CS 405 Project 3 Report

Batur Karakaya - 28881

January 15, 2024

Task 1

For this task it was asked to implement the "draw" function for the sceneNode class. In order for drawing the objects correctly, we need the correctly calculated matrices.

- Model-View-Projection Matrix Multiplication: The MVP matrix is a combination of the Model, View, and Projection matrices. MVP matrix is used to transform the coordinates of objects from world space to clip space, which is used for rendering on the screen.

  Multiplying the MVP matrix with the node's transformation matrix applies the node's local transformations (translation, rotation, scale) to the MVP matrix. This is necessary to position and orient the node correctly in the scene.

- Model-View Matrix Multiplication: The Model-View matrix is the result of multiplying the Model matrix with the View matrix. It transforms coordinates from model space to view space. By multiplying it with the node's transformation matrix, you apply the node's local transformations to the model-view matrix, placing the node correctly relative to the camera's view.

- Normal Matrix Multiplication: The normal matrix is used to transform normal vectors, which are essential for lighting calculations.

Multiplying the normal matrix with the node's transformation ensures that the normals are correctly oriented in relation to the node's transformations.

- Model Matrix Multiplication: The Model matrix represents the transformations that are applied to the object's coordinates to position it in the world. This multiplication applies the node's local transformations to the model matrix.

-
  - const mvp_multiplied = MatrixMult(mvp, this.trs.getTransformationMatrix());
  - const mv_multiplied = MatrixMult(modelView, this.trs.getTransformationMatrix());
  - const normal_multiplied = MatrixMult(normalMatrix, this.trs.getTransformationMatrix());
  - const mm_multiplied = MatrixMult(modelMatrix, this.trs.getTransformationMatrix());

- After implementing this method properly, any transformation applied on the parent also propagate to the children nodes.

Task 2

For this task it was asked to modify the fragment shader namely "MeshFS" to support not just ambient lighting, but also the diffuse and specular lighting methods by calculating them correctly.

- Diffuse Lighting Calculation: Here, we first calculate the dot product of the normal vector and the lightDirection. This dot product measures the cosine of the angle between the normal and the light direction. "max(..., 0.0);" ensures that the diffuse component is not negative. If the dot product is negative (which happens when the light source is behind the surface), the diffuse lighting is set to zero, as the surface is not directly illuminated by the light source in such cases. Lastly, diffuse factor "diff" represents how much direct light a surface receives based on its orientation to the light source.

- Reflection Direction Calculation: Here, we calculated reflection direction of the light. "-lightdir" is used during the calculation because reflect expects the incident vector to be pointing away from the surface.
- View Direction Calculation: Here, we calculated the view direction, which is the direction from the fragment to the viewer. "-fragpos" is used here because it is assumed that the viewer is at the origin.
- Specular Lighting: Here, we calculated the specular lighting, which represent the intensity of the specular highlight based on the viewer's position relative to the light's reflection direction. "dot(viewDirection, reflectDirection)" computes the dot product between the view direction and the reflection direction. "max(dot(viewDirection, reflectDirection), 0.0)" ensures that the specular component is not negative, similar to the diffuse calculation, and "pow(max(dot(viewDirection, reflectDirection), 0.0), phongExp)" raises the dot product to the power of phongExp, a value that controls the shininess of the specular highlight.
-
  - diff=max(dot(normal, lightdir), 0.0);
  - vec3 reflectDirection = reflect(-lightdir, normal);
  - this.ambientLoc = gl.getUniformLocation(this.prog, 'ambient');
  - vec3 viewDirection = normalize(-fragPos);
  - spec = pow(max(dot(viewDirection, reflectDirection), 0.0), phongExp);

Task 3

For this task it was asked to add Mars planet to the solar system. In order to do so, we were asked to draw Mars in the scene and add it to scene graph as the child of the Sun node.

- Creating Mars' Mesh Drawer: Here, we instantiated a new MeshDrawer object for Mars. This object responsible for handling the rendering of the mesh (the 3D model) of Mars.

- Setting Mesh for Mars: Here, the "setMesh" function is called on "marsMeshDrawer" with sphereBuffers's position, texture coordinate, and normal buffers. These buffers define the geometry of Mars.
- Applying Texture to Mars: Here, we applied a texture that is loaded from the given URL by using setTextureImg to marsMeshDrawer with the help of "setTextureImg()" method.
- Mars' Transformations: Here, we first created a new TRS (Translation, Rotation, Scale) object for Mars, then we applied the specified transformations on that object.
- Creating Mars' Scene Node: Here, a scene node is created for Mars that allows us to put Mars into the scene graph. The SceneNode constructor takes marsMeshDrawer (the mesh to render Mars), marsTrs (the transformation for Mars), and sunNode (the parent node, which is the Sun). This sets up Mars as a child of the Sun in the scene graph, meaning Mars will inherit transformations applied to the Sun.
-
  - marsMeshDrawer = new MeshDrawer();
  - marsMeshDrawer.setMesh(sphereBuffers.positionBuffer, sphereBuffers.texCoordBuffer, sphereBuffers.normalBuffer);
  - setTextureImg(marsMeshDrawer, "https://i.imgur.com/Mwsa16j.jpeg");
  - marsTrs = new TRS();
  - marsTrs.setTranslation(-6, 0, 5);
  - marsTrs.setScale(0.35, 0.35, 0.35);
  - marsNode = new SceneNode(marsMeshDrawer, marsTrs, sunNode);
- In the end, we call "renderLoop()" method which initializes the rendering process.

GitHub Link: https://github.com/Exion007/CS405/tree/main/Project3