Assignment #6 – Illumination effect

Yoke Kai Wen, 2020280598

May 5, 2021

1 Introduction

In this assignment, I modified the code in Tutorial 9_2 to create different illumination effects by using different parameters of the loaded object's material and lighting. All faces of the object have the same colour, and the object is displayed in triangular face mode and smooth mode (see Figures 1 and 2). The object and light source can be moved around with the keyboard; the object and light colour can be adjusted by the keyboard; the lighting parameters and object material parameters can also be adjusted by the keyboard. A demonstration of the effects of different parameters can be seen in the screen recording `illum.mp4`.

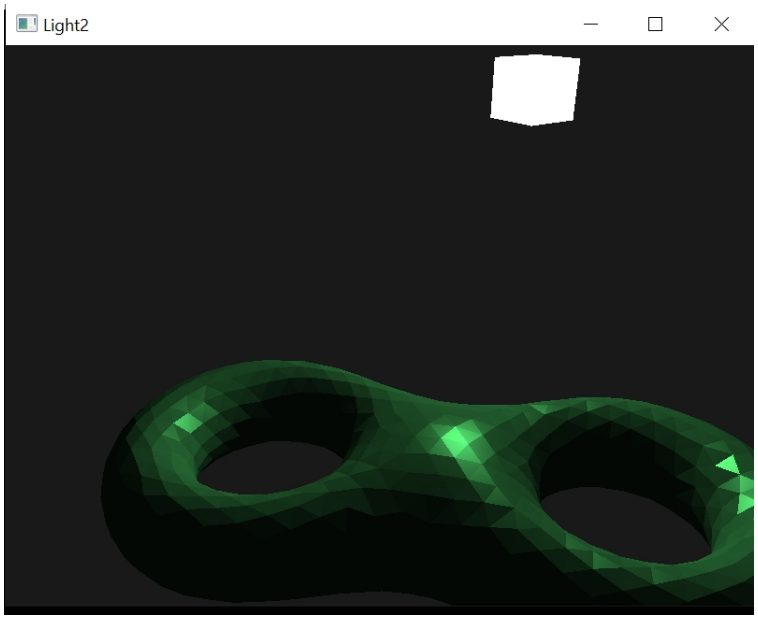

Figure 1: Illuminated object in triangular face mode

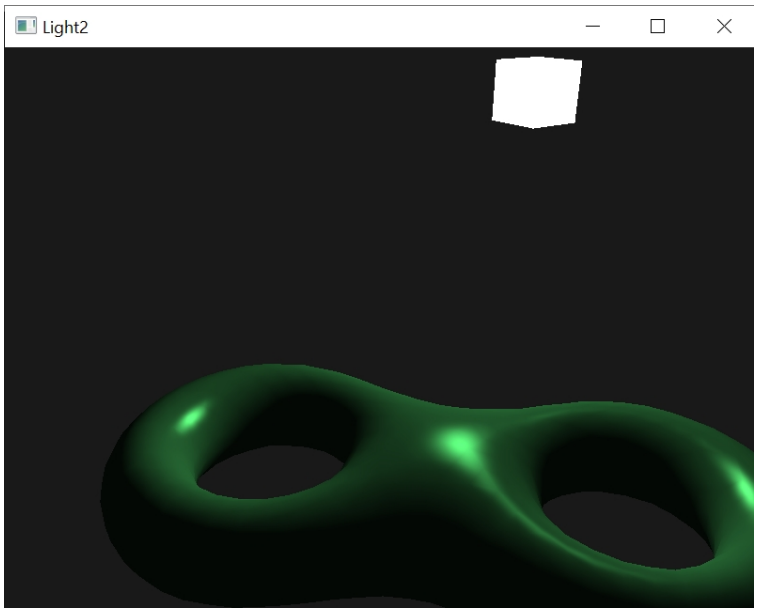

Figure 2: Illuminated object in smooth mode

2 Files and directory structure

1. Main script: main.cpp
2. Shader files in shaders/: material.vert.glsl, material.frag.glsl, lamp.frag.glsl, lamp.vert.glsl
3. Object file: eight.uniform.obj
4. Other header files: include/tut_headers/obj3dmodel.h, include/tut_headers/shader.h, include/tut_headers/camera.h
5. Set include/ and lib/ folders as include and library directories respectively in VS.

3 Usage

1. Object translation control: W, S, A, D, Z, X
2. Object rotation control: U, H, J, K, N, M
3. Light source translation control: R, T, F, G, V,B
4. Light colour control: 1, 2, 3
5. Light ambient, diffuse, specular params: 4, 5, 6
6. Material colour control: 0, 9, 8
7. Material shininess control: leftarrow, rightarrow
8. Material ambient, diffuse, specular params: uparrow, downarrow, slash

4 Design

The main work in this assignment was computing the normals and loading the object into the main script based on the object loading code from Assignment 1, while the rest of the lighting model pipeline mostly reuses the Tutorial 9_2 code.

4.1 Compute face and vertex normals of loaded object

In the object loading header file `obj3dmodel.h`, I compute the face normals and the vertex normals, which are loaded into the main script.

4.1.1 Face normals

Face normal refers to the normal vector pointing out of each triangular face, and are stored in a vector of size equivalent to the number of faces. All three vertices of the same triangular face will have the same normal vector, and each vertex can be associated with a number of face normals, resulting in the displayed model showing triangular patches on its surface as in Figure 1. To compute the face normal of a face with three vertices `v1, v2, v3`, I take the vector cross product of two edges `(v2-v1)x(v3-v1)` and normalise the result. The process of computing face normals is shown in the function below.

```
void calc_face_normals()
{
    for (int i=0; i < this->nf; i++) {
        face curr_f = this->faces[i];
        vertex v1 = this->vertices[curr_f.v1 - 1];
        vertex v2 = this->vertices[curr_f.v2 - 1];
        vertex v3 = this->vertices[curr_f.v3 - 1];

        vec e1, e2;
        e1.x = v2.x - v1.x;
        e1.y = v2.y - v1.y;
        e1.z = v2.z - v1.z;

        e2.x = v3.x - v1.x;
        e2.y = v3.y - v1.y;
        e2.z = v3.z - v1.z;

        vec facenorm;
        facenorm.x = (e1.y * e2.z) - (e1.z * e2.y);
        facenorm.y = (e1.z * e2.x) - (e1.x * e2.z);
        facenorm.z = (e1.x * e2.y) - (e1.y * e2.x);

        double mag = sqrt(facenorm.x*facenorm.x + facenorm.y*facenorm.y + facenorm
    .z*facenorm.z);
        facenorm.x /= mag;
        facenorm.y /= mag;
        facenorm.z /= mag;

        this->face_norms.push_back(facenorm);
    }
}
```

4.1.2 Vertex normals

Vertex normals are computed per vertex. Since each vertex can be associated with a number of faces, the average of the face normals of associated faces is then equal to the vertex normal. This leads to the resulting displayed model having a smooth surface as shown in Figure 2. The vertex normals are stored in a vector with size equivalent to the number of vertices.

```
void calc_vertex_normals()
{
    for (int v=0; v < this->nv; v++) {
        //get all the face norms associated with vertex v
        std::vector<vec> vnorms;
        for (int f=0; f < nf; f++) {
            face curr_f = this->faces[f];
            int v1_ind = curr_f.v1 - 1;
            int v2_ind = curr_f.v2 - 1;
            int v3_ind = curr_f.v3 - 1;

            if (v1_ind == v || v2_ind == v || v3_ind == v) {
                vnorms.push_back(this->face_norms[f]);
            }
        }

        //calculate average norm for vertex v
        vec vnorm;
        vnorm.x = 0;
        vnorm.y = 0;
        vnorm.z = 0;
        for (int i=0; i < vnorms.size(); i++) {
            vnorm.x += vnorms[i].x;
            vnorm.y += vnorms[i].y;
            vnorm.z += vnorms[i].z;
        }

        //normalise v norm
        double mag = sqrt(vnorm.x*vnorm.x + vnorm.y*vnorm.y + vnorm.z*vnorm.z);
        vnorm.x /= mag;
        vnorm.y /= mag;
        vnorm.z /= mag;

        this->vertex_norms.push_back(vnorm);
    }
}
```

4.1.3 Loading vertices into buffer object

After computing and storing the vertex normals in vectors, the vertex positions and normals of each triangular face are then loaded into vertices array buffers in the main script. Two buffers are loaded - one for face normals which produces the triangular mode display (`vertices_m1`), and one for vertex normals which produces the smooth mode display (`vertices_m2`).

```
//load object
obj3dmodel myModel("eight.uniform.obj");

// Set up vertex data (and buffer(s)) and attribute pointers
//mode 1 vertices
GLfloat* vertices_m1 = new GLfloat[18 * myModel.nf]; // each face has three
    vertices, each vertice has 3-d pos coords + 3-d face normals
```

```
for (int i = 0; i < myModel.nf; i++) {
    //fill in vertex positions
    vertices_m1[18 * i] = myModel.vertices[myModel.faces[i].v1 - 1].x;
    vertices_m1[18 * i + 1] = myModel.vertices[myModel.faces[i].v1 - 1].y;
    vertices_m1[18 * i + 2] = myModel.vertices[myModel.faces[i].v1 - 1].z;

    vertices_m1[18 * i + 6] = myModel.vertices[myModel.faces[i].v2 - 1].x;
    vertices_m1[18 * i + 7] = myModel.vertices[myModel.faces[i].v2 - 1].y;
    vertices_m1[18 * i + 8] = myModel.vertices[myModel.faces[i].v2 - 1].z;

    vertices_m1[18 * i + 12] = myModel.vertices[myModel.faces[i].v3 - 1].x;
    vertices_m1[18 * i + 13] = myModel.vertices[myModel.faces[i].v3 - 1].y;
    vertices_m1[18 * i + 14] = myModel.vertices[myModel.faces[i].v3 - 1].z;

    //fill in vertex normals
    vertices_m1[18 * i + 3] = vertices_m1[18 * i + 9] = vertices_m1[18 * i + 15] =
        myModel.face_norms[i].x;
    vertices_m1[18 * i + 4] = vertices_m1[18 * i + 10] = vertices_m1[18 * i + 16]
    =
        myModel.face_norms[i].y;
    vertices_m1[18 * i + 5] = vertices_m1[18 * i + 11] = vertices_m1[18 * i + 17]
    =
        myModel.face_norms[i].z;
}

//mode 2 vertices
GLfloat* vertices_m2 = new GLfloat[18 * myModel.nf]; // each face has three
    vertices, each vertice has 3-d pos coords + 3-d face normals
for (int i = 0; i < myModel.nf; i++) {
    //fill in vertex positions
    vertices_m2[18 * i] = myModel.vertices[myModel.faces[i].v1 - 1].x;
    vertices_m2[18 * i + 1] = myModel.vertices[myModel.faces[i].v1 - 1].y;
    vertices_m2[18 * i + 2] = myModel.vertices[myModel.faces[i].v1 - 1].z;

    vertices_m2[18 * i + 6] = myModel.vertices[myModel.faces[i].v2 - 1].x;
    vertices_m2[18 * i + 7] = myModel.vertices[myModel.faces[i].v2 - 1].y;
    vertices_m2[18 * i + 8] = myModel.vertices[myModel.faces[i].v2 - 1].z;

    vertices_m2[18 * i + 12] = myModel.vertices[myModel.faces[i].v3 - 1].x;
    vertices_m2[18 * i + 13] = myModel.vertices[myModel.faces[i].v3 - 1].y;
    vertices_m2[18 * i + 14] = myModel.vertices[myModel.faces[i].v3 - 1].z;

    //fill in vertex normals
    vertices_m2[18 * i + 3] = myModel.vertex_norms[myModel.faces[i].v1 - 1].x;
    vertices_m2[18 * i + 4] = myModel.vertex_norms[myModel.faces[i].v1 - 1].y;
    vertices_m2[18 * i + 5] = myModel.vertex_norms[myModel.faces[i].v1 - 1].z;

    vertices_m2[18 * i + 9] = myModel.vertex_norms[myModel.faces[i].v2 - 1].x;
    vertices_m2[18 * i + 10] = myModel.vertex_norms[myModel.faces[i].v2 - 1].y;
    vertices_m2[18 * i + 11] = myModel.vertex_norms[myModel.faces[i].v2 - 1].z;

    vertices_m2[18 * i + 15] = myModel.vertex_norms[myModel.faces[i].v3 - 1].x;
    vertices_m2[18 * i + 16] = myModel.vertex_norms[myModel.faces[i].v3 - 1].y;
    vertices_m2[18 * i + 17] = myModel.vertex_norms[myModel.faces[i].v3 - 1].z;
}
```

4.2 Object and light source movement (Asignment 1)

Object movement is achieved by first setting global parameters for object translational and rotational position, and then updating the positions with keyboard presses according to the preset translational and rotational speed. The object position is then passed to the `glm` translational and rotational functions to be applied to the object.

4.3 Illumination parameters and lighting model pipeline (Tutorial 9_2)

The illumination parameters are stored in global variables, including lighting and material ambient, diffuse and specular strength values, material shininess, and light and material colours. These are then updated based on keyboard presses.

4.3.1 Ambient, diffuse and specular strength parameters

Essentially, the ambient, diffuse and specular strength float parameters (range 0 to 1) are multiplied to the object/light colour vector, and then passed to the material fragment shader, where the actual ambient, diffuse and specular colours are computed based on light position, viewing direction, object fragment position, and object fragment normal. The ambient, diffuse and specular components are then added together to calculate the final colour of the object fragment. The entire process is shown below in the code fragment of the material fragment shader.

```
// 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);

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