Computer Graphics HW2 Report

310551067吳岱容

- **Key Events Explanation:**
  - Key 1: Changing the shader (Gouraud shading and Blinn-Phong shading).
  - Key 2: Changing the current light type (directional light, point light, and spot light).
- **Implementation:**
  - Texture mapping:
    - The mapping tasks have already been done for use, we only need to let the texture reading read the correct picture in the correct file path.

```
std::string woodFilename = "..\\assets\\texture\\wood.jpg";
wood.fromFile(woodFilename);
std::string diceFile[6] = {"..\\assets\\texture\\posx.jpg", "..\\assets\\texture\\negy.jpg",
                           "..\\assets\\texture\\posy.jpg", "..\\assets\\texture\\negy.jpg",
                           "..\\assets\\texture\\posz.jpg", "..\\assets\\texture\\negz.jpg"};
dice.fromFile(diceFile[0], diceFile[1], diceFile[2], diceFile[3], diceFile[4], diceFile[5]);
```

  - Key events switch:
    - Set switch to test the key event and change the variables to do the switching tasks. We control the shader type using the currentShader variable, and the light type use the currentLight variable.

```
case GLFW_KEY_1:
    //shader phong 1 ground 2
  if (currentShader == 1)
    currentShader = 2;
  else
    currentShader = 1;
  break;
case GLFW_KEY_2:
    //light d 0 p 1 s 2
    currentLight =currentLight+ 1;
    if (currentLight >= 3) currentLight -= 3;
    isLightChanged = true;
  break;
```

  - Binding and loading:
    - We bind the uniform block for light first, but we still don’t really bind anything here.

```
shaderPrograms[i].uniformBlockBinding("model", 0);
shaderPrograms[i].uniformBlockBinding("camera", 1);
shaderPrograms[i].uniformBlockBinding("light", 2);
```

- We create lights (3 types) and load all the parameters we need into a uniform block object.

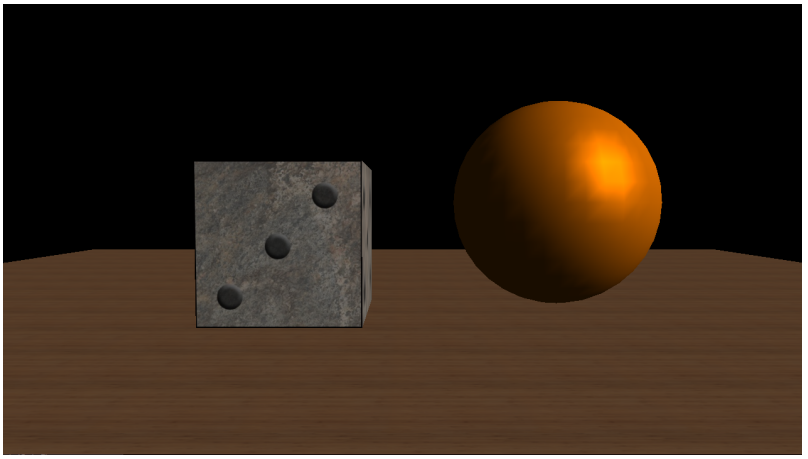
```
for (int i = 0; i < LIGHT_COUNT; ++i) {
  int offset = i * perLightOffset;
  //lights[i]->
  //glm::vec4 front = currentCamera->getFront();
  lightUBO.load(offset, sizeof(glm::mat4), lights[i]->getLightSpaceMatrixPTR());
  lightUBO.load(offset + sizeof(glm::mat4), sizeof(glm::vec4), lights[i]->getLightVectorPTR());
  lightUBO.load(offset + sizeof(glm::mat4) + sizeof(glm::vec4), sizeof(glm::vec4),
                lights[i]->getLightCoefficientsPTR());
}
```

- If the light is changed, we will bind the uniform block to the shader to the current type of light.

```
lightUBO.bindUniformBlockIndex(2, offset, perLightSize);
```

- Shaders and lightings:
  - Gouraud shading

    In Gouraud shading, we compute the color of each vertex, so we need to do the lighting calculation in the vertex shader, and then pass the data down to the fragment shader.
    - Directional light:



```
vec3 lightDir = normalize(vec3(lightVector[0],lightVector[1],lightVector[2]));
vec4 N_tmp = normalMatrix*vec4(Normal_in,1.0);
vec3 N = normalize(vec3(N_tmp[0],N_tmp[1],N_tmp[2]));
vec4 V_tmp = modelMatrix*vec4(Position_in[0],Position_in[1],Position_in[2],1.0);
vec3 V = vec3(V_tmp[0],V_tmp[1],V_tmp[2]);
vec3 E = normalize(vec3(viewPosition[0],viewPosition[1],viewPosition[2])-V);
vec3 R = normalize(-reflect(lightDir,N));

diffuse = kd*max(dot(N,lightDir),0.0);
specular = ks*pow(max(dot(R,E),0.0),8.0);

vertexLight = ambient+(diffuse+specular);
```

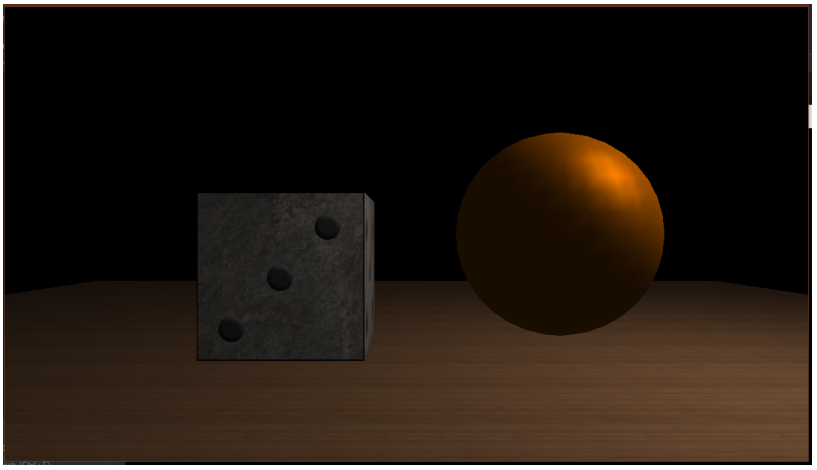
The result is a floating number. In the fragment shader, we can multiply it with the color and we can get the final result.

The code here perform these formula:

$$I = I_{ambient} + I_{diffuse} + I_{specular}$$
$$= k_a L_a + k_d (l \cdot n) L_d + k_s L_s (v \cdot r)^{\alpha}$$

l is the light direction, n is the normal vector, and v is the vector from the position to the perspecter's eye. The reflection vector r we can calculate from the function reflect. The result is stored in vertexLight which will be passed to the fragment shader.
In directional light, lightVector store the light direction.

- Point light:



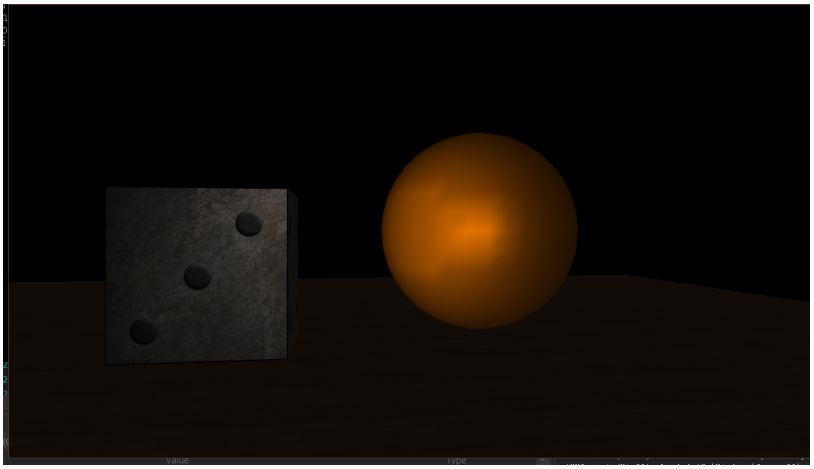
```
vec4 V_tmp = modelMatrix*vec4(Position_in,1.0);
vec3 V = vec3(V_tmp[0],V_tmp[1],V_tmp[2]);
vec3 pos = vec3(lightVector[0],lightVector[1],lightVector[2]);
vec4 N_tmp =normalMatrix*vec4(Normal_in,1.0);
vec3 N = normalize(vec3(N_tmp[0],N_tmp[1],N_tmp[2]));
vec3 L = normalize(pos-V);
vec3 E = normalize(vec3(viewPosition[0],viewPosition[1],viewPosition[2])-V);
vec3 R = normalize(-reflect(L,N));

diffuse =kd*max(dot(N,L),0.0);
specular = ks*pow(max(dot(R,E),0.0),8.0);

float d = distance(pos,V);

float attenuation = 1/(1+0.027*d+0.0028*d*d);

vertexLight = ambient + attenuation*(diffuse+specular);
```

Things are very similar to the directional, but the data stored lightVector isn't the light direction but the light position. Therefore, we need to calculate the light direction on our own. Also, we need to multiply the attenuation to the diffuse and specular variables. Attenuation shows the phenomenon that the light will become weaker as the distance increases. The formula is shown below:

$$1/(a + bd + cd^2)$$

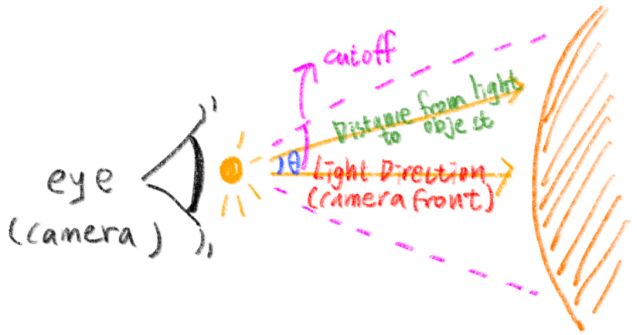
The variables a, b, and c are all given constants, and d means the distance.

- Spot light:



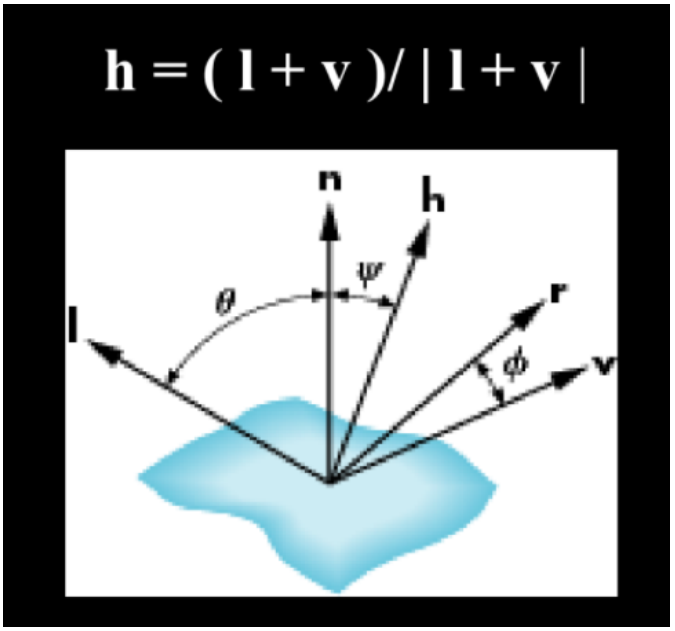
```
vec3 Lpos = vec3(viewPosition[0],viewPosition[1],viewPosition[2]);
vec4 V_tmp = modelMatrix*vec4(Position_in,1.0);
vec3 V = vec3(V_tmp[0],V_tmp[1],V_tmp[2]);
vec4 tmp = vec4(0.0,0.0,-1.0,0.0);
vec3 L=normalize(vec3(lightVector[0],lightVector[1],lightVector[2]));
vec4 N_tmp =normalMatrix*vec4(Normal_in,1.0);
vec3 N = normalize(vec3(N_tmp[0],N_tmp[1],N_tmp[2]));
vec3 E = normalize(vec3(viewPosition[0],viewPosition[1],viewPosition[2])-V);
vec3 R = normalize(-reflect(-L,N));

float d = distance(Lpos,V);

float attenuation = 1/(1+0.014*d+0.0007*d*d);

float specular =0.0;
float diffuse =0.0;

vec3 LtV = normalize(V-Lpos);

if (dot(LtV,L)<coefficients[1]) vertexLight=ambient;
else if (dot(LtV,L)>coefficients[0]){
  diffuse = kd*max(dot(N,-L),0.0);
  specular = ks*pow(max(dot(R,E),0.0),8.0);
  vertexLight = ambient+attenuation*(diffuse+specular);
}
else{
  diffuse = kd*max(dot(N,-L),0.0);
  specular = ks*pow(max(dot(R,E),0.0),8.0);
  vertexLight = ambient+pow(dot(LtV,L),50)*attenuation*(diffuse+specular);
```

In spot light, the light position is our camera position, and the light direction is the camera's front (stored in lightVector).



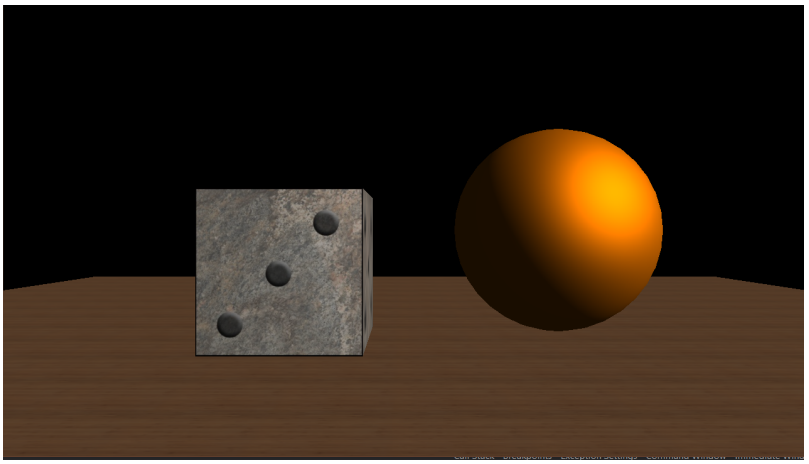
We are given the inner cutoff and outer cutoff. We need to calculate the angle between the light direction and the direction from the light source to the object, and we can use the cutoffs to perform different degrees of attenuation.

- Blinn-Phong shading:
  In Blinn-Phong shading, we calculate per pixel color rather than per vertex. Therefore, we calculate the variables we need in the vertex shader and pass it to the fragment shader.
  The implementation of the lighting in the fragment shader is similar to what we have done in the Gouraud shading's vertex shader, but specular calculation is a little different. If it remains the same, it will be called Phong shading. In Blinn-Phong shading, we replace (v·r) with (n·h), n is the normal vector direction, and h is the halfway direction:

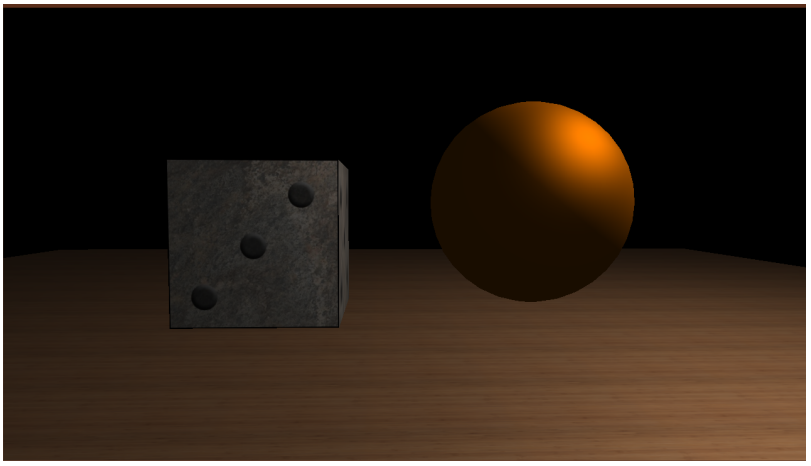



```
vec3 halfway = normalize(-lightDir+ViewDir);
specular = ks*pow(max(dot(normalize(N),halfway),0.0),8.0);
```

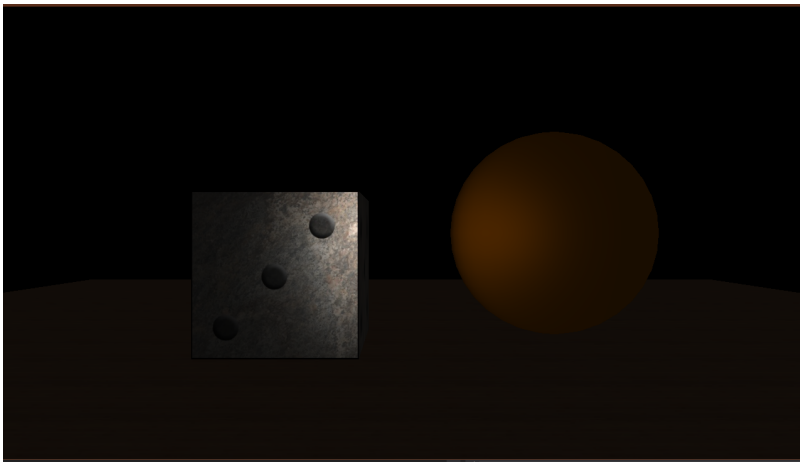
- Directional light:



- Point light:



- Spot light:



- **Problems and Discussion:**
  - Normlization:
    Remember to do the normalization if we want to get the direction. Sometimes, I forget to do the normalization and get strange results.
  - Transformation:
    When using Position_in and Normal_in, remember to transform them into the global presentation, or we will get strange results, too. At first, I

didn’t transform Normal_in and always got the light hit at the wrong position. Also, make sure it is done in the right way. The transformation matrix is mat4, but the input vectors are vec3, so we need to extend the vectors and turn them back to vec3.

In this assignment, we implemented 2 kinds of shader and 3 types of lighting. It is not very simple because it is the first time for me to write a shader, and also it is very difficult to debug. However, through the assignment this time, I learned and got more familiar with shading and lighting by doing them in practice.