Scientific Visualization

Robot Illumination

Anand Kamble
Department of Scientific Computing
Florida State University

1 Introduction

In this project, we will be adding illumination to our robot which we developed in the previous assignment. We will be adding different materials to different parts of the robot and also a light source in the scene.

2 Vertex Data

In this project for the illumination, we will also need the normal vectors for the vertices. These are defined along with the vertex position, like below:

Listing 1: main.cpp

```
float vertices[] = {
// |-----Coordinates----|------Normal-----|
    -0.5f, -0.5f, -0.5f, 0.0f, 0.0f, -1.0f,
     0.5f, -0.5f, -0.5f, 0.0f, 0.0f, -1.0f,
    ...
    -0.5f,  0.5f, -0.5f, 0.0f, 1.0f, 0.0f};
```

Here the first three values represent the x, y, and z coordinates of the vertex and the next three values represent the normal vector of the vertex.

3 Vertex Shader

In the vertex shader, we are passing the normal directions to the fragment shader. And we are also applying the projection and view matrix.

Listing 2: material.vert

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;

out vec3 FragPos;
out vec3 Normal;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = mat3(transpose(inverse(model))) * aNormal;

    gl_Position = projection * view * vec4(FragPos, 1.0);
}
```

4 Fragment Shader

4.1 Selecting a material

To select a material and apply it's properties to a part, we will be using the `getNewMaterial` function which returns a material object, this object looks like:

Listing 3: utils.cpp

```
struct Material
{
    std::string name;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};
```

4.2 Applying the material

Using values from this object we are updating our material properties by passing it to the shader like this:

Listing 4: main.cpp

```
ourShader.setVec3("material.ambient", Selected_Material->ambient);
ourShader.setVec3("material.diffuse", Selected_Material->diffuse);
ourShader.setVec3("material.specular", Selected_Material->specular);
ourShader.setFloat("material.shininess", Selected_Material->shininess);
```

Using these values, the fragment shader is calculating the color of each pixel. The fragment shader calculates the ambient, diffuse, and specular highlights.

Since our project has a single light source, we will be using,

$$I = I_E + K_A I_{AL} + K_D (N \cdot L) I_L + K_S (C \cdot R)^n I_L$$

Below is how these calculations are implemented in the fragment shader.

Listing 5: material.frag

```
// ambient
vec3 ambient = light.ambient * material.ambient;

// diffuse
vec3 norm = normalize(Normal);
vec3 lightDir = normalize(light.position - FragPos);
float diff = max(dot(norm, lightDir), 0.0);
vec3 diffuse = light.diffuse * (diff * material.diffuse);

// specular
vec3 viewDir = normalize(viewPos - FragPos);
vec3 reflectDir = reflect(-lightDir, norm);
float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
vec3 specular = light.specular * (spec * material.specular);
```

To calculate the attenuation factor, we are using this formula,

$$f = \frac{1}{k_c + k_l d + k_q d^2}$$

In this project, we are using,

$$k_c = 1.0$$
$$k_l = 0.045$$
$$k_q = 0.0075$$

This is implemented in the shader as follows:

Listing 6: material.frag

```
float distance = length(light.position - FragPos);
float attenuation = 1.0 / (1.0 + (0.045 * distance) + (0.0075 * distance *
    distance));
```

So finally the output color will be:

Listing 7: material.frag

```
vec3 result = (ambient + diffuse + specular) * attenuation;
FragColor = vec4(result, 1.0);
```

5 Usage

To compile and run the code, you can use the Makefile included with the project. You can directly start the program by using the following command which will compile and run the binary.

```
    #!/bin/bash
    make run
```

You can control the robot arm by following keys:

- `'I'` - Move the camera forward.
- `'K'` - Move the camera backward.
- `'J'` - Move the camera to left.
- `'L'` - Move the camera to right.
- `'S'` - Rotate the lower arm.
- `'E'` - Rotate the upper arm.
- `'O'` - Open the finger.
- `'C'` - Close the finger.
- `'Cursor'` - Move the cursor to rotate the camera.

After successful execution, the materials of the robot should be updated every second. The materials are looping over all the listed materials in the `utils.cpp` file.

6 Results

The program is able to render the robot arm, with proper material and illumination.

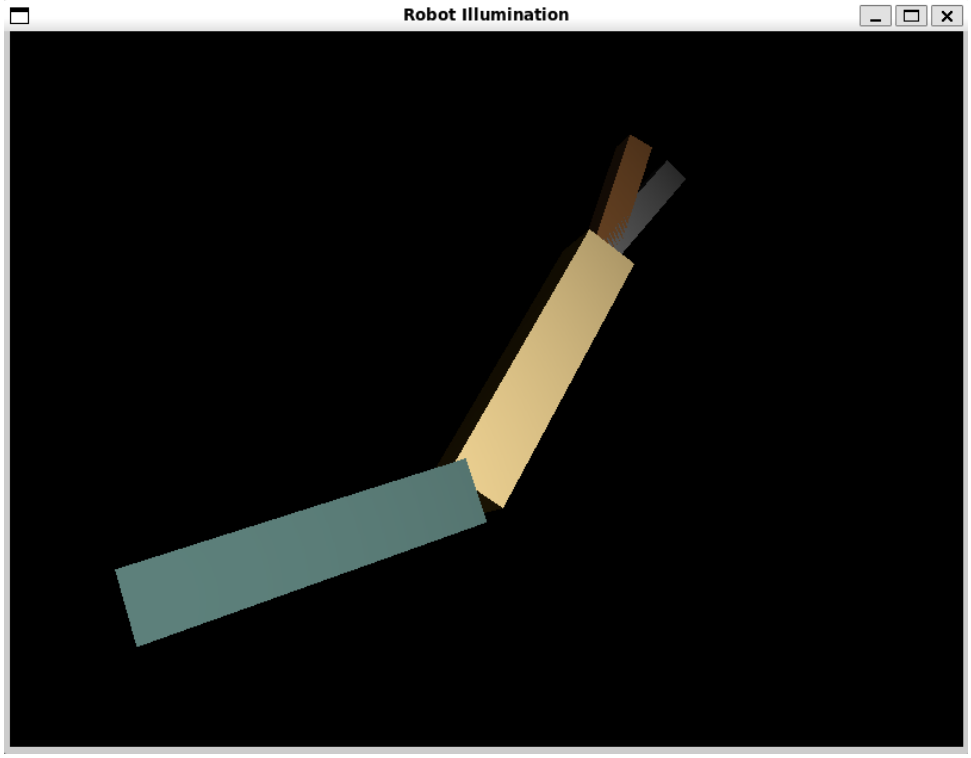
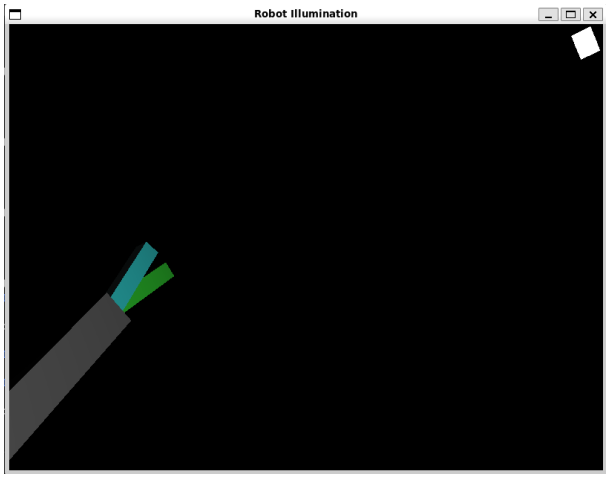
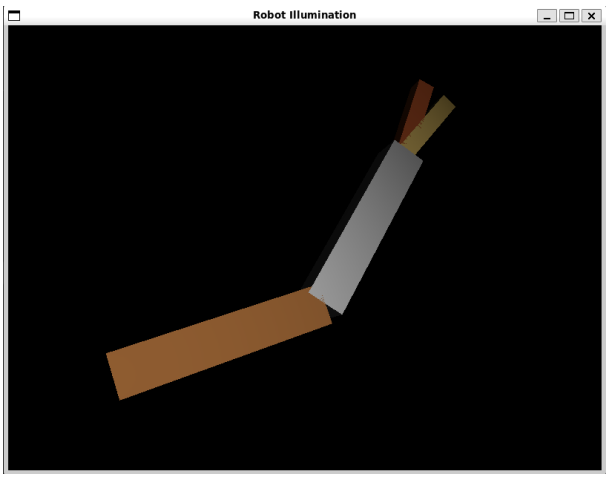


Figure 1: Output Window



(a) Rotated towards the light cube



(b) With Different material

Note: The light cube is only rendered to visualize the position of the point light.