计算机图形学代码作业报告

> 作者@陈屹峰20373860，全文共3716字。

计算机图形学代码作业报告

- 简介
- 自定义着色器类
- 纹理
  - 创建纹理
  - 应用纹理
- 坐标变换
- 摄像机系统
  - 移动摄像机
  - 平移视角
  - 缩放视角
- Phong光照模型
  - 环境光模拟
  - 漫反射模拟
  - 镜面光模拟
  - 物体材质与光照纹理贴图
- 运行

简介

本次计算机图形学代码作业实现了`OpenGL`下的**坐标变换、物体建模、自定义着色器、手动控制摄像机、Phong式光照、物体材质和光照纹理贴图**等效果。本项目按照助教给定的模板实现了增量开发，除模板外，总代码量约为500-600行。下面介绍各个部分的具体实现。

自定义着色器类

为了方便调试和抽象化接口，这里实现了定义了着色器类`Shader`，并放在`shader_m.h`中。

初始化首先读入着色器内容（这里顶点着色器和片元着色器以`.vs .fs`文件形式存储），存储到几个`string`对象里，接着编译和链接着色器，如果没有编译或者链接错误，则删除着色器。着色器也有几个类内函数，例如`use()`和`setter()`。这样在`main.cpp`中只要创建一个着色器对象，就可以开始使用Shader的各种函数了。

```
class Shader
{
public:
    unsigned int ID;
    Shader(const char* vertexPath, const char* fragmentPath)
    {
        ...
        ... // 1. try to open file path and read.
        const char* vShaderCode = vertexCode.c_str();
        const char * fShaderCode = fragmentCode.c_str();
        // 2. compile shaders
        unsigned int vertex, fragment;
```

```
        // vertex shader
        vertex = glCreateShader(GL_VERTEX_SHADER);
        glShaderSource(vertex, 1, &vShaderCode, NULL);
        glCompileShader(vertex);
        checkCompileErrors(vertex, "VERTEX");
        // fragment Shader
        ...
        // shader Program
        ID = glCreateProgram();
        glAttachShader(ID, vertex);
        glAttachShader(ID, fragment);
        glLinkProgram(ID);
        checkCompileErrors(ID, "PROGRAM");
        // delete the shaders as they're linked into our program now and no
longer necessery
        ...
    }
    // activate the shader
    // ------------------------------------------------------------------------
    void use() const
    {
        glUseProgram(ID);
    }
    // utility uniform functions
    // ------------------------------------------------------------------------
    void setBool(const std::string &name, bool value) const
    {
        glUniform1i(glGetUniformLocation(ID, name.c_str()), (int)value);
    }
    ...

    void setVec2(const std::string &name, const glm::vec2 &value) const
    {
        glUniform2fv(glGetUniformLocation(ID, name.c_str()), 1, &value[0]);
    }
    ...
```

关于灯光和物体的顶点着色器、片元着色器：物体的着色器较为复杂，这里直接定义两套顶点着色器和片元着色器，分别进行设置，减少了对于光源的渲染负担。

纹理

为了使得物体更加真实、拥有更多的细节，同时又不增加过多的顶点，这里采用纹理贴图的方法，增加图形的细节。

创建纹理

这里使用的是`stb_image.h`头文件图形加载库，在源码中引用：

```
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
```

为了为物体不同的位置生成不同的纹理效果，同时减少重复代码，这里将纹理生成的过程抽象成了`loadTexture()`函数，返回`texture`对象的唯一ID。生成纹理的过程与其他`OpenGL`对象一样，需要使用ID进行引用，首先使用`glGenTextures`函数生成一个纹理的ID存储到`int`数组中，然后进行绑定，使得后续的纹理指令可以配置当前绑定的纹理。与此同时，通过`stb_image`库读入一个纹理图像，接着调用`glTexImage2D`来生成纹理，调用`glGenerateMipmap`为当前绑定的纹理自动生成所有需要的多级渐远纹理。

```
unsigned int loadTexture(char const * path)
{
    // generate a texture ID and create a texture
    unsigned int textureID;
    glGenTextures(1, &textureID);

    // load image using stbi_load()
    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;
        // load and generate the texture
        glBindTexture(GL_TEXTURE_2D, textureID);
        glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format,
GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);
        // set the texture wrapping/filtering options (on the currently bound
texture object)
        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;
}
```

应用纹理

顶点坐标数据需要加上纹理坐标。由于额外增加顶点的属性，因此需要制定顶点着色器解释顶点数据方法，调整步长参数为`8 * sizeof(float)`。（normal attribute是后面会提到的法向量）

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

接着我们需要调整顶点着色器使其能够接受纹理坐标，并将坐标传给片段着色器。对于片元着色器，他需要访问纹理对象，这里采用`GLSL`的数据类型`Sampler`，声明一个`uniform sampler2D`就可以把纹理添加到片元着色器中。这里使用纹理单元，使得同一个着色器可以同时设置多个纹理。这里为了区分木板和铁皮在同样光照下的不同情况，设置了`diffuse`和`specluar`两个光照纹理（光照纹理内容见下光照模型章节详述）。

```
// bind diffuse map
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, diffuseMap);
// bind specular map
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, specularMap);
```

坐标变换

助教提供的例程的三角形是一个平面图形，要想实现三维图形的展示需要坐标变换的知识。

课上提到的各种矩阵和向量的抽象均可以由`GLM`头文件库提供。因此在程序头加上：

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

根据课上知识，为每一个空间定义一个变换矩阵：模型矩阵，观察矩阵和投影矩阵，则一个顶点坐标会根据以下的变换到裁剪空间：

$$V_{clip} = M_{projection} \cdot M_{view} \cdot M_{model} \cdot V_{local}$$

在顶点着色器中声明`uniform mat4 model, view, projection`，则`gl_Position =projection * view * model * vec4(aPos, 1.0)`，后面将变换矩阵传递给着色器即可。代码中给出了10个不同的木箱的坐标，通过给定的10个不同的变换矩阵生成了10个材质相同但位置和角度不同的木箱实体。

摄像机系统

OpenGL本身没有摄像机的概念，但我们可以通过把场景中的所有物体往相反方向移动的方式来模拟出摄像机，产生一种我们在移动的感觉，而不是场景在移动。为了方便调试和抽象化接口，这里实现了定义了摄像机类`Camera`，并放在`camera.h`中。

移动摄像机

需要定义摄像机的前方向，上方向和位置坐标，`lookat`函数作为观察矩阵将所有世界坐标变换到观察空间里去：

```
view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
```

定义键盘输入函数，实现摄像机的平移操作：

```
void ProcessKeyboard(Camera_Movement direction, float deltaTime)
    {
        float velocity = MovementSpeed * deltaTime;
        if (direction == FORWARD)
            Position += Front * velocity;
        if (direction == BACKWARD)
            Position -= Front * velocity;
        if (direction == LEFT)
            Position -= Right * velocity;
        if (direction == RIGHT)
            Position += Right * velocity;
    }
```

为了保证具有设备不变性的移动速度，这里使用渲染每一帧的时间差计算摄像机移动的速度，即`cameraSpeed=2.5f*deltatime`。

平移视角

这里使用鼠标的移动来转换视角方向。根据欧拉角的相关知识，给定俯仰角和偏航角，可以得到一个代表着新的摄像机的方向向量。

```
glm::vec3 front;
front.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
front.y = sin(glm::radians(pitch));
front.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
cameraFront = glm::normalize(front);
```

为了监听鼠标的移动，定义一个监听鼠标移动的函数`void mouse_callback(GLFWwindow* window, double xpos, double ypos)`。当注册到`glfw`系统中后，一移动鼠标就会调用该函数。

首先我们需要计算鼠标据上一帧的偏移量，然后把偏移量加到俯仰角和偏航角上，同时为了避免摄像机发生**死亡旋转**，这里需要限制俯仰角和偏航角的最大最小值，即两个角度必须落入$[-89^\circ, 89^\circ]$中，最后根据俯仰角和偏航角计算真正的方向向量。

缩放视角

所谓缩放视角就是调整摄像机能够在场景中看到多大的范围，当视野变小的时候，场景投影出来的空间就会减少，产生放大的效果。同样定义`scroll_callback`函数，改变视野值`camera.Zoom`，作为`projection`矩阵的输入。

```
glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom),
(float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
```

最终的`Camera`类：注意每次更改后需要更改类的数据，调用`updateCameraVectors()`

```
class Camera
{
public:
    // camera Attributes
    glm::vec3 Position;
    glm::vec3 Front;
    glm::vec3 Up;
    glm::vec3 Right;
    glm::vec3 WorldUp;
    // euler Angles
    float Yaw;
    float Pitch;
    // camera options
    float MovementSpeed;
    float MouseSensitivity;
    float Zoom;

    // constructor with vectors
    Camera(glm::vec3 position = glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3 up =
glm::vec3(0.0f, 1.0f, 0.0f), float yaw = YAW, float pitch = PITCH) :
Front(glm::vec3(0.0f, 0.0f, -1.0f)), MovementSpeed(SPEED),
MouseSensitivity(SENSITIVITY), Zoom(ZOOM)
    {
        ...
    }

    // returns the view matrix calculated using Euler Angles and the LookAt
Matrix
    glm::mat4 GetViewMatrix()
    {
        return glm::lookAt(Position, Position + Front, Up);
    }

    // processes input received from any keyboard-like input system. Accepts
input parameter in the form of camera defined ENUM (to abstract it from windowing
systems)
    void ProcessKeyboard(Camera_Movement direction, float deltaTime)
    {
        float velocity = MovementSpeed * deltaTime;
        if (direction == FORWARD)
            Position += Front * velocity;
        if (direction == BACKWARD)
            Position -= Front * velocity;
```

```
        if (direction == LEFT)
            Position -= Right * velocity;
        if (direction == RIGHT)
            Position += Right * velocity;
    }

    // processes input received from a mouse input system. Expects the offset
value in both the x and y direction.
    void ProcessMouseMovement(float xoffset, float yoffset, GLboolean
constrainPitch = true)
    {
        xoffset *= MouseSensitivity;
        yoffset *= MouseSensitivity;

        Yaw   += xoffset;
        Pitch += yoffset;

        // make sure that when pitch is out of bounds, screen doesn't get
flipped
        if (constrainPitch)
        {
            if (Pitch > 89.0f)
                Pitch = 89.0f;
            if (Pitch < -89.0f)
                Pitch = -89.0f;
        }

        // update Front, Right and Up Vectors using the updated Euler angles
        updateCameraVectors();
    }

    // processes input received from a mouse scroll-wheel event. Only requires
input on the vertical wheel-axis
    void ProcessMouseScroll(float yoffset)
    {
        Zoom -= (float)yoffset;
        if (Zoom < 1.0f)
            Zoom = 1.0f;
        if (Zoom > 45.0f)
            Zoom = 45.0f;
    }

private:
    // calculates the front vector from the Camera's (updated) Euler Angles
    void updateCameraVectors()
    {
        // calculate the new Front vector
        glm::vec3 front;
        front.x = cos(glm::radians(Yaw)) * cos(glm::radians(Pitch));
        front.y = sin(glm::radians(Pitch));
        front.z = sin(glm::radians(Yaw)) * cos(glm::radians(Pitch));
        Front = glm::normalize(front);
        // also re-calculate the Right and Up vector
```

```
        Right = glm::normalize(glm::cross(Front, WorldUp));  // normalize the
vectors, because their length gets closer to 0 the more you look up or down which
results in slower movement.
        Up    = glm::normalize(glm::cross(Right, Front));
    }
};
```

Phong光照模型

根据课上所讲内容，Phong光照模型有三个光照分量组成：环境光、漫反射和镜面光照。本节所讨论的光照模型以及改进都可以通过对于木箱对应的片元着色器进行调整而实现。

环境光模拟

由于全局光照开销较大比较麻烦，这里直接将环境光照添加到物体片段最终的颜色当中去。

```
FragColor = vec4(ambientStrength * lightColor * objectColor, 1.0)
```

漫反射模拟

根据课上知识，计算漫反射需要法向量和光线的方向向量。

由于立方体较为简单，则我们可以直接将法向量的数据直接添加到顶点数据数组`vertices[]`中，同时更新顶点着色器，将法向量由顶点着色器传递到片元着色器。

接下来需要需要光线的方向向量，由于我将光源的位置设置成了正弦波动的形式，所以需要在渲染循环内部进行更新光源的位置。

```
lightPos.x = 1.0f + sin(glfwGetTime()) * 2.0f;
lightPos.y = sin(glfwGetTime() / 2.0f) * 1.0f;
lightingShader.setVec3("light.position", lightPos);
```

我们还需要片元的位置，所以需要在顶点着色器中将模型转换到世界坐标中，进行计算

```
FragPos = vec3(model * vec4(aPos, 1.0));
```

在片元着色器中计算光源对当前片元实际的漫反射效果

```
in vec3 FragPos;
vec3 norm = normalize(Normal);
vec3 lightDir = normalize(lightPos - FragPos);
float diff = max(dot(norm, lightDir), 0.0); // note: when occluded, it is not
visible
vec3 diffuse = diff * lightColor;
```

注意法向量的变换不能简单上模型矩阵

```
Normal = mat3(transpose(inverse(model))) * aNormal;
```

镜面光模拟

根据课上知识，通过根据法向量翻折入射光的方向来计算反射向量。然后我们计算反射向量与观察方向的角度差，它们之间夹角越小，镜面光的作用就越大。由此产生的效果就是，我们看向在入射光在表面的反射方向时，会看到一点高光。

与漫反射模拟类似，同样是计算向量夹角，这里不再赘述

```
vec3 viewDir = normalize(viewPos - FragPos);
vec3 reflectDir = reflect(-lightDir, norm);
float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
vec3 specular = specularStrength * spec * lightColor;
```

最终我们将所有光照分量与物体本身的颜色相乘，就得到了最终的物体的画面

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

物体材质与光照纹理贴图

物体材质在图形学中就是物体在一定光照下的光照效果，实现光照纹理就是在一定光照条件下实现前文所述的纹理。为了实现这样的功能，在片元着色器中定义两个结构：

```
struct Material {
    vec3 ambient;
    sampler2D diffuse;
    sampler2D specular;
    float shininess;
};
struct Light {
    vec3 position;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};
uniform Material material;
uniform Light light;
```

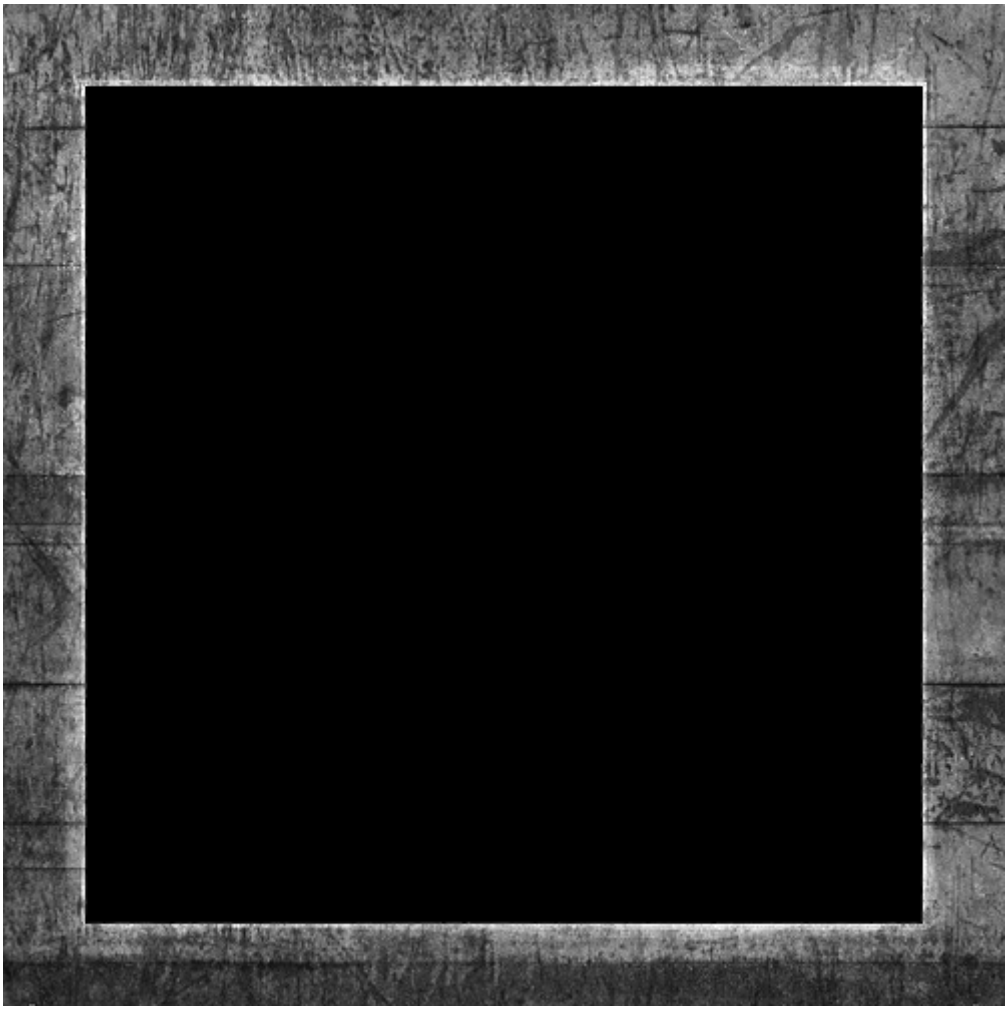
对于一个物体来说，可以通过更改 `Material.ambient, Material.diffuse, ..., Material.shininess` 来更改物体对于Phong光照模型的特性，来改变自身的材质属性。

对于一个光源来说，在给定物体的材质的情况下，为了使得效果更加真实，可以对于其各个分量的强度进行调整。

为了实现木箱的木条和铁皮在同样光照条件下的不同的光照特性，这里使用了光照纹理贴图（见纹理）。

对于漫反射而言，使用了一个正常视角下的铁皮木桶图片，同样将纹理存储为 `Material` 结构体中的一个 `sampler2D`。

对镜面光的纹理而言，使用了一个铁皮部分像素较亮、而模板部分完全为黑的木箱图片，这样就可以得到铁皮的那种高光反射、而木板几乎不产生镜反射的特点了。



环境光也使用了漫反射的纹理贴图。仿照纹理部分和Phong模型模拟部分对木箱的片元着色器更改，最终如下：

```
#version 330 core
out vec4 FragColor;

struct Material {
    sampler2D diffuse;
    sampler2D specular;
    float shininess;
};

struct Light {
    vec3 position;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

in vec3 FragPos;
in vec3 Normal;
in vec2 TexCoords;

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

void main()
```

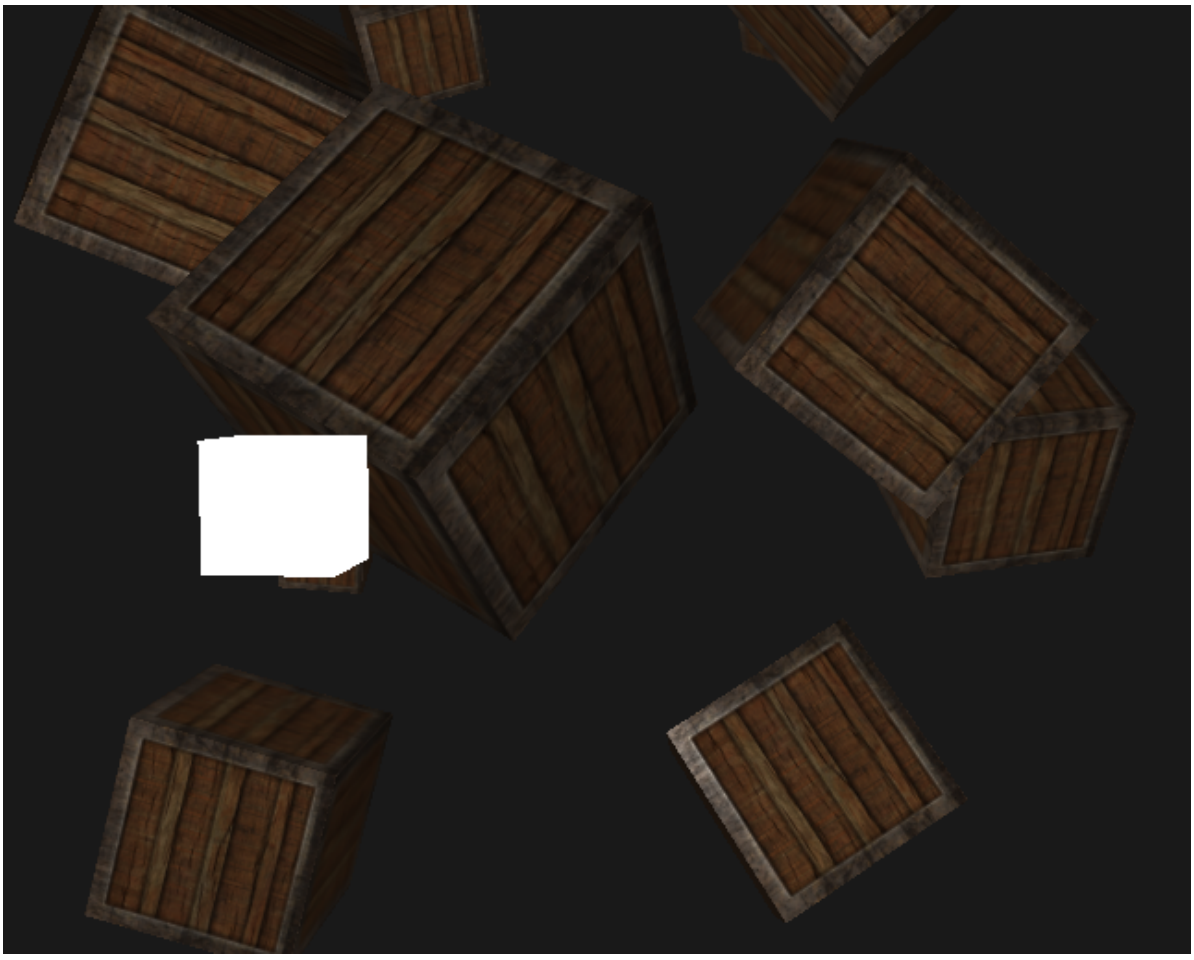
```
{
    // ambient
    vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));

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

    // 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 * vec3(texture(material.specular,
TexCoords)));

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

效果最终为



运行

```
$ make run
```

可以移动鼠标控制摄像机的视角，滑动鼠标中键进行视角缩放，按动`WSAD`键控制摄像机的运动方向，仔细观察Phong光照模型以及铁皮和木箱的材质在光照下的影响。按`ESC`退出。去掉源文件`main.cpp`207-208行的注释，可以实现光源的运动，以更好地观察。