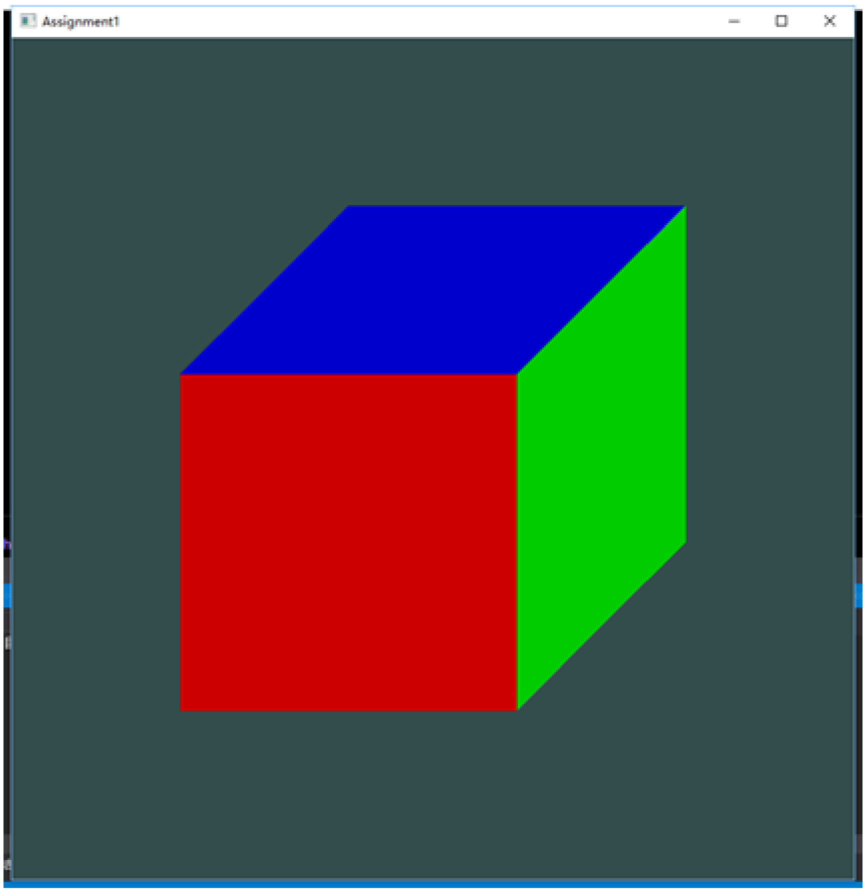
计算机图形学实验报告

- 姓名 张睦婕
- 学号 71117133

作业一报告

一 、实验内容

使用OPENGL绘制如下形状



- 说明：

1. 在完成作业之前，需要先了解学习OPENGL绘制简单图形的步骤，并选择IDE和进行环境配置，参考教程 请仔细阅读上述教程，进行IDE和环境的配置，之后根据教程提供的示例代码尝试绘制一个三角形，在练习绘制三角形之后，完成绘制上述图形的作业。
2. 该形状是简单的平面图形，通过绘制若干不同颜色的三角形实现即可，使用OPENGL绘制三角形见于：https://learnopengl-cn.github.io/01%20Getting%20started/04%20Hello%20Triangle/

二、环境及资源

- 编程语言：c++
- IDE：Visual Studio 2017
- OpenGL
- GLFW GLFW是一个专门针对OpenGL的C语言库，它提供了一些渲染物体所需的最低限度的接口。它允许用户创建OpenGL上下文，定义窗口参数以及处理用户输入，这正是我们需要的。
- CMake CMake是一个工程文件生成工具。用户可以使用预定义好的CMake脚本，根据自己的选择（像是Visual Studio, Code::Blocks, Eclipse）生成不同IDE的工程文件。
- GLAD

因为OpenGL只是一个标准/规范，具体的实现是由驱动开发商针对特定显卡实现的。由于OpenGL驱动版本众多，它大多数函数的位置都无法在编译时确定下来，需要在运行时查询。所以任务就落在了开发者身上，开发者需要在运行时获取函数地址并将其保存在一个函数指针中供以后使用。取得地址的方法因平台而异，在Windows上会是类似这样：

```
// 定义函数原型
typedef void (*GL_GENBUFFERS) (GLsizei,
GLuint*);
// 找到正确的函数并赋值给函数指针
GL_GENBUFFERS glGenBuffers  =
(GL_GENBUFFERS)wglGetProcAddress("glGenBuffers")
;
// 现在函数可以被正常调用了
GLuint buffer;
glGenBuffers(1, &buffer);
```

可以看到代码非常复杂，而且很繁琐，我们需要对每个可能使用的函数都要重复这个过程。幸运的是，有些库能简化此过程，其中**GLAD**是目前最新，也是最流行的库。

三、具体实现及设计

1. 建立一个新的c++项目
2. 把GLFW库链接（LINK）进工程
3. 添加OpenGL库，将opengl32.lib添加进连接器设置 因为opengl32.lib已经包含在Microsoft SDK里了
4. 在源文件中包含GLFW的头文件和glad的头文件

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
```

5. 在main中实例化窗口

```
int main()
{
  glfwInit();
  glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
  glfwWindowHint(GLFW_OPENGL_PROFILE,
GLFW_OPENGL_CORE_PROFILE);
  //glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT,
GL_TRUE);

  return 0;
}
```

6. 初始化glad

```
  if
(!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
  {
    std::cout << "Failed to initialize GLAD" <<
std::endl;
    return -1;
  }
```

7. 调用glViewport函数来设置窗口的维度(Dimension)
8. 对窗口注册一个回调函数(Callback Function)，它会在每次窗口大小被调整的时候被调用。
9. 在程序中添加一个while循环，我们可以把它称之为渲染循环(Render Loop)，它能在我们让GLFW退出前一直保持运行。

   下面几行的代码就实现了一个简单的渲染循环：

   ```
   while(!glfwWindowShouldClose(window))
   {
       glfwSwapBuffers(window);
       glfwPollEvents();
   }
   ```

10. 当渲染循环结束后我们需要正确释放/删除之前的分配的所有资源。

    我们可以在main函数的最后调用glfwTerminate函数来完成。

    ```
    glfwTerminate();
    return 0;
    ```

11. 顶点输入

    ```
    float frontVertices[] = {
        -0.5f,  0.0f,   0.0f,
        0.0f,   0.0f,   0.0f,
        0.0f,   -0.5f,  0.0f,
        -0.5f,  -0.5f,  0.0f
    };
    float topVertices[] = {
        -0.3f,  0.3f,   0.0f,
        0.2f,   0.3f,   0.0f,
        0.0f,   0.0f,   0.0f,
        -0.5f,  0.0f,   0.0f
    };
    float rightVertices[] = {
        0.0f,   0.0f,   0.0f,
        0.2f,   0.3f,   0.0f,
    ```

```
    0.2f,   -0.2f,  0.0f,
    0.0f,   -0.5f,  0.0f,


};
```

12. 顶点着色器

```
const char *vertexShaderSource = "#version 330
core\n"
"layout (location = 0) in vec3 aPos;\n"
"void main()\n"
"{\n"
"   gl_Position = vec4(aPos.x, aPos.y, aPos.z,
1.0);\n"
"}\0";
const char *color[3] = { "#version 330 core\n"
"out vec4 FragColor;\n"
"void main()\n"
"{\n"
"   FragColor = vec4(1.0f, 0.0f, 0.0f, 1.0f);\n"
"}\n\0",
"#version 330 core\n"
"out vec4 FragColor;\n"
"void main()\n"
"{\n"
"   FragColor = vec4(0.0f, 0.0f, 1.0f, 1.0f);\n"
"}\n\0",
"#version 330 core\n"
"out vec4 FragColor;\n"
"void main()\n"
"{\n"
"   FragColor = vec4(0.0f, 1.0f, 0.0f, 1.0f);\n"
"}\n\0" };
```

13. 片段着色器

```
unsigned int VBO[3],VAO[3],EBO[3];
    glGenBuffers(3, VBO);
    glGenVertexArrays(3, VAO);
    glGenBuffers(3, EBO);

    //front
    glBindBuffer(GL_ARRAY_BUFFER, VBO[0]);
    glBindVertexArray(VAO[0]);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
EBO[0]);

```

```
    glBufferData(GL_ARRAY_BUFFER,
sizeof(frontVertices), frontVertices,
GL_STATIC_DRAW);
    glVertexAttribPointer(0, 3, GL_FLOAT,
GL_FALSE, 3 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER,
sizeof(indices), indices, GL_STATIC_DRAW);

    //top
    glBindBuffer(GL_ARRAY_BUFFER, VBO[1]);
    glBindVertexArray(VAO[1]);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
EBO[1]);

    glBufferData(GL_ARRAY_BUFFER,
sizeof(topVertices), topVertices,
GL_STATIC_DRAW);
    glVertexAttribPointer(0, 3, GL_FLOAT,
GL_FALSE, 3 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER,
sizeof(indices), indices, GL_STATIC_DRAW);

    //right
    glBindBuffer(GL_ARRAY_BUFFER, VBO[2]);
    glBindVertexArray(VAO[2]);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
EBO[2]);

    glBufferData(GL_ARRAY_BUFFER,
sizeof(rightVertices), rightVertices,
GL_STATIC_DRAW);
    glVertexAttribPointer(0, 3, GL_FLOAT,
GL_FALSE, 3 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER,
sizeof(indices), indices, GL_STATIC_DRAW);

    // vertex Shader
    unsigned int vertexShader[3];

    // fragment shader
    unsigned int fragmentShader[3];
```

14. 索引缓冲对象

```
unsigned int indices[] = {
0, 1, 3,
1, 2, 3
};
```

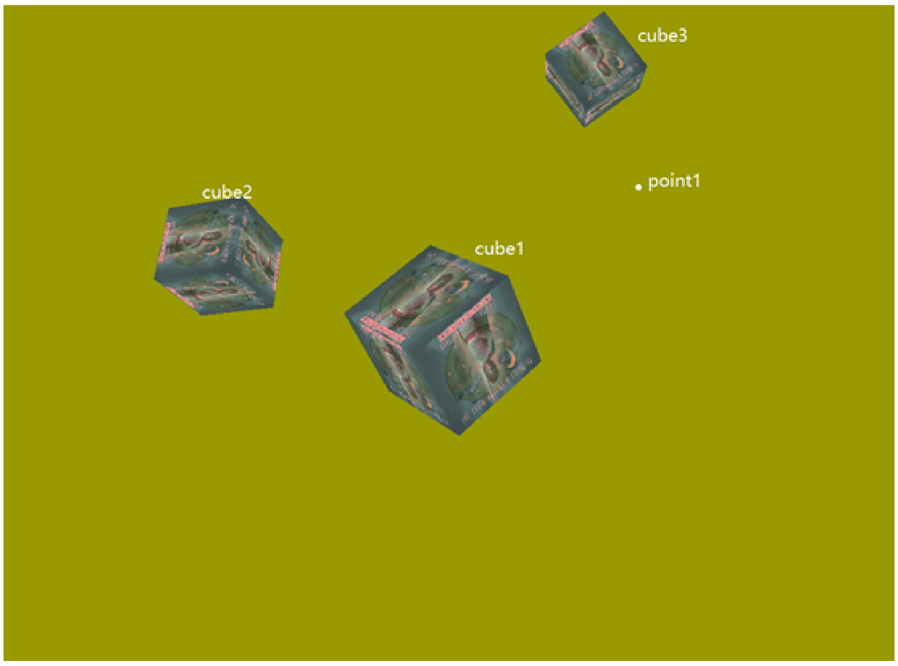
四 、实验结果及总结

最终得到一个看起来像是立方体的平面图形，对OpenGL更加好奇，如何画出一个真的的立方体呢？

实验二实验报告

一、 实验内容

使用OPENGL绘制如下三维图形及其简单动作



说明：

1. 在第一次作业中，练习了使用vao vbo ebo绘制平面三角形，在此基础上，第二次作业将进行三维立方体的绘制以及平移、旋转、缩放变换的练习。
2. 请继续向后阅读教程中着色器、纹理贴图、坐标变换三个小章节的内容，并仔细思考和练习坐标变换，掌握其原理和使用方法。教程
3. 在上述1 2的基础上，完成第二次作业，如mp4视频中所示。

① 绘制三个处于同一平面但大小不同的立方体，并且自选图片对其 进行纹理贴图

② cube1位于中心原点(0.0f，0.0f，0.0f)，并绕其斜对角线自转；

cube2绕任意轴自转，并以远离中心原点一定的距离绕z轴公转。

cube3绕任意轴自转，并以远离上图中point1(0.5f，0.5f，0.0f) 一定的距离绕该点处的y-z平面法向量旋转。

4. 在练习和完成作业的过程中，思考纹理贴图与插值的过程，思考着色器中变量的定义和数据的传入传出，思考变换矩阵是如何作用于某一点的坐标向量使其产生旋转或位移或缩放的，熟悉并掌握使用glm库构造变换矩阵的技巧。

二、环境及资源

- 编程语言：c++
- IDE：Visual Studio 2017
- OpenGL
- GLFW GLFW是一个专门针对OpenGL的C语言库，它提供了一些渲染物体所需的最低限度的接口。它允许用户创建OpenGL上下文，定义窗口参数以及处理用户输入，这正是我们需要的。
- CMake CMake是一个工程文件生成工具。用户可以使用预定义好的CMake脚本，根据自己的选择（像是Visual Studio, Code::Blocks, Eclipse）生成不同IDE的工程文件。
- GLAD 因为OpenGL只是一个标准/规范，具体的实现是由驱动开发商针对特定显卡实现的。由于OpenGL驱动版本众多，它大多数函数的位置都无法在编译时确定下来，需要在运行时查询。但是代码非常复杂，而且很繁琐，我们需要对每个可能使用的函数都要重复这个过程。幸运的是，有些库能简化此过程，其中**GLAD**是目前最新，也是最流行的库。
- stb_image.h

  `stb_image.h` 是Sean Barrett的一个非常流行的单头文件图像加载库，它能够加载大部分流行的文件格式，并且能够很简单得整合到工程之中。
- GLM OpenGL没有自带任何的矩阵和向量知识，所以我们必须定义自己的数学类和函数。有个易于使用，专门为OpenGL量身定做的数学库，那就是GLM。

三、具体设计及实现

1. 绘制三维立方体
2. 加载与创建纹理
3. 进行坐标变换
4. 设计3D视角

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
#include "Shader.h"
```

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include <iostream>
#include<string.h>
using namespace std;

void framebuffer_size_callback(GLFWwindow* window, int
width, int height);
void processInput(GLFWwindow *window);

// settings
const unsigned int SCR_WIDTH = 800;
const unsigned int SCR_HEIGHT = 600;

int main()
{
    // glfw: initialize and configure
    // ------------------------------
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE,
GLFW_OPENGL_CORE_PROFILE);

#ifdef __APPLE__
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
// uncomment this statement to fix compilation on OS X
#endif

    // glfw window creation
    // --------------------
    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH,
SCR_HEIGHT, "Homework2", NULL, NULL);
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" <<
std::endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window,
framebuffer_size_callback);

    // glad: load all OpenGL function pointers
    // ---------------------------------------
    if
(!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
        std::cout << "Failed to initialize GLAD" <<
std::endl;
        return -1;
    }
    // configure global opengl state
   // -----------------------------
    glEnable(GL_DEPTH_TEST);

    // build and compile our shader program
    // ------------------------------------
    Shader ourShader("3.3.shader.vs", "3.3.shader.fs");
// you can name your shader files however you like

    // set up vertex data (and buffer(s)) and configure
vertex attributes
    // ---------------------------------------------------
-----------------
    float vertices[6][20] = {
        // positions          // texture coords
        {
        -0.5f, -0.5f, -0.5f,  0.0f, 0.0f,
         0.5f, -0.5f, -0.5f,  1.0f, 0.0f,
         0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
        -0.5f,  0.5f, -0.5f,  0.0f, 1.0f
        },
        {
        -0.5f, -0.5f,  0.5f,  0.0f, 0.0f,
         0.5f, -0.5f,  0.5f,  1.0f, 0.0f,
         0.5f,  0.5f,  0.5f,  1.0f, 1.0f,
        -0.5f,  0.5f,  0.5f,  0.0f, 1.0f
        },
        {
        -0.5f,  0.5f,  0.5f,  1.0f, 0.0f,
        -0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
        -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
        -0.5f, -0.5f,  0.5f,  0.0f, 0.0f
        },
        {
        0.5f,  0.5f,  0.5f,  1.0f, 0.0f,
         0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
         0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
         0.5f, -0.5f,  0.5f,  0.0f, 0.0f
        },
        {
        -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
         0.5f, -0.5f, -0.5f,  1.0f, 1.0f,
         0.5f, -0.5f,  0.5f,  1.0f, 0.0f,
        -0.5f, -0.5f,  0.5f,  0.0f, 0.0f
        },
        {
         -0.5f,  0.5f, -0.5f,  0.0f, 1.0f,
         0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
         0.5f,  0.5f,  0.5f,  1.0f, 0.0f,
        -0.5f,  0.5f,  0.5f,  0.0f, 0.0f
        }

    };
    unsigned int indices[] = {
        0, 1, 2, // first triangle
        2, 3 ,0 // second triangle
    };


    unsigned int VBO[6], VAO[6], EBO[6];
    glGenVertexArrays(6, VAO);
    glGenBuffers(6, VBO);
    glGenBuffers(6, EBO);
    // texture 1
    unsigned int texture[6];
    glGenTextures(6, texture);


    for (int i = 0; i < 6; i++)
    {
        glBindVertexArray(VAO[i]);

        glBindBuffer(GL_ARRAY_BUFFER, VBO[i]);
        glBufferData(GL_ARRAY_BUFFER,
sizeof(vertices[i]), vertices[i], GL_STATIC_DRAW);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO[i]);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER,
sizeof(indices), indices, GL_STATIC_DRAW);

        // position attribute
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
5 * sizeof(float), (void*)0);
        glEnableVertexAttribArray(0);
        // texture coord attribute
        glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE,
5 * sizeof(float), (void*)(3 * sizeof(float)));
        glEnableVertexAttribArray(1);
        glBindTexture(GL_TEXTURE_2D, texture[i]);
        // set the texture wrapping parameters
        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_T, GL_REPEAT);
        // set texture filtering parameters
        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        // load image, create texture and generate
mipmaps
        int width, height, nrChannels;
        stbi_set_flip_vertically_on_load(true); // tell
stb_image.h to flip loaded texture's on the y-axis.
        string filename = "head-"+to_string(i)+".png";
        unsigned char *data = stbi_load(filename.data(),
&width, &height, &nrChannels, 0);
        if (data)
        {
            glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB,
width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
            glGenerateMipmap(GL_TEXTURE_2D);
        }
        else
        {
            std::cout << "Failed to load texture" <<
std::endl;
        }
        stbi_image_free(data);

        // tell opengl for each sampler to which texture
unit it belongs to (only has to be done once)
        // ------------------------------------------------
-----------------------------------------------
        ourShader.use();
        ourShader.setInt("texture1", 0);
    }

    // render loop
    // -----------
    while (!glfwWindowShouldClose(window))
    {
        // input
        // -----
        processInput(window);
        // render
        // ------
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);

        ourShader.use();

        // create transformations
        glm::mat4 view;
        glm::mat4 projection;

        view = glm::translate(view, glm::vec3(0.0f,
0.0f, -8.0f));
        projection =
glm::perspective(glm::radians(45.0f), (float)SCR_WIDTH /
(float)SCR_HEIGHT, 0.1f, 100.0f);
        ourShader.setMat4("projection", projection);
        ourShader.setMat4("view", view);
        for (int j = 0; j < 3; j++) {
            glm::mat4 model;
            switch (j)
            {
            case 1:
            {
                model = glm::rotate(model,
(float)glfwGetTime(), glm::vec3(0.0f, 0.0f, 1.0f));
                model = glm::translate(model,
glm::vec3(2.0f, 5.0f, -15.0f));
                break;
            }
            case 2:
            {
                view = glm::translate(view,
glm::vec3(1.0f, 3.0f, -3.0f));
                model = glm::rotate(model,
(float)glfwGetTime(), glm::vec3(1.0f, 0.0f, 0.0f));
                model = glm::translate(model,
glm::vec3(1.5f, 2.5f, 3.0f));
                break;
            }
            }

            model = glm::rotate(model, (float)
(glfwGetTime()), glm::vec3(0.5f, 0.5f, 0.5f));
            ourShader.setMat4("model", model);

            // render container
            for (int i = 0; i < 6; i++)
            {
                glBindVertexArray(VAO[i]);
                glDrawElements(GL_TRIANGLES, 6,
GL_UNSIGNED_INT, 0);
                // bind textures on corresponding
texture units
                glActiveTexture(GL_TEXTURE0);
                glBindTexture(GL_TEXTURE_2D,
texture[i]);
            }
        }


        // glfw: swap buffers and poll IO events (keys
pressed/released, mouse moved etc.)
        // -------------------------------------------------
----------------------------------
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    // optional: de-allocate all resources once they've
outlived their purpose:
    // -----------------------------------------------------
-----------------------
    glDeleteVertexArrays(6, VAO);
    glDeleteBuffers(6, VBO);
    glDeleteBuffers(6, EBO);

    // glfw: terminate, clearing all previously
allocated GLFW resources.
    // -----------------------------------------------------
-----------------
    glfwTerminate();
    return 0;
}

// process all input: query GLFW whether relevant keys
are pressed/released this frame and react accordingly
// -----------------------------------------------------
----------------------------------------------------
void processInput(GLFWwindow *window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) ==
GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
}
```

```
// glfw: whenever the window size changed (by OS or user
resize) this callback function executes
// -----------------------------------------------------
----------------------------------------
void framebuffer_size_callback(GLFWwindow* window, int
width, int height)
{
    // make sure the viewport matches the new window
dimensions; note that width and
    // height will be significantly larger than
specified on retina displays.
    glViewport(0, 0, width, height);
}
```

四、实验结果及总结

结果

最终得到了三个处于同一平面但大小不同的立方体，并且自选图片对其 进行纹理贴图。三个立方体的平移旋转满足cube1位于中心原点(0.0f，0.0f，0.0f)，并绕其斜对角线自转；cube2绕任意轴自转，并以远离中心原点一定的距离绕z轴公转；cube3绕任意轴自转，并以远离point1(0.5f，0.5f，0.0f) 一定的距离绕该点处的y-z平面法向量旋转。

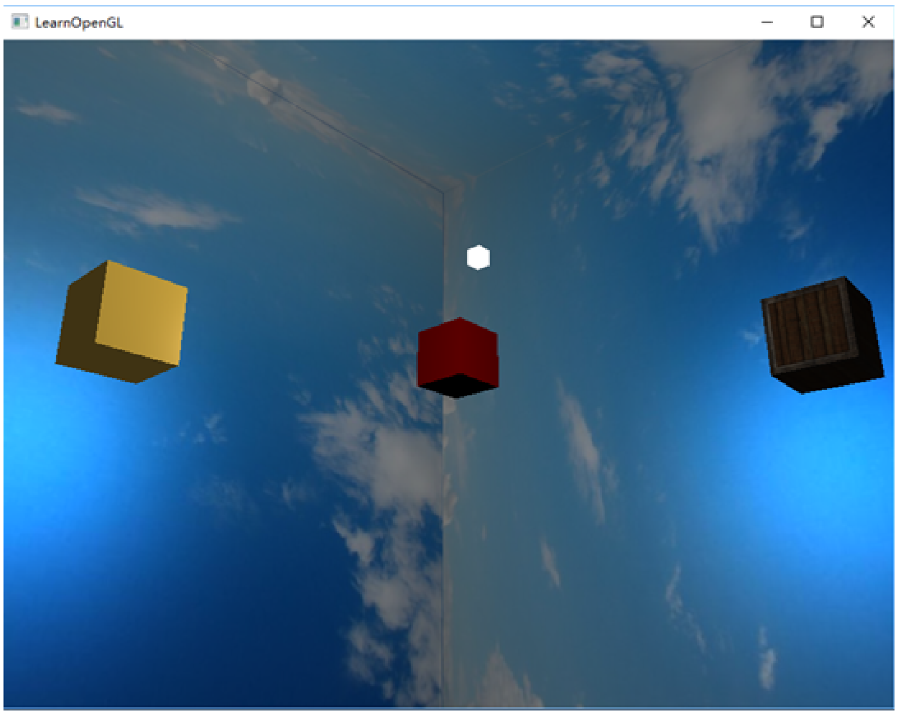
感悟

实现代码的过程中，对于特定的旋转应该如何设计变换矩阵是一大难点，同时对于平移、旋转、缩放的矩阵的顺序也有严格的要求，否则很容易出现不可预期的结果。我在编程的过程中遇到一个问题，就是每次对model的操作都会在前方做矩阵相乘的运算，因此每个立方体都需要新建一个model，否则，前面的影响对后面的变换是不可预期的。

实验三实验报告

一、实验内容

使用OPENGL配置“摄像机”并模拟实现冯氏光照模型(Phong Lighting Model)，然后创建视频中的包含一些不同材质和贴图立方体的光照场景。



说明：

1. 在完成作业前，还是先阅读教程，本次作业涉及的章节为：摄像机、颜色、基础光照、材质、光照贴图。

   请仔细阅读上述教程，理解观察空间和摄像机的实现方法，熟悉物体颜色与材质、光照、视角等因素的关系，掌握包括光照贴图在内的光照场景的实现。

   具体的演示见视频文件“简单的光照场景.mp4”，常见材质的环境、漫反射、镜面及光照强度参数信息见文件“OpenGL_Materials.html”。

   请大家仔细阅读教程，然后每个章节都动手练习，理解坐标变换、摄像机的实现以及冯氏光照模型的原理。

2. 在样例中如上图，我绘制了一个红色塑料、黄金以、教程中示例的金属边框木箱以及蓝天白云的背景。请同学们自己理解光照贴图的素材并写一遍代码实现木箱的绘制，并自己用图片创建一个背景空间，然后根据“OpenGL_Materials.html”中提供的参数自行选择并绘制若干种材质的立方体。

3. 录制一个短视频，通过鼠标和键盘的操作，在背景空间中演示各个物体的光照情形，反应出不同物体的漫反射和镜面反射的变化细节(加上物体的旋转等变换也是很好的体现光照场景的方式)。

二、环境及资源

- 编程语言：c++
- IDE：Visual Studio 2017
- OpenGL

- GLFW GLFW是一个专门针对OpenGL的C语言库，它提供了一些渲染物体所需的最低限度的接口。它允许用户创建OpenGL上下文，定义窗口参数以及处理用户输入，这正是我们需要的。
- CMake CMake是一个工程文件生成工具。用户可以使用预定义好的CMake脚本，根据自己的选择（像是Visual Studio, Code::Blocks, Eclipse）生成不同IDE的工程文件。
- GLAD

  因为OpenGL只是一个标准/规范，具体的实现是由驱动开发商针对特定显卡实现的。由于OpenGL驱动版本众多，它大多数函数的位置都无法在编译时确定下来，需要在运行时查询。但是代码非常复杂，而且很繁琐，我们需要对每个可能使用的函数都要重复这个过程。幸运的是，有些库能简化此过程，其中**GLAD**是目前最新，也是最流行的库。
- camera.h

  摄像机的库

三、具体设计及实现

1. 设计着色器

```
    Shader colorShader("1.colors.vs", "1.colors.fs");
    Shader lampShader("1.lamp.vs", "1.lamp.fs");
    Shader boxShader("wood.vs", "wood.fs");
```

```
colorShader.use();
colorShader.setVec3("objectColor", 1.0f, 0.5f, 0.31f);
colorShader.setVec3("lightColor", 1.0f, 1.0f, 1.0f);
colorShader.setVec3("lightPos", lightPos);
colorShader.setVec3("viewPos", camera.Position);
colorShader.setVec3("material.ambient", 1.0f, 0.5f,
0.31f);
colorShader.setVec3("material.diffuse", 1.0f, 0.5f,
0.31f);
colorShader.setVec3("material.specular", 0.5f, 0.5f,
0.5f);
colorShader.setFloat("material.shininess", 32.0f);
colorShader.setVec3("light.ambient", 0.2f, 0.2f, 0.2f);
colorShader.setVec3("light.diffuse", 0.5f, 0.5f, 0.5f);
colorShader.setVec3("light.specular", 1.0f, 1.0f, 1.0f);

// world transformation
colorShader.setMat4("projection", projection);
colorShader.setMat4("view", view);
colorShader.setMat4("model", model);
```

```
lampShader.use();
                lampShader.setMat4("projection", 
projection);
                lampShader.setMat4("view", view);
                model = glm::scale(model, 
glm::vec3(0.3f)); // a smaller cube
                lampShader.setMat4("model", model);
```

```
boxShader.use();
boxShader.setInt("material.diffuse", 0);
boxShader.setInt("material.specular", 1);
boxShader.setVec3("lightPos", lightPos);
boxShader.setVec3("viewPos", camera.Position);

boxShader.setVec3("light.ambient", 0.2f, 0.2f, 0.2f);
boxShader.setVec3("light.diffuse", 0.5f, 0.5f, 0.5f);
boxShader.setVec3("light.specular", 1.0f, 1.0f, 1.0f);

boxShader.setVec3("material.specular", 0.5f, 0.5f, 
0.5f);
boxShader.setFloat("material.shininess", 64.0f);

model = glm::scale(model, glm::vec3(0.5f));
boxShader.setMat4("model", model);
boxShader.setMat4("projection", projection);
boxShader.setMat4("view", view);
```

2. 加载贴图

```
unsigned int container = 
loadTexture("container2.png");
    unsigned int specular = 
loadTexture("container2_specular.png");
    unsigned int bkg = loadTexture("timg.jpg");
```

```
unsigned int loadTexture(char const * path)
{
    unsigned int textureID;
    glGenTextures(1, &textureID);

    int width, height, nrComponents;
    unsigned char *data = stbi_load(path, 
&width, &height, &nrComponents, 0);
    if (data)
    {
        GLenum format;
        if (nrComponents == 1)
            format = GL_RED;
        else if (nrComponents == 3)
            format = GL_RGB;
        else if (nrComponents == 4)
            format = GL_RGBA;

        glBindTexture(GL_TEXTURE_2D, textureID);
        glTexImage2D(GL_TEXTURE_2D, 0, format,
width, height, 0, format, GL_UNSIGNED_BYTE,
data);
        glGenerateMipmap(GL_TEXTURE_2D);

        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);

        stbi_image_free(data);
    }
    else
    {
        std::cout << "Texture failed to load at
path: " << path << std::endl;
        stbi_image_free(data);
    }

    return textureID;
}
```

3. 键盘鼠标等IO设备交互

```
    void processInput(GLFWwindow *window)
    {
     if (glfwGetKey(window, GLFW_KEY_ESCAPE) ==
GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

     if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        camera.ProcessKeyboard(FORWARD, deltaTime);
     if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        camera.ProcessKeyboard(BACKWARD, deltaTime);
```

最终绘制出一个大的立方体作为背景，里面有几个小的立方体，有其对应的材质和一个光源的光照，并且可以通过键盘上的wasd键和和鼠标进行视角转换。

```
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        camera.ProcessKeyboard(LEFT, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        camera.ProcessKeyboard(RIGHT, deltaTime);
   }
   // glfw: whenever the mouse moves, this callback is
called
   // ---------------------------------------------------
-----
   void mouse_callback(GLFWwindow* window, double xpos,
double ypos)
   {
    if (firstMouse)
    {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }

    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos; // reversed since y-
coordinates go from bottom to top

    lastX = xpos;
    lastY = ypos;

    camera.ProcessMouseMovement(xoffset, yoffset);
   }

   // glfw: whenever the mouse scroll wheel scrolls,
this callback is called
   // ---------------------------------------------------
--------------------
   void scroll_callback(GLFWwindow* window, double
xoffset, double yoffset)
   {
    camera.ProcessMouseScroll(yoffset);
   }
```

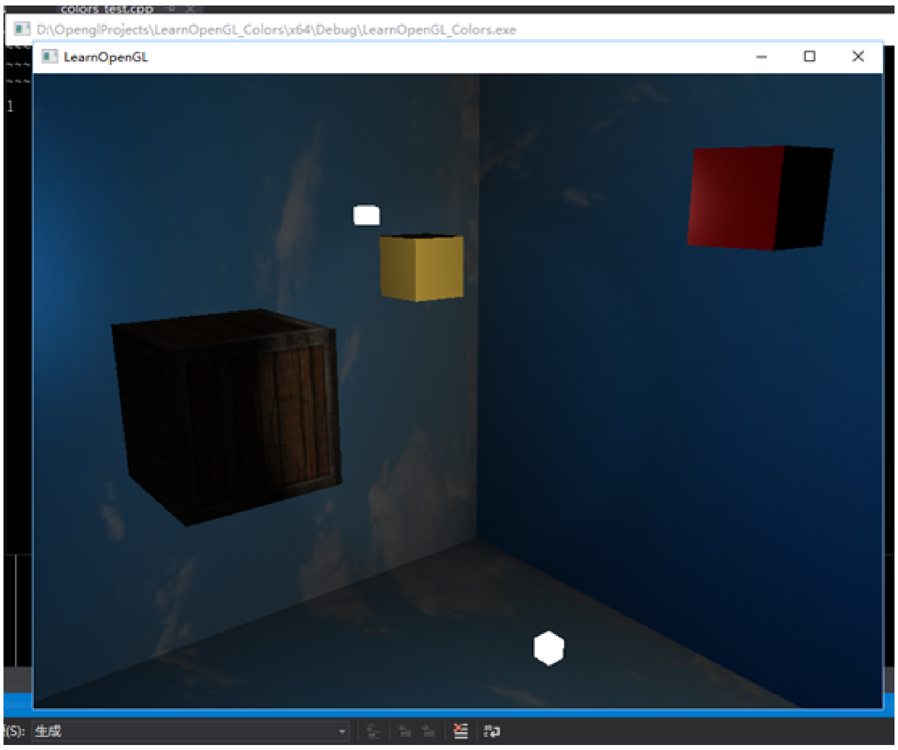
四、实验结果及总结

最终绘制出一个大的立方体作为背景，里面有几个小的立方体，有其对应的材质和一个光源的光照，并且可以通过键盘上的wasd键和和鼠标进行视角转换。

实验四实验报告

一、实验内容

在作业三基础上，实现多光源和聚光的局部光照环境，并加入光照衰减。


colors test.cpp
D:\OpenglProjects\LearnOpenGL_Colors\x64\Debug\LearnOpenGL_Colors.exe
LearnOpenGL
(S): 生成

说明：

1. 在上次作业内容的基础上，阅读教程中，光照部分的最后两个小节：投光物，多光源。

   演示见视频文件“作业四.mp4”，常见材质的环境、漫反射、镜面及光照强度参数信息见文件“OpenGL_Materials.html”。

2. 在作业三中，实现了单光源的冯氏光照，在此基础上，阅读教程后，进一步实现多光源，并考虑与实现如下具体细节：

   (1) 两个或两个以上的点光源。

   (2) 光照强度随距离变化的衰减。

   (3) 实现一个随着镜头视角移动的“手电筒”。(Spotlight，聚光)两个或两个以上的点光源。

3. 录制一个短视频，通过鼠标和键盘的操作，展示在多个点光源和“手电筒”的光照作用下，物体的光照强度信息。

二、环境及资源

- 编程语言：c++
- IDE：Visual Studio 2017
- OpenGL
- GLFW GLFW是一个专门针对OpenGL的C语言库，它提供了一些渲染物体所需的最低限度的接口。它允许用户创建OpenGL上下文，定义窗口参数以及处理用户输入，这正是我们需要的。

- CMake CMake是一个工程文件生成工具。用户可以使用预定义好的CMake脚本，根据自己的选择（像是Visual Studio, Code::Blocks, Eclipse）生成不同IDE的工程文件。
- GLAD

  因为OpenGL只是一个标准/规范，具体的实现是由驱动开发商针对特定显卡实现的。由于OpenGL驱动版本众多，它大多数函数的位置都无法在编译时确定下来，需要在运行时查询。但是代码非常复杂，而且很繁琐，我们需要对每个可能使用的函数都要重复这个过程。幸运的是，有些库能简化此过程，其中**GLAD**是目前最新，也是最流行的库。
- stb_image.h

  `stb_image.h` 是Sean Barrett的一个非常流行的单头文件图像加载库，它能够加载大部分流行的文件格式，并且能够很简单得整合到工程之中。
- GLM OpenGL没有自带任何的矩阵和向量知识，所以我们必须定义自己的数学类和函数。有个易于使用，专门为OpenGL量身定做的数学库，那就是GLM。
- camera.h

  摄像机的头文件

三、具体设计及实现

1. 着色器

color.fs

```
#version 330 core
out vec4 FragColor;

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

struct DirLight {
    vec3 direction;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

struct PointLight {
    vec3 position;
    float constant;
    float linear;
    float quadratic;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

struct SpotLight {
    vec3 position;
    vec3 direction;
    float cutOff;
    float outerCutOff;

    float constant;
    float linear;
    float quadratic;

    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

#define NR_POINT_LIGHTS 4
in vec3 Normal;
in vec3 FragPos;

uniform DirLight dirLight;
uniform PointLight pointLights[NR_POINT_LIGHTS];
uniform SpotLight spotLight;
uniform vec3 viewPos;
uniform vec3 lightColor;
uniform vec3 objectColor;
uniform Material material;

// function prototypes
vec3 CalcDirLight(DirLight light, vec3 normal, vec3
viewDir);
vec3 CalcPointLight(PointLight light, vec3 normal, vec3
fragPos, vec3 viewDir);
vec3 CalcSpotLight(SpotLight light, vec3 normal, vec3
fragPos, vec3 viewDir);

void main()
{
    // properties
    vec3 norm = normalize(Normal);
    vec3 viewDir = normalize(viewPos - FragPos);

    // ==
=====================================================
    // Our lighting is set up in 3 phases: directional,
point lights and an optional flashlight
    // For each phase, a calculate function is defined
that calculates the corresponding color
    // per lamp. In the main() function we take all the
calculated colors and sum them up for
    // this fragment's final color.
    // ==
=====================================================
    // phase 1: directional lighting
    vec3 result = CalcDirLight(dirLight, norm, viewDir);
    // phase 2: point lights
    for(int i = 0; i < NR_POINT_LIGHTS; i++)
        result += CalcPointLight(pointLights[i], norm,
FragPos, viewDir);
    // phase 3: spot light
    result += CalcSpotLight(spotLight, norm, FragPos,
viewDir);

    FragColor = vec4(result, 1.0);

}
// calculates the color when using a directional light.
vec3 CalcDirLight(DirLight light, vec3 normal, vec3
viewDir)
{
    vec3 lightDir = normalize(-light.direction);
    // diffuse shading
    float diff = max(dot(normal, lightDir), 0.0);
    // specular shading
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0),
material.shininess);
    // combine results
    vec3 ambient  = light.ambient * material.ambient;
    vec3 diffuse  = light.diffuse * (diff *
material.diffuse);
    vec3 specular = light.specular * (spec *
material.specular);
    return (ambient + diffuse + specular);
}

// calculates the color when using a point light.
```

```
vec3 CalcPointLight(PointLight light, vec3 normal, vec3
fragPos, vec3 viewDir)
{
    vec3 lightDir = normalize(light.position - fragPos);
    // diffuse shading
    float diff = max(dot(normal, lightDir), 0.0);
    // specular shading
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0),
material.shininess);
    // attenuation
    float distance = length(light.position - fragPos);
    float attenuation = 1.0 / (light.constant +
light.linear * distance + light.quadratic * (distance *
distance));
    // combine results
     vec3 ambient  = light.ambient * material.ambient;
    vec3 diffuse  = light.diffuse * (diff *
material.diffuse);
    vec3 specular = light.specular * (spec *
material.specular);
    ambient *= attenuation;
    diffuse *= attenuation;
    specular *= attenuation;
    return (ambient + diffuse + specular);
}

// calculates the color when using a spot light.
vec3 CalcSpotLight(SpotLight light, vec3 normal, vec3
fragPos, vec3 viewDir)
{
    vec3 lightDir = normalize(light.position - fragPos);
    // diffuse shading
    float diff = max(dot(normal, lightDir), 0.0);
    // specular shading
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0),
material.shininess);
    // attenuation
    float distance = length(light.position - fragPos);
    float attenuation = 1.0 / (light.constant +
light.linear * distance + light.quadratic * (distance *
distance));
    // spotlight intensity
    float theta = dot(lightDir, normalize(-
light.direction));
    float epsilon = light.cutOff - light.outerCutOff;
    float intensity = clamp((theta - light.outerCutOff)
/ epsilon, 0.0, 1.0);
    // combine results
     vec3 ambient  = light.ambient * material.ambient;
    vec3 diffuse  = light.diffuse * (diff *
material.diffuse);
    vec3 specular = light.specular * (spec *
material.specular);
    ambient *= attenuation * intensity;
    diffuse *= attenuation * intensity;
    specular *= attenuation * intensity;
    return (ambient + diffuse + specular);
}
```

color.vs

```
#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);
}
```

lamp.fs

```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(1.0); // set alle 4 vector values
to 1.0
}
```

lamp.vs

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

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

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

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
#include "Shader.h"
#include"camera.h"
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include <iostream>
#include<string.h>
using namespace std;

void framebuffer_size_callback(GLFWwindow* window, int
width, int height);
void mouse_callback(GLFWwindow* window, double xpos,
double ypos);
void scroll_callback(GLFWwindow* window, double xoffset,
double yoffset);
void processInput(GLFWwindow *window);
unsigned int loadTexture(const char *path);
void drawWoodBox(Shader &, glm::mat4 &projection,
glm::mat4 &view, glm::mat4 &model, unsigned int VAO,
unsigned int &diffuseMap, unsigned int &specularMap);

void drawCommonBox(Shader &lightingShader, glm::mat4
&projection, glm::mat4 &view, glm::mat4 &model, unsigned
int cubeVAO);
```

```
void drawLampBox(Shader &lampShader, glm::mat4
&projection, glm::mat4 &view, glm::mat4 &model, unsigned
int lightVAO);

void drawbackBox(Shader & shader, unsigned int
&backgroundMap);

// settings
const unsigned int SCR_WIDTH = 800;
const unsigned int SCR_HEIGHT = 600;

// camera
Camera camera(glm::vec3(0.0f, 0.0f, 3.0f));
float lastX = SCR_WIDTH / 2.0f;
float lastY = SCR_HEIGHT / 2.0f;
bool firstMouse = true;

// timing
float deltaTime = 0.0f; // time between current frame
and last frame
float lastFrame = 0.0f;

// lighting
//glm::vec3 lightPos(1.2f, 1.0f, 2.0f);
glm::vec3 pointLightPositions[] = {
        glm::vec3(0.7f,  0.2f,  2.0f),
        glm::vec3(2.3f, -3.3f, -1.0f),
        glm::vec3(-4.0f,  2.0f, -2.0f),
        glm::vec3(-3.0f,  -4.0f, -1.5f)
};
glm::vec3 cubePositions[] = {
        glm::vec3(0.0f,  0.0f,  1.0f),
        glm::vec3(2.4f, -0.4f, -1.0f),
        glm::vec3(-1.7f,  0.5f, 0.0f),
        glm::vec3(-3.2f,  -1.0f, -2.0f)
};
glm::vec3 centerPos(0.0f, 0.0f, 0.0f);

int main()
{
    // glfw: initialize and configure
    // ------------------------------
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE,
GLFW_OPENGL_CORE_PROFILE);
```

```
#ifdef __APPLE__
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
// uncomment this statement to fix compilation on OS X
#endif

    // glfw window creation
    // --------------------
    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH,
SCR_HEIGHT, "LearnOpenGL", NULL, NULL);
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" <<
std::endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window,
framebuffer_size_callback);
    glfwSetCursorPosCallback(window, mouse_callback);
    glfwSetScrollCallback(window, scroll_callback);

    // tell GLFW to capture our mouse
    glfwSetInputMode(window, GLFW_CURSOR,
GLFW_CURSOR_DISABLED);

    // glad: load all OpenGL function pointers
    // ---------------------------------------
    if
(!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
        std::cout << "Failed to initialize GLAD" <<
std::endl;
        return -1;
    }

    // configure global opengl state
    // -----------------------------
    glEnable(GL_DEPTH_TEST);

    // build and compile our shader zprogram
    // ------------------------------------
    Shader lightingShader("1.colors.vs", "1.colors.fs");
    Shader lampShader("1.lamp.vs", "1.lamp.fs");
    Shader woodShader("wood.vs", "wood.fs");

    //texture
    unsigned int container =
loadTexture("container2.png");
    unsigned int specular =
loadTexture("container2_specular.png");
    unsigned int bkg = loadTexture("timg.jpg");

    // set up vertex data (and buffer(s)) and configure
vertex attributes
    // -------------------------------------------------
-----------------
    float vertices[] = {
        // positions          // normals           //
texture coords
        -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f,
 0.0f,
         0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  1.0f,
 0.0f,
         0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  1.0f,
 1.0f,
         0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  1.0f,
 1.0f,
        -0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f,
 1.0f,
        -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f,
 0.0f,

        -0.5f, -0.5f,  0.5f,  0.0f,  0.0f,  1.0f,  0.0f,
 0.0f,
         0.5f, -0.5f,  0.5f,  0.0f,  0.0f,  1.0f,  1.0f,
 0.0f,
         0.5f,  0.5f,  0.5f,  0.0f,  0.0f,  1.0f,  1.0f,
 1.0f,
         0.5f,  0.5f,  0.5f,  0.0f,  0.0f,  1.0f,  1.0f,
 1.0f,
        -0.5f,  0.5f,  0.5f,  0.0f,  0.0f,  1.0f,  0.0f,
 1.0f,
        -0.5f, -0.5f,  0.5f,  0.0f,  0.0f,  1.0f,  0.0f,
 0.0f,

        -0.5f,  0.5f,  0.5f, -1.0f,  0.0f,  0.0f,  1.0f,
 0.0f,
        -0.5f,  0.5f, -0.5f, -1.0f,  0.0f,  0.0f,  1.0f,
 1.0f,
        -0.5f, -0.5f, -0.5f, -1.0f,  0.0f,  0.0f,  0.0f,
 1.0f,
        -0.5f, -0.5f, -0.5f, -1.0f,  0.0f,  0.0f,  0.0f,
 1.0f,
        -0.5f, -0.5f,  0.5f, -1.0f,  0.0f,  0.0f,  0.0f,
0.0f,
        -0.5f,  0.5f,  0.5f, -1.0f,  0.0f,  0.0f,  1.0f,
0.0f,

         0.5f,  0.5f,  0.5f,  1.0f,  0.0f,  0.0f,  1.0f,
0.0f,
         0.5f,  0.5f, -0.5f,  1.0f,  0.0f,  0.0f,  1.0f,
1.0f,
         0.5f, -0.5f, -0.5f,  1.0f,  0.0f,  0.0f,  0.0f,
1.0f,
         0.5f, -0.5f, -0.5f,  1.0f,  0.0f,  0.0f,  0.0f,
1.0f,
         0.5f, -0.5f,  0.5f,  1.0f,  0.0f,  0.0f,  0.0f,
0.0f,
         0.5f,  0.5f,  0.5f,  1.0f,  0.0f,  0.0f,  1.0f,
0.0f,

        -0.5f, -0.5f, -0.5f,  0.0f, -1.0f,  0.0f,  0.0f,
1.0f,
         0.5f, -0.5f, -0.5f,  0.0f, -1.0f,  0.0f,  1.0f,
1.0f,
         0.5f, -0.5f,  0.5f,  0.0f, -1.0f,  0.0f,  1.0f,
0.0f,
         0.5f, -0.5f,  0.5f,  0.0f, -1.0f,  0.0f,  1.0f,
0.0f,
        -0.5f, -0.5f,  0.5f,  0.0f, -1.0f,  0.0f,  0.0f,
0.0f,
        -0.5f, -0.5f, -0.5f,  0.0f, -1.0f,  0.0f,  0.0f,
1.0f,

        -0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  0.0f,  0.0f,
1.0f,
         0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  0.0f,  1.0f,
1.0f,
         0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  0.0f,  1.0f,
0.0f,
         0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  0.0f,  1.0f,
0.0f,
        -0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  0.0f,  0.0f,
0.0f,
        -0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  0.0f,  0.0f,
1.0f
   };
   // world space positions of our cubes
   unsigned int VBO, cubeVAO;
   glGenVertexArrays(1, &cubeVAO);
   glGenBuffers(1, &VBO);
```

```
    glBindVertexArray(cubeVAO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
vertices, GL_STATIC_DRAW);

    // position attribute
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 *
sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 *
sizeof(float), (void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 *
sizeof(float), (void*)(6 * sizeof(float)));
    glEnableVertexAttribArray(2);

    // second, configure the light's VAO (VBO stays the
same; the vertices are the same for the light object
which is also a 3D cube)
    unsigned int lightVAO;
    glGenVertexArrays(1, &lightVAO);
    glBindVertexArray(lightVAO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    // note that we update the lamp's position
attribute's stride to reflect the updated buffer data
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 *
sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);

    // render loop
    // -----------
    while (!glfwWindowShouldClose(window))
    {
        // per-frame time logic
        // --------------------
        float currentFrame = glfwGetTime();
        deltaTime = currentFrame - lastFrame;
        lastFrame = currentFrame;

        // input
        // -----
        processInput(window);

        // render
        // ------
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);

        // pass projection matrix to shader (note that
in this case it could change every frame)
        glm::mat4 projection =
glm::perspective(glm::radians(camera.Zoom),
(float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);

        // camera/view transformation
        glm::mat4 view = camera.GetViewMatrix();
        glm::mat4 model;

        // render boxes
        drawCommonBox(lightingShader, projection, view,
model, cubeVAO);

        drawLampBox(lampShader, projection, view, model,
lightVAO);

        drawWoodBox(woodShader, projection, view, model,
cubeVAO, container, specular);

        drawbackBox(woodShader, bkg);

        // glfw: swap buffers and poll IO events (keys
pressed/released, mouse moved etc.)
        // -------------------------------------------
----------------------------------
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    // optional: de-allocate all resources once they've
outlived their purpose:
    // ------------------------------------------------
-----------------------
    glDeleteVertexArrays(1, &cubeVAO);
    glDeleteVertexArrays(1, &lightVAO);
    glDeleteBuffers(1, &VBO);

    // glfw: terminate, clearing all previously
allocated GLFW resources.
    // ------------------------------------------------
-----------------
    glfwTerminate();
    return 0;
```

```
}

void drawLampBox(Shader &lampShader, glm::mat4
&projection, glm::mat4 &view, glm::mat4 &model, unsigned
int lightVAO)
{
    // also draw the lamp object
    lampShader.use();
    lampShader.setMat4("projection", projection);
    lampShader.setMat4("view", view);
    glBindVertexArray(lightVAO);
    for (int i = 0; i < 4; i++)
    {
        model = glm::mat4();
        model = glm::translate(model,
pointLightPositions[i]);
        model = glm::scale(model, glm::vec3(0.2f)); // a
smaller cube
        lampShader.setMat4("model", model);
        glDrawArrays(GL_TRIANGLES, 0, 36);
    }
}

void drawCommonBox(Shader &lightingShader, glm::mat4
&projection, glm::mat4 &view, glm::mat4 &model, unsigned
int cubeVAO)
{
    // be sure to activate shader when setting
uniforms/drawing objects
    lightingShader.use();
    lightingShader.setVec3("objectColor", 1.0f, 0.5f,
0.31f);
    lightingShader.setVec3("lightColor", 1.0f, 1.0f,
1.0f);
    //lightingShader.setVec3("lightPos", lightPos);
    lightingShader.setVec3("viewPos", camera.Position);
    lightingShader.setVec3("material.ambient", 1.0f,
0.5f, 0.31f);
    lightingShader.setVec3("material.diffuse", 1.0f,
0.5f, 0.31f);
    lightingShader.setVec3("material.specular", 0.5f,
0.5f, 0.5f);
    lightingShader.setFloat("material.shininess",
32.0f);
    // directional light
    lightingShader.setVec3("dirLight.direction", -0.2f,
-1.0f, -0.3f);
```

```
    lightingShader.setVec3("dirLight.ambient", 0.05f,
0.05f, 0.05f);
    lightingShader.setVec3("dirLight.diffuse", 0.4f,
0.4f, 0.4f);
    lightingShader.setVec3("dirLight.specular", 0.5f,
0.5f, 0.5f);
    // point light 1
    lightingShader.setVec3("pointLights[0].position",
pointLightPositions[0]);
    lightingShader.setVec3("pointLights[0].ambient",
0.05f, 0.05f, 0.05f);
    lightingShader.setVec3("pointLights[0].diffuse",
0.8f, 0.8f, 0.8f);
    lightingShader.setVec3("pointLights[0].specular",
1.0f, 1.0f, 1.0f);
    lightingShader.setFloat("pointLights[0].constant",
1.0f);
    lightingShader.setFloat("pointLights[0].linear",
0.09);
    lightingShader.setFloat("pointLights[0].quadratic",
0.032);
    // point light 2
    lightingShader.setVec3("pointLights[1].position",
pointLightPositions[1]);
    lightingShader.setVec3("pointLights[1].ambient",
0.05f, 0.05f, 0.05f);
    lightingShader.setVec3("pointLights[1].diffuse",
0.8f, 0.8f, 0.8f);
    lightingShader.setVec3("pointLights[1].specular",
1.0f, 1.0f, 1.0f);
    lightingShader.setFloat("pointLights[1].constant",
1.0f);
    lightingShader.setFloat("pointLights[1].linear",
0.09);
    lightingShader.setFloat("pointLights[1].quadratic",
0.032);
    // point light 3
    lightingShader.setVec3("pointLights[2].position",
pointLightPositions[2]);
    lightingShader.setVec3("pointLights[2].ambient",
0.05f, 0.05f, 0.05f);
    lightingShader.setVec3("pointLights[2].diffuse",
0.8f, 0.8f, 0.8f);
    lightingShader.setVec3("pointLights[2].specular",
1.0f, 1.0f, 1.0f);
    lightingShader.setFloat("pointLights[2].constant",
1.0f);
```

```
    lightingShader.setFloat("pointLights[2].linear",
0.09);
    lightingShader.setFloat("pointLights[2].quadratic",
0.032);
    // point light 4
    lightingShader.setVec3("pointLights[3].position",
pointLightPositions[3]);
    lightingShader.setVec3("pointLights[3].ambient",
0.05f, 0.05f, 0.05f);
    lightingShader.setVec3("pointLights[3].diffuse",
0.8f, 0.8f, 0.8f);
    lightingShader.setVec3("pointLights[3].specular",
1.0f, 1.0f, 1.0f);
    lightingShader.setFloat("pointLights[3].constant",
1.0f);
    lightingShader.setFloat("pointLights[3].linear",
0.09);
    lightingShader.setFloat("pointLights[3].quadratic",
0.032);
    // spotLight
    lightingShader.setVec3("spotLight.position",
camera.Position);
    lightingShader.setVec3("spotLight.direction",
camera.Front);
    lightingShader.setVec3("spotLight.ambient", 0.0f,
0.0f, 0.0f);
    lightingShader.setVec3("spotLight.diffuse", 1.0f,
1.0f, 1.0f);
    lightingShader.setVec3("spotLight.specular", 1.0f,
1.0f, 1.0f);
    lightingShader.setFloat("spotLight.constant", 1.0f);
    lightingShader.setFloat("spotLight.linear", 0.09);
    lightingShader.setFloat("spotLight.quadratic",
0.032);
    lightingShader.setFloat("spotLight.cutOff",
glm::cos(glm::radians(12.5f)));
    lightingShader.setFloat("spotLight.outerCutOff",
glm::cos(glm::radians(15.0f)));

    // view/projection transformations

    lightingShader.setMat4("projection", projection);
    lightingShader.setMat4("view", view);

    // render the cube
    glBindVertexArray(cubeVAO);
    model = glm::mat4();
    model = glm::translate(model, cubePositions[1]);
    model = glm::scale(model, glm::vec3(0.5f)); // a
smaller cube
    // world transformation
    lightingShader.setMat4("model", model);
    glDrawArrays(GL_TRIANGLES, 0, 36);

        model = glm::mat4();
        model = glm::translate(model, cubePositions[0]);
        model = glm::scale(model, glm::vec3(0.2f)); // a
smaller cube
        // world transformation
        lightingShader.setMat4("model", model);
        glDrawArrays(GL_TRIANGLES, 0, 36);

}

void drawWoodBox(Shader &shader, glm::mat4 &projection,
glm::mat4 &view, glm::mat4 &model, unsigned int VAO,
unsigned int &diffuseMap, unsigned int &specularMap)
{
    shader.use();
    shader.setInt("material.diffuse", 0);
    shader.setInt("material.specular", 1);
    shader.setVec3("viewPos", camera.Position);

    shader.setVec3("material.specular", 0.5f, 0.5f,
0.5f);
    shader.setFloat("material.shininess", 64.0f);

    // directional light
    shader.setVec3("dirLight.direction", -0.2f, -1.0f,
-0.3f);
    shader.setVec3("dirLight.ambient", 0.05f, 0.05f,
0.05f);
    shader.setVec3("dirLight.diffuse", 0.4f, 0.4f,
0.4f);
    shader.setVec3("dirLight.specular", 0.5f, 0.5f,
0.5f);
    // point light 1
    shader.setVec3("pointLights[0].position",
pointLightPositions[0]);
    shader.setVec3("pointLights[0].ambient", 0.05f,
0.05f, 0.05f);
    shader.setVec3("pointLights[0].diffuse", 0.8f, 0.8f,
0.8f);
    shader.setVec3("pointLights[0].specular", 1.0f,
1.0f, 1.0f);
    shader.setFloat("pointLights[0].constant", 1.0f);
    shader.setFloat("pointLights[0].linear", 0.09);
    shader.setFloat("pointLights[0].quadratic", 0.032);
    // point light 2
    shader.setVec3("pointLights[1].position",
pointLightPositions[1]);
    shader.setVec3("pointLights[1].ambient", 0.05f,
0.05f, 0.05f);
    shader.setVec3("pointLights[1].diffuse", 0.8f, 0.8f,
0.8f);
    shader.setVec3("pointLights[1].specular", 1.0f,
1.0f, 1.0f);
    shader.setFloat("pointLights[1].constant", 1.0f);
    shader.setFloat("pointLights[1].linear", 0.09);
    shader.setFloat("pointLights[1].quadratic", 0.032);
    // point light 3
    shader.setVec3("pointLights[2].position",
pointLightPositions[2]);
    shader.setVec3("pointLights[2].ambient", 0.05f,
0.05f, 0.05f);
    shader.setVec3("pointLights[2].diffuse", 0.8f, 0.8f,
0.8f);
    shader.setVec3("pointLights[2].specular", 1.0f,
1.0f, 1.0f);
    shader.setFloat("pointLights[2].constant", 1.0f);
    shader.setFloat("pointLights[2].linear", 0.09);
    shader.setFloat("pointLights[2].quadratic", 0.032);
    // point light 4
    shader.setVec3("pointLights[3].position",
pointLightPositions[3]);
    shader.setVec3("pointLights[3].ambient", 0.05f,
0.05f, 0.05f);
    shader.setVec3("pointLights[3].diffuse", 0.8f, 0.8f,
0.8f);
    shader.setVec3("pointLights[3].specular", 1.0f,
1.0f, 1.0f);
    shader.setFloat("pointLights[3].constant", 1.0f);
    shader.setFloat("pointLights[3].linear", 0.09);
    shader.setFloat("pointLights[3].quadratic", 0.032);
    // spotLight
    shader.setVec3("spotLight.position",
camera.Position);
    shader.setVec3("spotLight.direction", camera.Front);
    shader.setVec3("spotLight.ambient", 0.0f, 0.0f,
0.0f);
    shader.setVec3("spotLight.diffuse", 1.0f, 1.0f,
1.0f);
    shader.setVec3("spotLight.specular", 1.0f, 1.0f,
1.0f);
```

```
    shader.setFloat("spotLight.constant", 1.0f);
    shader.setFloat("spotLight.linear", 0.09);
    shader.setFloat("spotLight.quadratic", 0.032);
    shader.setFloat("spotLight.cutOff",
glm::cos(glm::radians(12.5f)));
    shader.setFloat("spotLight.outerCutOff",
glm::cos(glm::radians(15.0f)));

    shader.setMat4("projection", projection);
    shader.setMat4("view", view);

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, diffuseMap);

    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_2D, specularMap);

    glBindVertexArray(VAO);

    model = glm::mat4();
    model = glm::translate(model, cubePositions[2]);
    model = glm::scale(model, glm::vec3(1.2f));
    // world transformation
    shader.setMat4("model", model);
    glDrawArrays(GL_TRIANGLES, 0, 36);

    model = glm::mat4();
    model = glm::translate(model, cubePositions[3]);
    model = glm::scale(model, glm::vec3(0.8f)); // a
smaller cube
    // world transformation
    shader.setMat4("model", model);
    glDrawArrays(GL_TRIANGLES, 0, 36);



}

void drawbackBox(Shader & shader, unsigned int
&backgroundMap)
{
    glm::mat4 model;
    model = glm::translate(model, centerPos);
    model = glm::scale(model, glm::vec3(20.0f));    //a
much bigger box as the background
    shader.setMat4("model", model);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, backgroundMap);
    glDrawArrays(GL_TRIANGLES, 0, 36);
```

```
}

// process all input: query GLFW whether relevant keys
are pressed/released this frame and react accordingly
// -----------------------------------------------------
----------------------------------------------------
void processInput(GLFWwindow *window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) ==
GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        camera.ProcessKeyboard(FORWARD, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        camera.ProcessKeyboard(BACKWARD, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        camera.ProcessKeyboard(LEFT, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        camera.ProcessKeyboard(RIGHT, deltaTime);
}

// glfw: whenever the window size changed (by OS or user
resize) this callback function executes
// -----------------------------------------------------
----------------------------------------
void framebuffer_size_callback(GLFWwindow* window, int
width, int height)
{
    // make sure the viewport matches the new window
dimensions; note that width and
    // height will be significantly larger than
specified on retina displays.
    glViewport(0, 0, width, height);
}

// glfw: whenever the mouse moves, this callback is
called
// -----------------------------------------------------
--
void mouse_callback(GLFWwindow* window, double xpos,
double ypos)
{
    if (firstMouse)
    {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }

    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos; // reversed since y-
coordinates go from bottom to top

    lastX = xpos;
    lastY = ypos;

    camera.ProcessMouseMovement(xoffset, yoffset);
}

// glfw: whenever the mouse scroll wheel scrolls, this
callback is called
// -----------------------------------------------------
-----------------
void scroll_callback(GLFWwindow* window, double xoffset,
double yoffset)
{
    camera.ProcessMouseScroll(yoffset);
}

unsigned int loadTexture(char const * path)
{
    unsigned int textureID;
    glGenTextures(1, &textureID);

    int width, height, nrComponents;
    unsigned char *data = stbi_load(path, &width,
&height, &nrComponents, 0);
    if (data)
    {
        GLenum format;
        if (nrComponents == 1)
            format = GL_RED;
        else if (nrComponents == 3)
            format = GL_RGB;
        else if (nrComponents == 4)
            format = GL_RGBA;

        glBindTexture(GL_TEXTURE_2D, textureID);
        glTexImage2D(GL_TEXTURE_2D, 0, format, width,
height, 0, format, GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);

        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_S, GL_REPEAT);
```

```
        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);

        stbi_image_free(data);
    }
    else
    {
        std::cout << "Texture failed to load at path: "
<< path << std::endl;
        stbi_image_free(data);
    }

    return textureID;
}
```

四、实验结果及总结

实现了较为复杂的光照条件：两个或两个以上的点光源；光照强度随距离变化的衰减；一个随着镜头视角移动的“手电筒”。

实验五实验报告

一、实验内容

绘制一个三维场景，其中至少含有2个三维物体。具有光照，纹理效果，可实现交互式的场景漫游。可以使用WebGL、OpenGL或其他函数库，要求独立完成从建模、绘制和渲染、变换(放缩、旋转、位移)、光照、面绘制算法、纹理映射的全部过程。

二、环境及资源

- 编程语言：c++
- IDE：Visual Studio 2017
- OpenGL
- GLFW GLFW是一个专门针对OpenGL的C语言库，它提供了一些渲染物体所需的最低限度的接口。它允许用户创建OpenGL上下文，定义窗口参数以及处理用户输入，这正是我们需要的。
- CMake CMake是一个工程文件生成工具。用户可以使用预定义好的CMake脚本，根据自己的选择（像是Visual Studio, Code::Blocks, Eclipse）生成不同IDE的工程文件。

- GLAD

  因为OpenGL只是一个标准/规范，具体的实现是由驱动开发商针对特定显卡实现的。由于OpenGL驱动版本众多，它大多数函数的位置都无法在编译时确定下来，需要在运行时查询。但是代码非常复杂，而且很繁琐，我们需要对每个可能使用的函数都要重复这个过程。幸运的是，有些库能简化此过程，其中**GLAD**是目前最新，也是最流行的库。

- stb_image.h

  `stb_image.h` 是Sean Barrett的一个非常流行的单头文件图像加载库，它能够加载大部分流行的文件格式，并且能够很简单得整合到工程之中。

- GLM OpenGL没有自带任何的矩阵和向量知识，所以我们必须定义自己的数学类和函数。有个易于使用，专门为OpenGL量身定做的数学库，那就是GLM。

- camera.h

  摄像机的头文件

三、具体设计及实现

1. 三棱锥的顶点坐标

```
float vertices[] = {
// positions          // normals          // texture
coords

 0.5f, -0.5f,  0.5f,  0.0f,  0.0f,  1.0f,  1.0f,  0.0f,
 //1
 -0.5f, -0.5f, -0.5f,  0.0f,  0.0f,  1.0f,  1.0f,
 1.0f,//2
 -0.5f,  0.5f,  0.5f,  0.0f,  0.0f,  1.0f,  0.0f,
 1.0f,//3

 -0.5f, -0.5f, -0.5f,  1.0f,  0.0f,  0.0f,  1.0f,
 0.0f,//2
 0.5f,  0.5f, -0.5f,  1.0f,  0.0f,  0.0f,  1.0f,
 1.0f,//4
 0.5f, -0.5f,  0.5f,  1.0f,  0.0f,  0.0f,  0.0f,
 1.0f,//1

 0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  0.0f,  1.0f,
 1.0f,//4
 -0.5f, -0.5f, -0.5f,  0.0f,  1.0f,  0.0f,  1.0f,
 0.0f,//2
 -0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  0.0f,  0.0f,
 1.0f,//3
 0.5f,  0.5f, -0.5f, -1.0f,  0.0f,  0.0f,  1.0f,
 1.0f,//4
 0.5f, -0.5f,  0.5f, -1.0f,  0.0f,  0.0f,  1.0f,
 0.0f,//1
 -0.5f,  0.5f,  0.5f, -1.0f,  0.0f,  0.0f,  0.0f,
 1.0f,//3
 };
```

2. 球体的实现

```
unsigned int sphereVAO = 0;
unsigned int indexCount;
void renderSphere()
{
    if (sphereVAO == 0)
    {
        glGenVertexArrays(1, &sphereVAO);

        unsigned int vbo, ebo;
        glGenBuffers(1, &vbo);
        glGenBuffers(1, &ebo);

        std::vector<glm::vec3> positions;
        std::vector<glm::vec2> uv;
        std::vector<glm::vec3> normals;
        std::vector<unsigned int> indices;

        const unsigned int X_SEGMENTS = 256;
        const unsigned int Y_SEGMENTS = 256;
        const float PI = 3.14159265359;
        for (unsigned int y = 0; y <= Y_SEGMENTS; ++y)
        {
            for (unsigned int x = 0; x <= X_SEGMENTS;
++x)
            {
                float xSegment = (float)x /
(float)X_SEGMENTS;
                float ySegment = (float)y /
(float)Y_SEGMENTS;
                float xPos = std::cos(xSegment * 2.0f *
PI) * std::sin(ySegment * PI);
                float yPos = std::cos(ySegment * PI);
                float zPos = std::sin(xSegment * 2.0f *
PI) * std::sin(ySegment * PI);

                positions.push_back(glm::vec3(xPos,
yPos, zPos));
```

```
                uv.push_back(glm::vec2(xSegment,
ySegment));
                normals.push_back(glm::vec3(xPos, yPos,
zPos));
            }
        }

        bool oddRow = false;
        for (int y = 0; y < Y_SEGMENTS; ++y)
        {
            if (!oddRow) // even rows: y == 0, y == 2;
and so on
            {
                for (int x = 0; x <= X_SEGMENTS; ++x)
                {
                    indices.push_back(y       *
(X_SEGMENTS + 1) + x);
                    indices.push_back((y + 1) *
(X_SEGMENTS + 1) + x);
                }
            }
            else
            {
                for (int x = X_SEGMENTS; x >= 0; --x)
                {
                    indices.push_back((y + 1) *
(X_SEGMENTS + 1) + x);
                    indices.push_back(y       *
(X_SEGMENTS + 1) + x);
                }
            }
            oddRow = !oddRow;
        }
        indexCount = indices.size();

        std::vector<float> data;
        for (int i = 0; i < positions.size(); ++i)
        {
            data.push_back(positions[i].x);
            data.push_back(positions[i].y);
            data.push_back(positions[i].z);
            if (uv.size() > 0)
            {
                data.push_back(uv[i].x);
                data.push_back(uv[i].y);
            }
            if (normals.size() > 0)
            {
                data.push_back(normals[i].x);
                data.push_back(normals[i].y);
                data.push_back(normals[i].z);
            }
        }
        glBindVertexArray(sphereVAO);
        glBindBuffer(GL_ARRAY_BUFFER, vbo);
        glBufferData(GL_ARRAY_BUFFER, data.size() *
sizeof(float), &data[0], GL_STATIC_DRAW);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER,
indices.size() * sizeof(unsigned int), &indices[0],
GL_STATIC_DRAW);
        float stride = (3 + 2 + 3) * sizeof(float);
        glEnableVertexAttribArray(0);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
stride, (void*)0);
        glEnableVertexAttribArray(1);
        glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,
stride, (void*)(3 * sizeof(float)));
        glEnableVertexAttribArray(2);
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
stride, (void*)(5 * sizeof(float)));
    }
    glBindVertexArray(sphereVAO);
    glDrawElements(GL_TRIANGLE_STRIP, indexCount,
GL_UNSIGNED_INT, 0);
}
```

四、实验结果及总结

结合所有所学知识，画出了四个灯，两个球体，两个三棱锥，并且为之加上在空间内围绕一点的旋转，以及一个手电筒和随输入变换的视角和摄像机。