计算机图形学作业 6

一、代码解释

(一) Phong Shading

- 环境光照

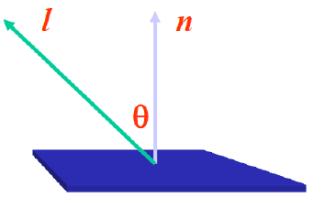
光通常都不是来自于同一个光源，而是来自于我们周围分散的很多光源，即使它们可能并不是那么显而易见。光的一个属性是，它可以向很多方向发散并反弹，从而能够到达不是非常临近的点。所以，光能够在其它表面上反射，对一个物体产生间接影响。考虑到这种情况的算法叫做全局照明(Global Illumination)算法，但是这种算法既开销高昂又极其复杂。环境光照是一个简化的全局照明模型。将一个很小的常量颜色添加到物体片段的最终颜色中能够实现环境光照。

fs_lighting.glsl

```
uniform vec3 lightPos;
uniform vec3 viewPos;
uniform vec3 lightColor;
uniform vec3 objectColor;

uniform float ambientStrength;
uniform float diffuseStrength;
uniform float specularStrength;
uniform int shininess;

void main() {
    // 环境光照
    vec3 ambient = ambientStrength * lightColor;
```

ambientStrength 为环境光照强度，默认为 0.1f，可以通过 GUI 调节。环境光照强度乘以光源颜色得到环境光照分量 ambient。

- 漫反射光照



$$I_{diffuse} = k_d I_{light}(n \cdot l)$$

- k_d: diffuse coefficient
- I_{light}: incoming light intensity
- $I_{diffuse}$: outgoing light intensity (for diffuse reflection)

1. 将法线数据添加到顶点数据

```
float vertices[] = {
    // 位置                 // 法向量
    -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,  0.0f, -1.0f,
     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,  0.0f, -1.0f,
```

```
// 配置立方体VAO，VBO，顶点属性
unsigned int cubeVAO;
glGenVertexArrays(1, &cubeVAO);
glBindVertexArray(cubeVAO);

unsigned int VBO;
glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void *)0); // 位置属性
glEnableVertexAttribArray(0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float))); // 法向量属性
glEnableVertexAttribArray(1);

// 配置灯源VAO，VBO，顶点属性
unsigned int lightVAO;
glGenVertexArrays(1, &lightVAO);
glBindVertexArray(lightVAO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void *)0); // 位置属性
glEnableVertexAttribArray(0);
```

立方体和光源使用相同的顶点数组 vertices。立方体使用位置属性和法向量属性，光源只使用位置属性。

2. 变换顶点位置向量和法向量

因为所有的光照计算都在世界空间进行，因此需要将顶点位置向量和法向量变换到世界空间。

vs_lighting.glsl

```
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;
```

① 顶点位置向量乘以模型矩阵获得变换到世界空间的顶点位置向量 FragPos，并输出到片段着色器。

② 通过 inverse 函数和 transpose 函数生成法线矩阵，将矩阵强制转换为 3×3 矩阵，保证其失去位移属性，从而获得变换到世界空间的法向量，并输出到片段着色器。

3. 计算漫反射光照

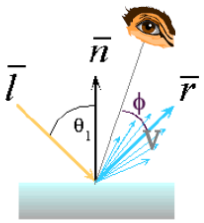
fs_lightning.glsl

```
// 漫反射光照
vec3 norm = normalize(Normal);
vec3 lightDir = normalize(lightPos - FragPos);
float diff = max(dot(norm, lightDir), 0.0);
vec3 diffuse = diffuseStrength * diff * lightColor;
```

① 标准化法向量得到 norm。
② 通过光源位置和片段位置计算光源方向向量，并标准化得到 lightDir。
③ 通过对 norm 向量和 lightDir 向量进行点乘，得到漫反射光照影响 diff。使用 max 函数返回两个参数之中较大的参数，从而保证 diff 不会变成负数。
④ diffuseStrength 为漫反射光照强度，默认为 0.8f，可以通过 GUI 调节。漫反射光照强度乘以 diff 再乘以光源颜色得到漫反射光照分量 diffuse。

- 镜面光照

$$I_{specular} = k_s I_{light} (\bar{v} \cdot \bar{r})^{n_{shiny}}$$



- V is the unit vector towards the viewer
- r is the ideal reflectance direction
- K_s: specular component
- I_{light}: incoming light intensity
- n_{shiny}: purely empirical constant, varies rate of falloff(材质发光常数，值越大，表面越接近镜面，高光面积越小。)

fs_lighting.glsl

```
// 镜面光照
vec3 viewDir = normalize(viewPos - FragPos);
vec3 reflectDir = reflect(-lightDir, norm);
float spec = pow(max(dot(viewDir, reflectDir), 0.0), shininess);
vec3 specular = specularStrength * spec * lightColor;
```

① 通过观察者（摄像机）位置和片段位置计算观察方向向量，并标准化得到 viewDir。
② 通过光源方向向量和法向量计算计算反射方向向量，并标准化得到 reflectDir。
③ 通过对 viewDir 向量和 reflectDir 向量进行点乘，取非负数，再进行幂运算，得到镜面光照影响 spec。观察方向向量和反射向量之间的夹角越小，镜面光照的影响越大。shininess 为反光度。一个物体的反光度越高，反射光的能力越强，散射得越少，高光点就会越小。
④ specularStrength 为镜面光照强度，默认为 0.8f，可以通过 GUI 调节。镜面光照强度乘以 spec 再乘以光源颜色得到镜面光照分量 specular。

- 片段颜色

fs_lighting.glsl

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

环境光照分量加上漫反射光照分量加上镜面光照分量，再用结果乘以物体颜色，得到片段的最终颜色。

(二) Gouraud Shading

在顶点着色器中实现的冯氏光照模型称为 Gouraud Shading。在顶点着色器中实现光照的优势是，相比片段来说，顶点要少得多，因此会更高效，所以（开销大的）光照计算频率会更低。然而，顶点着色器中的最终颜色值是仅仅只是那个顶点的颜色值，片段的颜色值是由插值光照颜色所得来的。结果就是这种光照看起来不会非常真实，除非使用了大量顶点。

vs_Gouraud.glsl

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;

out vec3 LightingColor;

uniform vec3 lightPos;
uniform vec3 viewPos;
uniform vec3 lightColor;

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

uniform float ambientStrength;
uniform float diffuseStrength;
uniform float specularStrength;
uniform int shininess;
```

```
void main() {
    gl_Position = projection * view * model * vec4(aPos, 1.0);

    vec3 Position = vec3(model * vec4(aPos, 1.0));
    vec3 Normal = mat3(transpose(inverse(model))) * aNormal;

    // 环境光照
    vec3 ambient = ambientStrength * lightColor;

    // 漫反射光照
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - Position);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diffuseStrength * diff * lightColor;

    // 镜面光照
    vec3 viewDir = normalize(viewPos - Position);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), shininess);
    vec3 specular = specularStrength * spec * lightColor;

    LightingColor = ambient + diffuse + specular;
}
```

在顶点着色器中计算环境光照分量、漫反射光照分量和镜面光照分量，相加后得到光照分量 LightingColor，并输出到片段着色器。

fs_Gouraud.glsl

```
out vec4 FragColor;

in vec3 LightingColor;

uniform vec3 objectColor;

void main() {
   FragColor = vec4(LightingColor * objectColor, 1.0);
}
```

在片段着色器中通过光照分量乘以物体颜色计算片段的最终颜色。

(三) 添加 GUI

```
float ambientStrength = 0.1f; // 环境光照强度
float diffuseStrength = 0.8f; // 漫反射光照强度
float specularStrength = 0.8f; // 镜面光照强度
int shininess = 5; // 反光度
bool gouraudShading = false;

// ImGui界面
ImGui::Begin("Option");
ImGui::Checkbox("gouraud", &gouraudShading);
ImGui::SliderFloat("ambient", &ambientStrength, 0.0f, 1.0f);
ImGui::SliderFloat("diffuse", &diffuseStrength, 0.0f, 1.0f);
ImGui::SliderFloat("specular", &specularStrength, 0.0f, 1.0f);
ImGui::SliderInt("shininess", &shininess, 1, 8);
ImGui::End();
```

① 切换两种 Shading

```
const char *vertexPath;
const char *fragmentPath;
if (!gouraudShading) {
    vertexPath = "E:\\shader_source\\homework6\\vs_lighting.glsl";
    fragmentPath = "E:\\shader_source\\homework6\\fs_lighting.glsl";
}
else {
    vertexPath = "E:\\shader_source\\homework6\\vs_Gouraud.glsl";
    fragmentPath = "E:\\shader_source\\homework6\\fs_Gouraud.glsl";
}
Shader lightingShader(vertexPath, fragmentPath);
```

② 调节 ambient 因子、diffuse 因子、specular 因子和反光度

```
lightingShader.setFloat("ambientStrength", ambientStrength);
lightingShader.setFloat("specularStrength", specularStrength);
lightingShader.setFloat("diffuseStrength", diffuseStrength);
lightingShader.setInt("shininess", (int)pow(2, shininess));
```

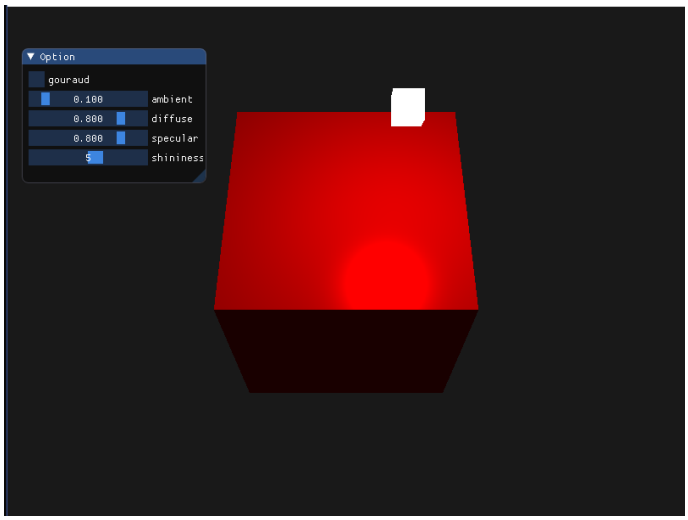
(四) 移动光源

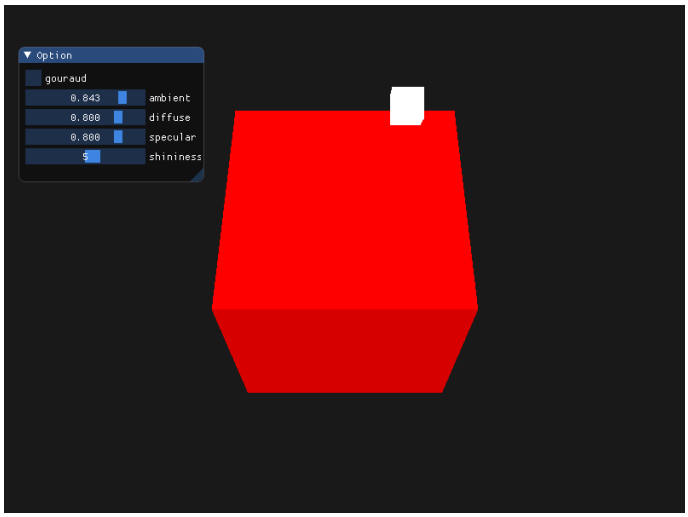
```
// 光源位置
glm::vec3 lightPos(sin(glfwGetTime()), 0.5f, 1.0f);

// 激活光源着色器
lampShader.use();

model = glm::mat4(1.0f);
model = glm::translate(model, lightPos);
model = glm::scale(model, glm::vec3(0.1f));

lampShader.setMat4("model", model);
lampShader.setMat4("view", view);
lampShader.setMat4("projection", projection);

// 渲染光源
glBindVertexArray(lightVAO);
glDrawArrays(GL_TRIANGLES, 0, 36);
```

二、运行结果

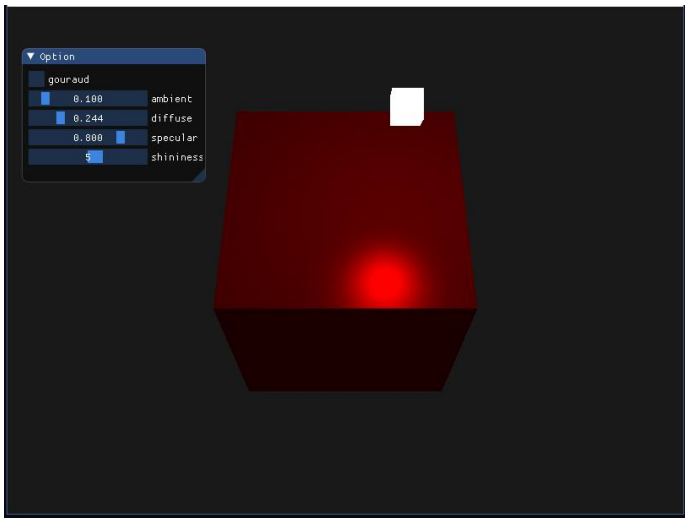
1. Phong Shading

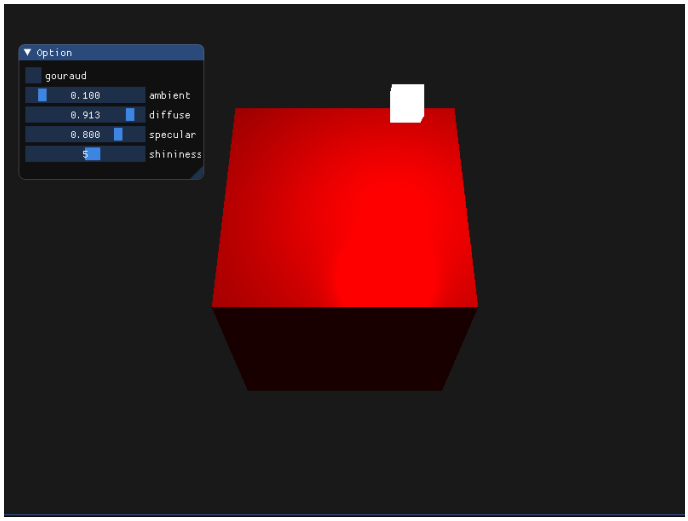
① ambient 因子



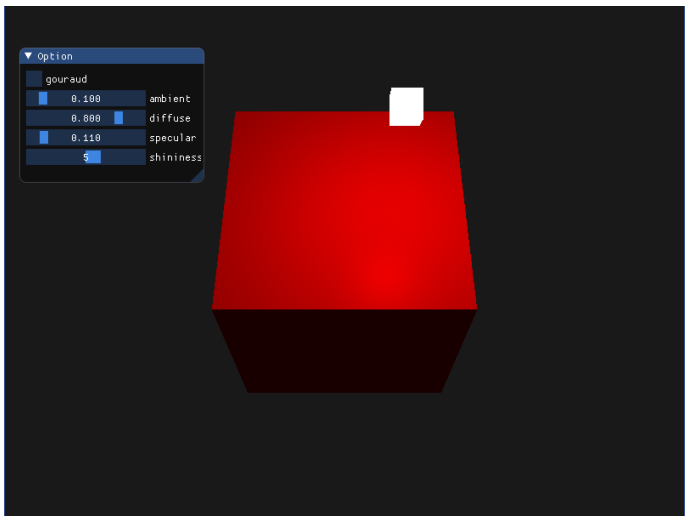


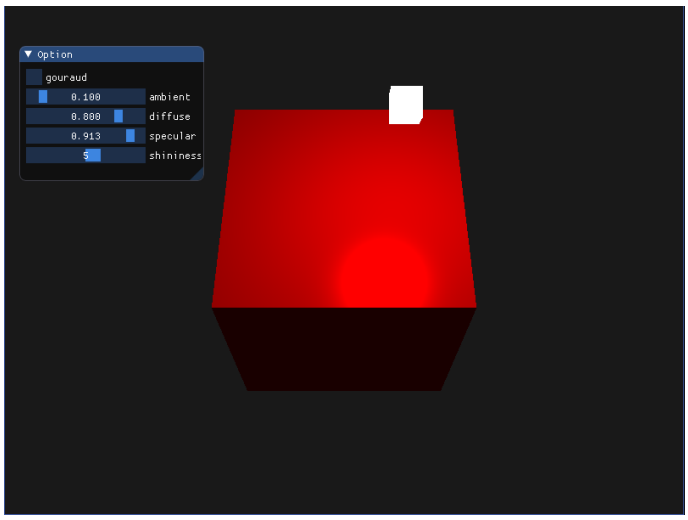
② diffuse 因子



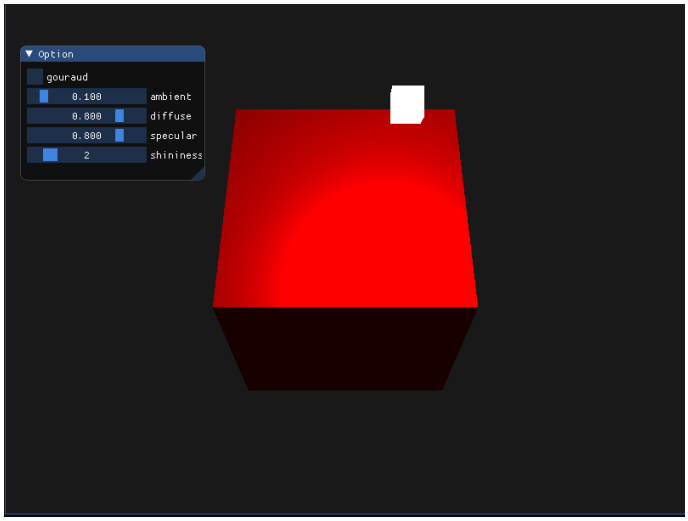


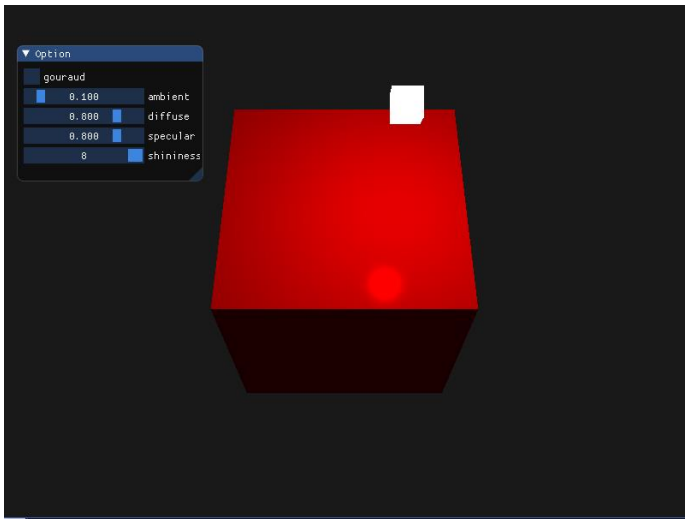
③ specular 因子





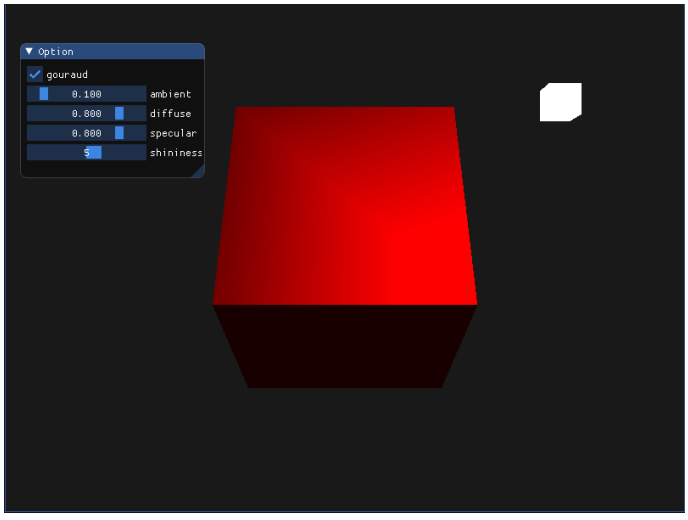
④ 反光度

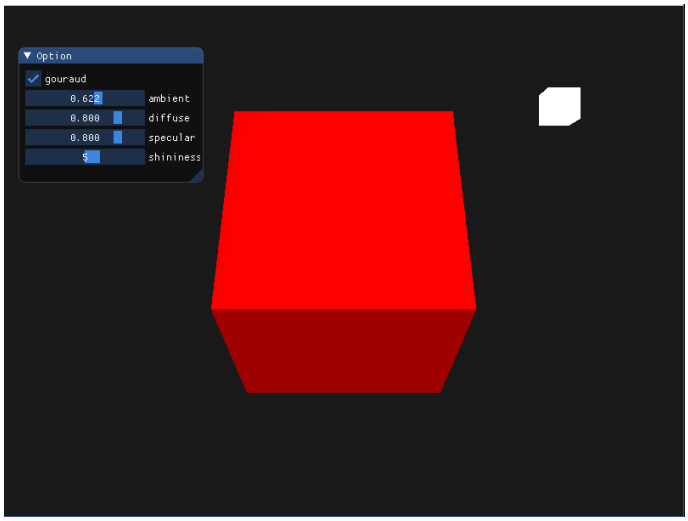




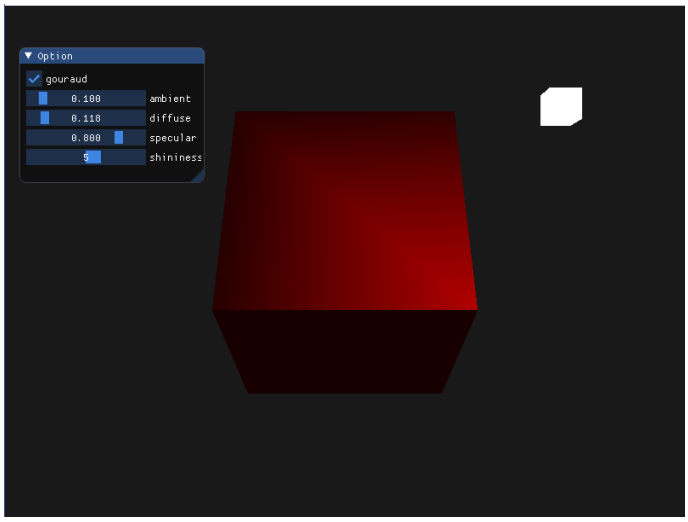
2. Gouraud Shading

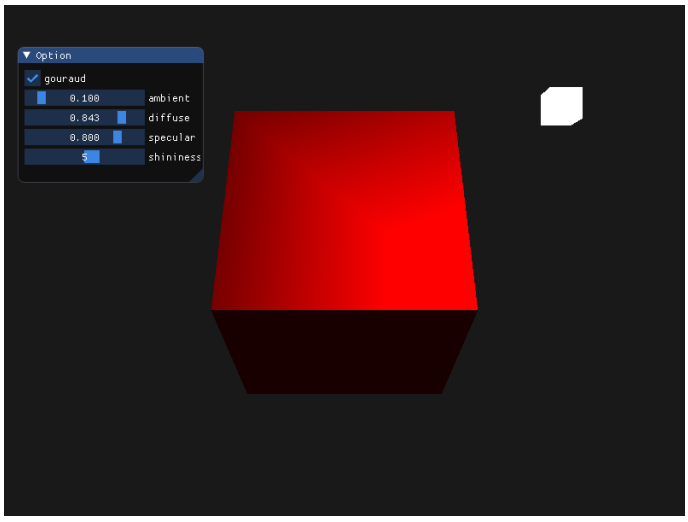
① ambient 因子



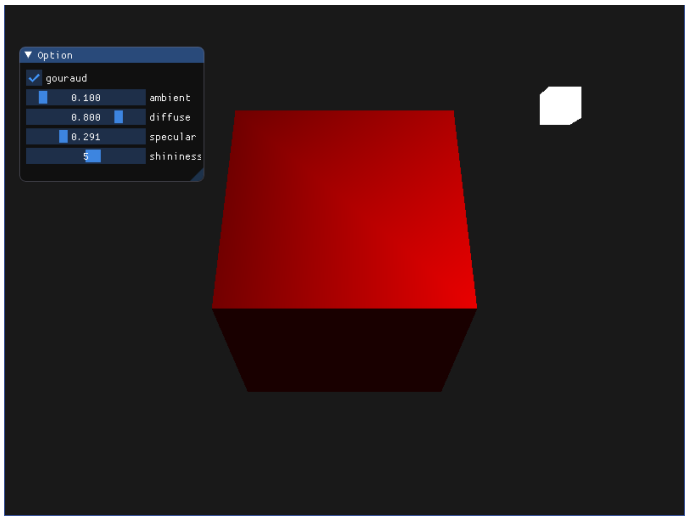


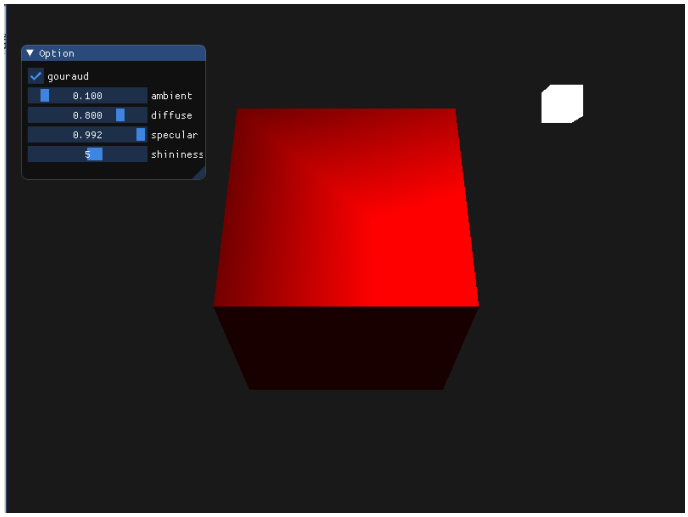
② diffuse 因子



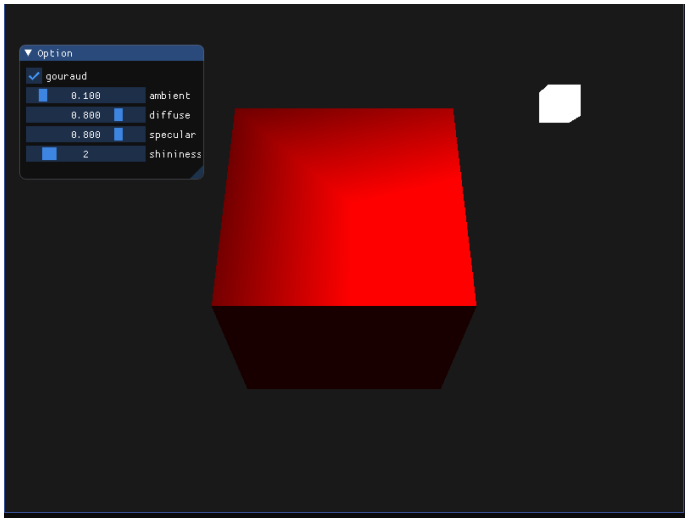


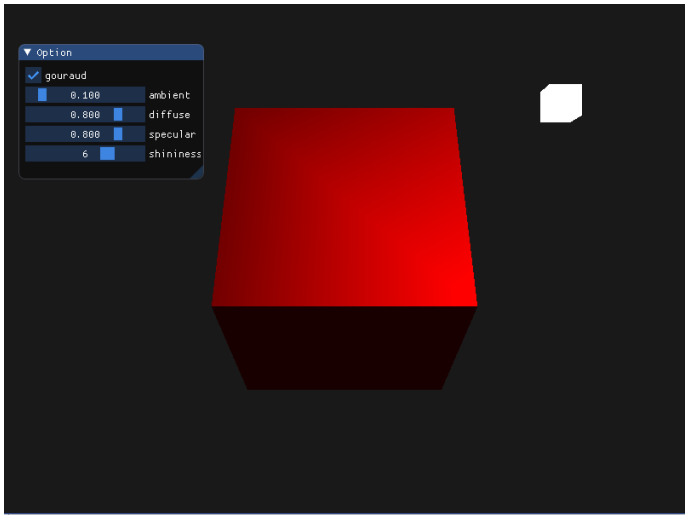
③ specular 因子


Option
gouraud
0.100
ambient
0.800
diffuse
0.291
specular
5
shininess


Option
gouraud
0.100
ambient
0.800
diffuse
0.992
specular
5
shininess

④ 反光度


Option
gouraud
0.100
ambient
0.800
diffuse
0.800
specular
2
shininess



3. 移动光源

见.gif 文件

三、结果分析

Phong Shading 在片段着色器中计算光照，速度相对较慢，效果相对较好；Gouraud Shading 在顶点着色器中计算光照，速度相对较快，效果相对较差。

顶点着色器中的最终颜色值仅仅只是顶点的颜色值，片段的颜色值通过插值得到，因此效果不太真实。由于内插值小于顶点最大值，所以高光部分只在顶点出现，在高光面甚至可以分辨出各个小的图元。