计算机图形学基础

C 类作业实验报告

杨宇菲 计算机系 2020215224
yangyufe20@mails.tsinghua.edu.cn

实验环境：Windows 10, Microsoft Visual Studio 2012

题目一：绘制三角形和四边形并着色，同时旋转和平移

- **实验原理**

以 OpenGL 绘制三角形图元为基础，将包含顶点位置和颜色属性的顶点数组复制到缓冲内存，并设置顶点属性指针，在顶点着色器中输出位置坐标，在片段着色器中输出颜色，将三角形和四边形的绘制转化为多个三角形的绘制。对于图形的旋转和平移，利用 GLM 库中的矩阵和向量运算进行，先后将平移操作和旋转操作作用在 model 矩阵上，再在顶点着色器中用 model 矩阵左乘原位置向量，即可实现同时旋转和平移。

- **实验步骤**

1. 初始化，创建窗口，设置回调函数（按键操作）
2. 分别创建和编译 smooth 着色模式和 flat 着色模式的两个 shader

要改变着色模式，只需将将顶点着色器和片段着色器中的颜色输出和颜色输入变量前加上限定符 smooth/flat 即可。

例如：flat 着色模式

在顶点着色器中，`flat out vec3 ourColor;`

在片段着色器中，`flat in vec3 ourColor;`

3. 设置满足题意的顶点数组并复制到缓冲内存：
   a) 三角形三边长不同：设三顶点分别为(0.0, -0.2)，(0.2, -0.2)，(-0.2, 0.2)，z 轴坐标均为 0.0。
   b) 三角形三个顶点的颜色不同：设三顶点分别为橙色、柠檬黄、黄绿色。
   c) 四边形形状任意：设四顶点分别为(-0.2, -0.2)，(-0.2, 0.2)，(0.2, 0.2)，(0.2, -0.2)，z 轴坐标均为 0.0，构成一个正方形。对于 smooth 着色模式，输入顶点 ABC 和 CDA，分别绘制两个三角形；对于 flat 着色模式，依次输入原点 O 和 ABCDA，按 GL_TRIANGLE_FAN 模式绘制四个三角形。
   d) 四个顶点颜色不同：设四顶点分别为粉色、浅绿色、天蓝色和浅紫色。

这里设置所有坐标均在-1.0 到 1.0 之间，是因为本题中不对坐标进行投影矩阵处理，直接在屏幕空间中设置坐标。以(0.0, 0.0)为中心设置坐标，绕原点旋转即为自身旋转。

4. 设置顶点属性指针

```
// Position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(0);
// Color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
glEnableVertexAttribArray(1);
```

5. 计算 model 矩阵，将矩阵传入顶点着色器

我们令一个三角形和两个四边形在不同的水平位置沿水平方向平移，三角形顺时针旋转，四边形逆时针旋转。在每个循环的渲染指令中，计算图形的水平位置，对初始化为单位阵的

model 矩阵先进行 glm::translate 操作，再进行 glm::rotate 操作，从而在顶点着色器中用 model 左乘位置向量时，可以实现先自身旋转再平移到指定位置。

三角形：

```
model[0] = glm::translate(model[0], glm::vec3(x0[0], 0.6f, 0.0f));
model[0] = glm::rotate(model[0], (GLfloat)glfwGetTime() * glm::radians(-180.0f), glm::vec3(0.0f, 0.0f, 1.0f));
```

四边形：

```
model[1] = glm::translate(model[1], glm::vec3(x0[1], 0.0f, 0.0f));
model[1] = glm::rotate(model[1], (GLfloat)glfwGetTime() * glm::radians(180.0f), glm::vec3(0.0f, 0.0f, 1.0f));
```

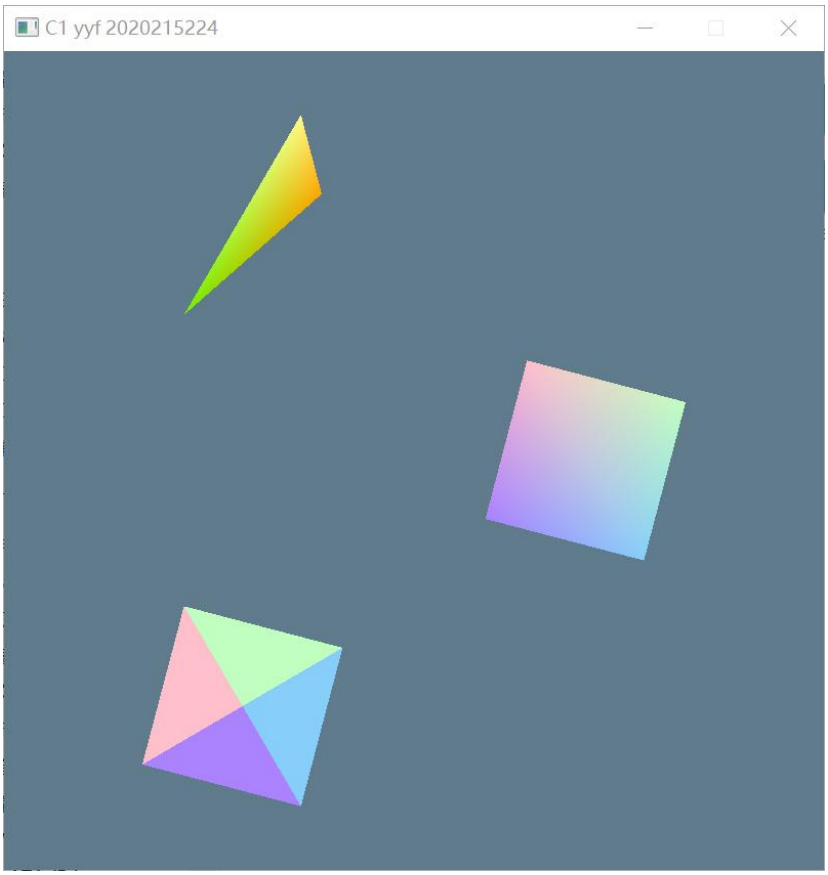
顶点着色器：

```
gl_Position = model * vec4(position, 1.0f);
```

6. 绘制图形

对三角形，使用着色模式为 smooth 的 shader，绘制 3 个顶点；对着色模式为 smooth 的四边形，使用着色模式为 smooth 的 shader，绘制 6 个顶点（2 个三角形）；对着色模式为 flat 的四边形，使用着色模式为 flat 的 shader，用 GL_TRIANGLE_FAN 绘制 6 个顶点（4 个三角形）。

- **实验效果（见录屏）**



交互方式：

ESCAPE 键——关闭窗口

题目二：绘制一个彩色的四棱锥并添加光照效果

- **实验原理**

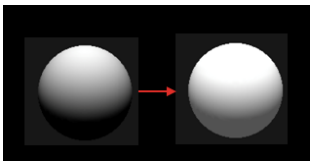
基于 PHONG 光照模型添加光照效果。PHONG 光照模型的主要结构由 3 个元素组成：

环境光照、漫反射光照和镜面光照。

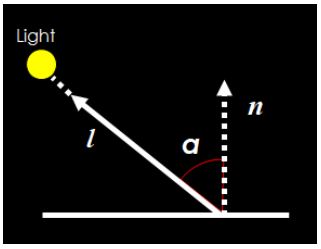
环境光照：即使在黑暗的情况下，世界上也仍然有一些光亮，所以物体永远不会是完全黑暗的。我们使用环境光照来模拟这种情况，即无论如何永远都给物体一些颜色。

漫反射光照：模拟一个发光物对物体的方向性影响，面向光源的一面比其他面会更亮。它是 PHONG 光照模型最显著的组成部分。

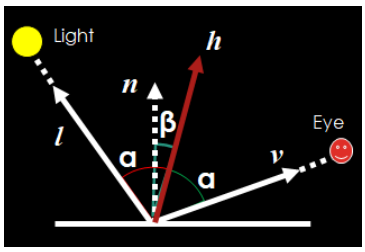
镜面光照：模拟有光泽物体上面出现的亮点。镜面光照的颜色，相比于物体的颜色更倾向于光的颜色。



环境光照



漫反射光照



镜面光照

用公式表示为：

$$L = k_a I max(0, n \cdot l) + k_s I max(0, n \cdot h)^p + k_a I_a$$

漫反射光 + 镜面反射光 + 环境光

- **实验步骤**

1. 初始化，创建窗口，设置回调函数（调整窗口大小、按键操作、鼠标移动、鼠标点击）
2. 创建和编译 shader

在顶点着色器中，计算原位置向量依次左乘 model 矩阵、view 矩阵和 projection 矩阵后得到的屏幕空间位置向量 gl_Position，原位置向量左乘 model 矩阵后得到的世界坐标系下位置向量 FragPos，原法向量左乘正规化 model 矩阵得到的世界坐标系下法向量 Normal。FragPos、Normal 和输入的颜色向量 Color 作为顶点着色器输出，传递给片段着色器。

```
#version 410 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec3 color;

out vec3 Color;
out vec3 Normal;
out vec3 FragPos;

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

void main()
{
    gl_Position = projection * view * model * vec4(position, 1.0f);
    FragPos = vec3(model * vec4(position, 1.0f));
    Normal = mat3(transpose(inverse(model))) * normal;
    Color = color;
}
```

在片段着色器中，创建材质结构体 Material 定义材质属性和光源结构体 Light 定义光源属性和光源位置，并创建 uniform 类型的材质结构体变量 material 和光源结构体变量 light。根据顶点着色器传入的顶点位置和法向量，以及主程序中传入的材质和光源属性，按原理中

给出的公式，分别计算三种光照下的顶点颜色，相加后作为最后颜色输出。

```
#version 410 core

out vec4 FragColor;

in vec3 Color;
in vec3 Normal;
in vec3 FragPos;

struct Material {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};

struct Light {
    vec3 position;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

uniform vec3 viewPos;
uniform Material material;
uniform Light light;
```

```
void main()
{
    //ambient
    vec3 ambient = light.ambient * Color;

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

    //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);
}
```

3. 设置满足题意的顶点数组并复制到缓冲内存：
   a) 四棱锥边长均为 2：设正四棱锥顶点为分别(-1.0, 0.0, -1.0), (1.0, 0.0, -1.0), (1.0, 0.0, 1.0), (-1.0, 0.0, 1.0), (0.0, sqrt(2.0), 0.0)。
   b) 四棱锥各个顶点颜色不同：设顶点颜色分别为红色、绿色、蓝色、紫色、橙色。
   c) 5 个面法向量分别为（未归一化）：(0.0, 1.0, sqrt(2.0)), (sqrt(2.0), 1.0, 0.0), (0.0, 1.0, -sqrt(2.0)), (-sqrt(2.0), 1.0, 0.0), (0.0, -1.0, 0.0)。

   每个面的三个顶点结合各自法向量作为一组输入，共 6 组（对于底面的正方形，分为两

个三角形进行输入）。

4. 设置顶点属性指针

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 9 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(0);

glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 9 * sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
glEnableVertexAttribArray(1);

glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, 9 * sizeof(GLfloat), (GLvoid*)(6 * sizeof(GLfloat)));
glEnableVertexAttribArray(2);
```

5. 计算 model、view 和 projection 矩阵，将矩阵和当前相机位置传入顶点着色器

为满足题目四棱锥中心在(1.0, 2.0, 3.0)的要求，先对 model 单位矩阵进行 glm::translate 操作，若开启旋转功能，再对其进行 glm::rotate 操作。

6. 设置材质属性，传入片段着色器

将 ambient 和 diffuse 元素设置成物体原本颜色（在片段着色器中直接将 material.ambient 和 material.diffuse 替换为 Color）。由于不希望 specular 元素对指定物体产生过于强烈的影响，设置 specular 元素为中等亮度颜色(0.5, 0.5, 0.5)。发光值 shininess 与全局变量 matShine 相等，可通过按键控制其大小。

```
//Material
GLint matSpecularLoc = glGetUniformLocation(ourShader.Program, "material.specular");
GLint matShineLoc = glGetUniformLocation(ourShader.Program, "material.shininess");

glUniform3f(matSpecularLoc, 0.5f, 0.5f, 0.5f);
glUniform1f(matShineLoc, matShine);
```

7. 设置光源属性，传入片段着色器

如果不开启变色，设置光源颜色 lightColor 为白光(1.0, 1.0, 1.0)，否则其颜色随时间变化。光源的 ambient 元素颜色亮度比较低，设为 0.3 * lightColor，diffuse 元素颜色暗一点更自然，设为 0.5 * lightColor，镜面反射光颜色设为一个明亮的白色(1.0, 1.0, 1.0)。设置光源位置在四棱锥坐标系中为(1.0, 4.0, 4.0)。

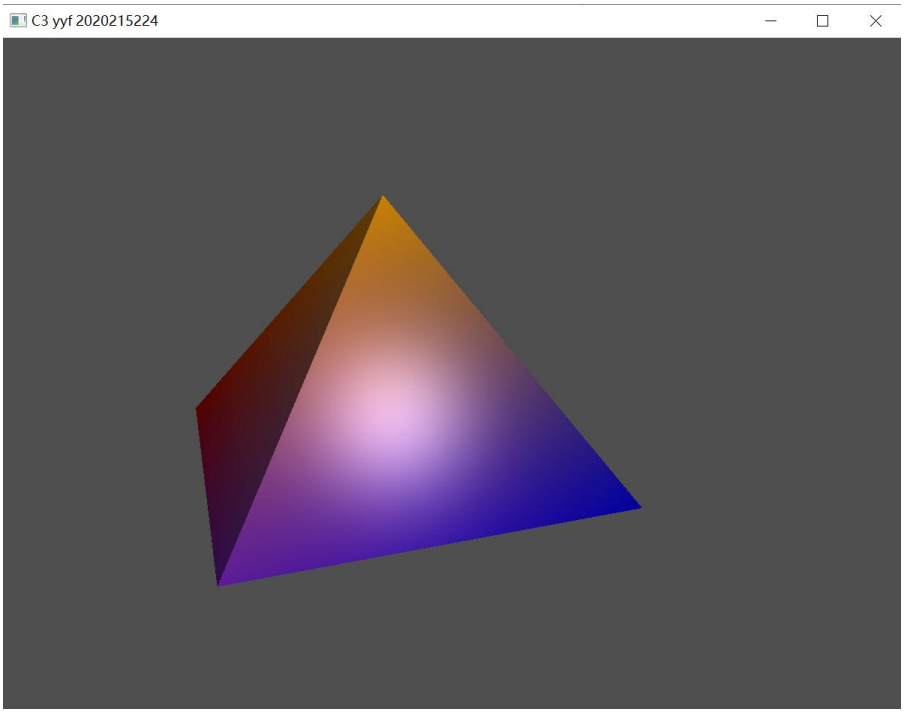
```
//Light
GLint lightPosLoc = glGetUniformLocation(ourShader.Program, "light.position");
GLint lightAmbientLoc = glGetUniformLocation(ourShader.Program, "light.ambient");
GLint lightDiffuseLoc = glGetUniformLocation(ourShader.Program, "light.diffuse");
GLint lightSpecularLoc = glGetUniformLocation(ourShader.Program, "light.specular");

glm::vec3 lightColor;
if (isChanging)
{
    changingTime += deltaTime;
    lightColor.x = sin(changingTime * 2.0f);
    lightColor.y = sin(changingTime * 0.7f);
    lightColor.z = sin(changingTime * 1.3f);
}
else
    lightColor = glm::vec3(1.0f);
glm::vec3 diffuseColor = lightColor * glm::vec3(0.5f);
glm::vec3 ambientColor = diffuseColor * glm::vec3(0.6f);
glUniform3f(lightPosLoc, 1.0f + 1.0f, 4.0f + 2.0f, 4.0f + 3.0f);
glUniform3f(lightAmbientLoc, ambientColor.x, ambientColor.y, ambientColor.z);
glUniform3f(lightDiffuseLoc, diffuseColor.x, diffuseColor.y, diffuseColor.z);
glUniform3f(lightSpecularLoc, 1.0f, 1.0f, 1.0f);
```

8. 回调函数

除窗口大小调整、相机位置和视野变化外，设置按“+”/“-”增大或减小 matShine 值，按鼠标左键开启或关闭模型绕 y 轴旋转，按鼠标右键开启或关闭光源颜色变化。

- **实验效果（见录屏）**



可以看到，材质发光值越高，反射光的能力越强，散射得越少，导致模型上的高光点越小。而材质发光值约小，高光点越大，发光值约为 1 时整个面对光源的平面反射高光。

交互方式:

ESCAPE 键——关闭窗口

W/A/S/D 键——相机位置上（世界坐标系）/下（世界坐标系）/左（自身坐标系）/右（自身坐标系）移动

鼠标滚轮——视野缩放

鼠标移动——改变相机俯仰角和偏航角

+/-键——调整材质发光值大小，最大值稍大于 1024，最小值稍小于 1

点击鼠标左键——开启/关闭模型绕 y 轴旋转

点击鼠标右键——开启/关闭光源颜色变化

题目三：对正方体加载纹理

- **实验原理**

通过纹理映射实现对模型加载纹理。为了把纹理映射到三维物体上（由一个个三角形面片组成），需要指明三角形每个顶点从纹理图像的哪个位置采集颜色，我们通过给每个顶点关联纹理坐标实现。对于三角形的其他部分，通过顶点纹理坐标的插值得到纹理坐标，进而获取颜色，插值方式/纹理过滤方式有邻近过滤、线性过滤和多级渐远纹理过滤。

- **实验步骤**

1. 初始化，创建窗口，设置回调函数（调整窗口大小、按键操作、鼠标移动、鼠标点击）
2. 创建和编译 shader

在顶点着色器中，计算原位置向量依次左乘 model 矩阵、view 矩阵和 projection 矩阵后得到的屏幕空间位置向量 gl_Position。同时，有一个纹理坐标的输入变量以接收顶点纹理数据，并有一个纹理坐标的输出变量以传给片段着色器。由于用 SOIL 读入的图片数据 y 轴 0 坐标在图片的顶部，而 OpenGL 的纹理空间中定义 y 轴 0 坐标在图片底部，我们需要在顶点着色器中对 y 坐标进行翻转。

```
#version 410 core

layout (location = 0) in vec3 position;
layout (location = 1) in vec2 texCoord;

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

out vec2 TexCoord;

void main()
{
    gl_Position = projection * view * model * vec4(position, 1.0f);
    TexCoord = vec2(texCoord.x, 1.0f - texCoord.y);
}
```

在片段着色中，除了有一个纹理坐标的输入变量以接收顶点着色器的输出，还声明一个 uniform 类型的 sampler2D 采样器，通过将纹理单元赋值给这个 uniform 变量，用来访问生成的纹理对象。然后，使用 GLSL 内建的 texture 函数，根据纹理对象中保存的纹理环绕和过滤方式来采样纹理的颜色。

```
#version 410 core

in vec2 TexCoord;

out vec4 color;

uniform sampler2D ourTexture;

void main()
{
    color = texture(ourTexture, TexCoord);
}
```

3. 设置立方体顶点数组和索引数组，复制到缓冲内存，设置顶点属性指针

立方体顶点分为 6 组，每组为一个面的 4 个顶点，索引数组共 3×2×6 个元素，依次绘制每个面的两个三角形。顶点数组包含顶点的位置坐标和纹理坐标。

```
// Position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(0);
// TexCoord attribute
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
glEnableVertexAttribArray(1);
```

4. 创建纹理对象数组，分别设置纹理环绕和过滤方式，读取纹理图片，生成纹理

要求每个面纹理不同，并使用三种纹理过滤方式加载纹理，因此纹理对象数组共 6×3=18 个元素。

5. 计算 model、view 和 projection 矩阵，将矩阵传入顶点着色器

若开启旋转功能，需对 model 矩阵进行 glm::rotate 操作：

```
if (isRotate)
    model = glm::rotate(model, deltaTime * glm::radians(50.0f), glm::vec3(0.5f, 1.0f, 0.0f));
```

6. 绑定纹理并绘制

对正方形的每个面，先加载当前选定的纹理过滤方式和指定面纹理对应的纹理对象，用 glDrawElements 函数绘制对应的两个三角形。

7. 回调函数

除窗口大小调整、相机位置和视野变化外，设置按“1”/“2”/“3”切换纹理过滤方式，按鼠标左键开启或关闭模型旋转。

- **实验效果（见录屏）**

GL_NEAREST 产生了颗粒状的图案，能够清晰看到组成纹理的像素。而 GL_LINEAR 和 GL_LINEAR_MIPMAP_LINEAR 差别不大，产生的图案更平滑，很难看出单个的纹理像素。GL_LINEAR 和 GL_LINEAR_MIPMAP_LINEAR 可以产生更真实的输出。

交互方式：
ESCAPE 键——关闭窗口
W/A/S/D 键——相机位置上（世界坐标系）/下（世界坐标系）/左（自身坐标系）/右（自身坐标系）移动
鼠标滚轮——视野缩放
鼠标移动——改变相机俯仰角和偏航角
1 键——纹理过滤方式（放大和缩小）为 GL_LINEAR
2 键——纹理过滤方式（放大和缩小）为 GL_LINEAR_MIPMAP_LINEAR
3 键——纹理过滤方式（放大和缩小）为 GL_NEAREST
点击鼠标左键——开启/关闭模型旋转