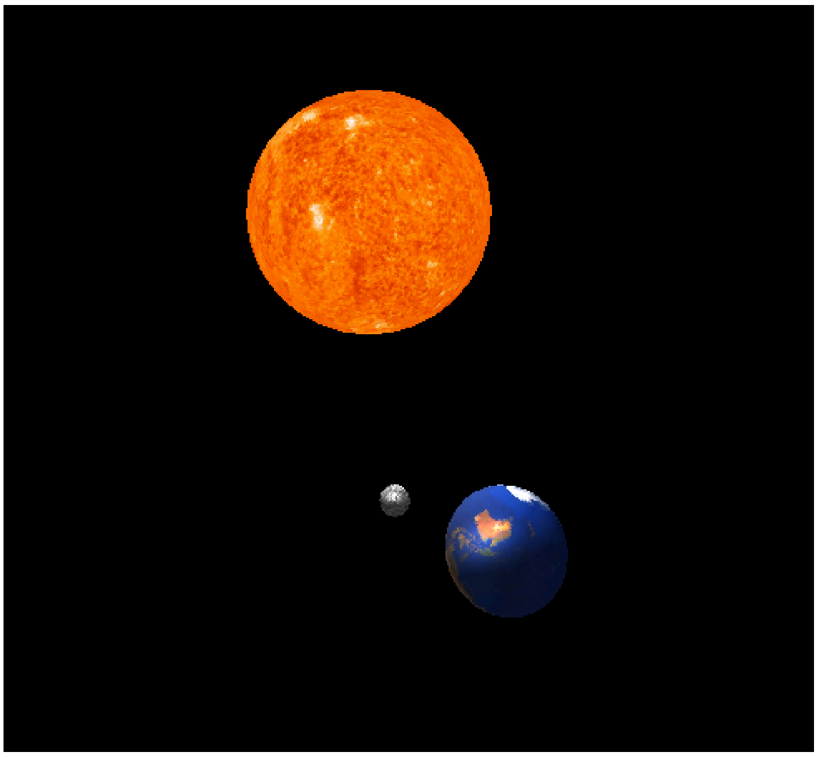
Implementation Process:

- I integrated this code into the fragment shader within `meshDrawer.js`, ensuring that both diffuse and specular lighting components worked alongside the existing ambient lighting.
- These lighting calculations were blended with the texture color of the objects to produce the final rendered result.

Outcome:

The task was completed successfully. After implementing these changes, the objects in the scene displayed more realistic lighting effects:

1. **Diffuse Lighting** made the surfaces appear brighter or darker based on their angle to the light source, creating a sense of depth and realism.
2. **Specular Highlights** introduced shiny reflections, particularly noticeable on smooth surfaces, enhancing the dynamic appearance of the objects.



TASK 3 -

Task 3: Adding Mars to the Solar System

In Task 3, the objective was to add the planet Mars to the solar system simulation. To achieve this, Mars needed to be drawn in the scene and integrated into the scene graph as a child node of the Sun. This required creating a Mars node with specific transformations and attributes to correctly position and animate it within the system.

Requirements:

1. **Child of the Sun**:
   - Mars was added as a child node to the Sun in the scene graph hierarchy.
2. **Mesh Object**:
   - Mars used the provided "sphere" mesh object for its geometry.

3. **Translation**:
    - Mars was translated by `-6` units on the X-axis relative to the Sun.
4. **Scaling**:
    - Mars was scaled uniformly by `0.35` across all axes (`x`, `y`, and `z`) to represent its size relative to other celestial bodies.
5. **Rotation**:
    - Mars was programmed to rotate around its Z-axis at `1.5` times the Sun's rotation speed.

Implementation:

I modified the `project3.html` file by adding the necessary code for Mars. Below is a summary of what was added:

Mars Node Creation:

```
marsMeshDrawer = new MeshDrawer(); // Create Mars mesh drawer

marsMeshDrawer.setMesh(sphereBuffers.positionBuffer, sphereBuffers.texCoordBuffer, sphereBuffers.normalBuffer);

setTextureImg(marsMeshDrawer, "https://i.imgur.com/Mwsa16j.jpeg"); // Set Mars texture

marsTrs = new TRS();

marsTrs.setTranslation(-6, 0, 0); // Translate Mars -6 units on X-axis

marsTrs.setScale(0.35, 0.35, 0.35); // Scale Mars to 0.35

marsNode = new SceneNode(marsMeshDrawer, marsTrs, sunNode); // Add Mars as a child of the Sun node
```

Mars Rotation in `renderLoop`:

```
marsNode.trs.setRotation(0, 0, zRotation * 1.5); // Mars rotates 1.5 times faster than the Sun
```

Outcome:

This task was completed successfully with the following results:

1. **Scene Integration**:
    - Mars was added to the solar system as a child of the Sun, inheriting the Sun's transformations while maintaining its independent translation, scaling, and rotation.
2. **Visual Appearance**:
    - Mars was correctly positioned at `-6` units on the X-axis, scaled to `0.35`, and rendered with its unique texture.
    - The rotation logic ensured that Mars rotated around its Z-axis 1.5 times faster than the Sun, creating realistic orbital dynamics.
3. **Assistance**:
    - This task required integrating multiple concepts, including hierarchical transformations and texture application. To ensure the implementation was efficient and accurate, I utilized ChatGPT for guidance in structuring the scene graph and rotation logic.