计算机图形学课程报告

计算机科学与技术学院

班　　级： CS1904

学　　号： U201914958

姓　　名： 陶宇飞

指导教师： 何云峰

完成日期： 2022/12/28

1 简答

（1） 你选修计算机图形学课程，想得到的是什么知识？现在课程结束，对于所得的知识是否满意？如果不满意，你准备如何寻找自己需要的知识。

我选修计算机图形学课程是想学习如何使用计算机来生成和处理图像和三维图形的知识。具体来说，有如何使用计算机图形学软件和框架（如 OpenGL）来生成图像和三维图形，如何使用图形学算法来渲染三维图形，包括光照模型，材质和纹理。如何使用图形学技术来实现计算机动画，包括运动插值，碰撞检测和物理模拟等方面的知识，对图形学理论有基本了解。

在学习的过程中，我感觉自己在这门课程中学到了足够多的知识，并且觉得这些知识对我有帮助。

（2） 你对计算机图形学课程中的哪一个部分的内容最感兴趣，请叙述一下，并谈谈你现在的认识。

我对课程中光照模型的部分最感兴趣。光照模型是用于描述物体在不同光照条件下的表面外观的方法。在计算机图形学中，光照模型用于在计算机屏幕上渲染三维图形时，模拟光照和阴影的效果。

常用的光照模型包括漫反射光照模型、镜面光照模型和折射光照模型。漫反射光照模型描述物体表面在漫反射光线照射下的散射行为，镜面光照模型描述物体表面在反射光线照射下的反射行为，折射光照模型描述物体表面在透射光线照射下的折射行为。

在本课程中，我学习了光照模型的基本原理和常用的计算方法。我现在能够理解光照模型的作用，并能够使用 phong 模型来渲染三维图形，使图形看起来更真实。

Phong 光照模型计算物体表面的颜色时，考虑了光源的颜色、强度和方向以及观察者的位置。它使用了环境光照和漫反射光照两种光照模型来计算物体表面的颜色。环境光照模型描述了物体表面受到的环境光的影响，它把环境光看做是从整个场景中同时照射到物体表面的光。漫反射光照模型描述了物体表面受到的单个光源的影响，它计算出物体表面每个点受到光源的影响的程度，并根据这个程度计算出物体表面的颜色。

除此之外，我还对光线追踪技术十分感兴趣。光线追踪是一种用于生成图像的算法，被广泛的应用于现代的大型 3D 游戏中，被许多显卡硬件支持和加速。它通过从观察者的视线发射虚拟光线，并模拟光线在场景中的传播和反射行为来生成图像。光线追踪可以生成高质量的图像，并且可以模拟真实世界中光线的行为。光线追踪算法可以分为两个阶段：光线追踪和光线求交。在光线追踪阶段，算法从观察者的视点发射虚拟光线，并计算光线在场景中的传播路径。在光线求交阶段，算法检测光线是否与场景中的物体相交，并计算出交点的位置。光线追

踪算法可以使用不同的方法来提高渲染效率，例如层次结构光线追踪和八叉树光线追踪。这些方法可以帮助减少计算量，提高渲染效率。我在学习计算机图形学课程时扩充了解了光线追踪的基本原理和常用的算法。我现在能够理解光线追踪算法的基本原理，对其更底层的优化和实现需要在之后的学习中进一步了解。

（3） 你对计算机图形学课程的教学内容和教学方法有什么看法和建议。

在本课程中，教学内容包括了图形学的基础知识，例如三维图形的表示方法、图形变换、光照模型、渲染算法等。教学方法采用了讲授和实践相结合的方式，在讲授基础知识的同时，也给出了大量的实践练习的例子，帮助我们加深对知识的理解。我觉得教学内容涵盖了图形学的基础知识，让我们对这门学科有了一个较为全面的认识。教学方法也很有效，通过实践练习，我们很好地掌握了所学知识。

但是，我还有一些建议。我觉得可以在课程中加入更多的实战项目，让我们在掌握基础知识的同时，也可以体验到解决实际问题的乐趣。还可以引入更多的新技术和最新发展，帮助我们了解图形学领域的最新进展。

同时，随着计算机图形学技术的发展，新的技术和方法也不断涌现。例如，近年来有很多新的图形学技术和方法被提出，例如物理渲染、单位光源模型、立体视觉技术、深度图像技术等。这些新技术和方法可以帮助我们更好地处理图形学中的问题，也为我们提供了更多的可能性和想象空间。

因此，我建议在计算机图形学课程中，可以引入更多的新技术和最新发展，帮助我们了解图形学领域的最新进展，并在实践中学习这些新技术和方法。

2 论述

2.1 实验内容

利用 OpenGL 框架，设计一个独孤信印章模型动画。要求如下：

（1）构建独孤信印章模型，该模型可以通过指定的外接球半径控制大小。

（2）增加纹理（凹凸纹理），可以使用木纹纹理或者其它图片纹理。

（3）构建观察空间，使用三维投影对模型进行观察，要求模型能够进行自转，通过键盘可以控制模型的大小和转速。

（4）增加光照处理，光源可以设在左上方。

（5）在上面的基础上，构建两个印章模型，一大一小，大印章自转，小印章绕着大印章旋转，要求处理好消隐。

2.2 实验方法和过程

（1） 模型渲染

使用 blender 制作模型，“独孤信印章”为正二十六面体。在 blender 中添加

正方体，使用 ctrl+b 快捷键进入倒角模式，调整面数为 26 可得到所谓的“独孤信印章”，统计信息见图 2.1 左上角。

在模型表面为其添加准备好的纹理和纹理的法线贴图，添加完后将其导出为 wavefront 格式进行保存。

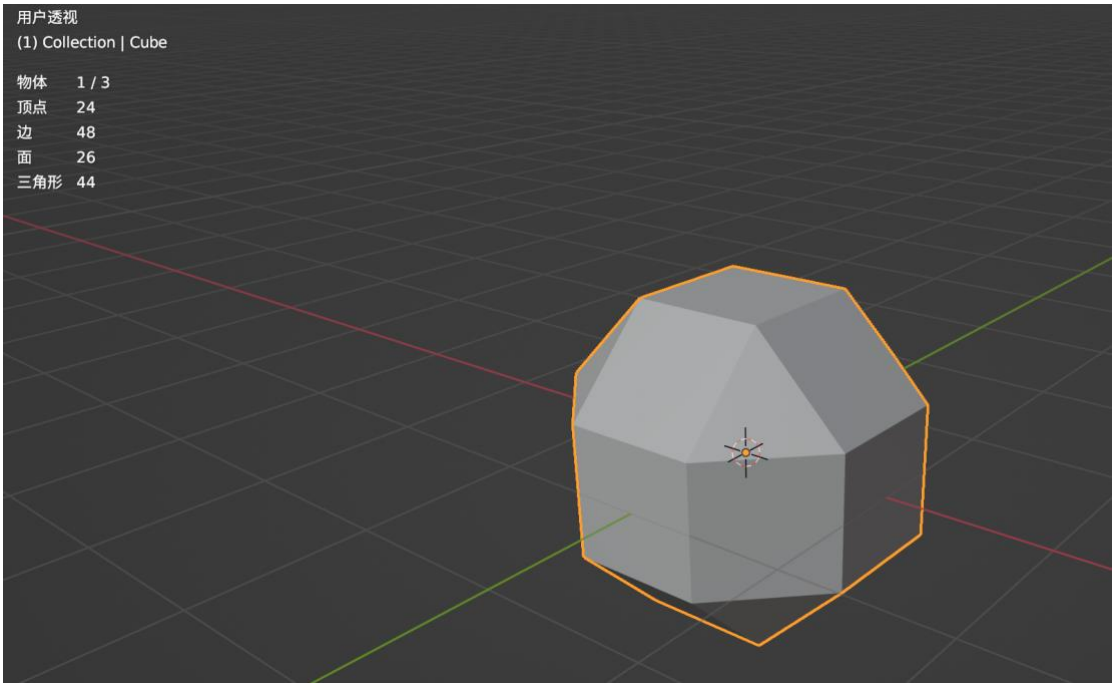

图 2.1 blender 中的模型统计信息

Assimp 是一个开源的模型处理库，能够从各种模型文件格式（如 wavefront）中读取模型的顶点信息和贴图路径，并将读取结果保存为统一的格式供用户解析和渲染。

在本实验代码中，实现了从 assmip 类层次中读取顶点信息和贴图路径，并将其转换为 opengl 的 VAO 和 VBO 的工具类。使用 stb 库读取并解码纹理贴图，将其转化为 opengl 使用的 rgb 数组。

使用 modern opengl 接口渲染模型，使用 GLSL 编写模型的 vertex shader 和 fragment shader。在顶点 shader 中，将顶点与 MVP 矩阵相乘，得到顶点的标准坐标。在片段 shader 中，使用 texture 函数进行贴图的纹理映射，根据顶点的纹理坐标进行采样。

（2） 模型变换

任务要求能够改变二十六面体外接圆大小，修改渲染过程中的模型变换矩阵即可。下面的代码示例首先将模型按 y 轴旋转 theta 角度，在放缩为 scale 倍，最后平移到模型当前的坐标即可。

模型变换矩阵的计算
glm::mat4 model = glm::mat4(1.0f); **model = glm::translate(model, context.position);**

```
model = glm::scale(model, glm::vec3(context.scale));
model = glm::rotate(model, context.theta, glm::vec3(0.0f, 1.0f, 0.0f));
```

接着需要实现模型的视图变换，代码实现了 camera 类，记录了摄像机当前的位置和角度。该类内部记录鼠标的位置和移动，并持续更新摄像机的位置和角度，方法 GetViewMatrix 返回摄像机对应的视图变换矩阵。

视图变换矩阵的计算

```
glm::mat4 GetViewMatrix() const {
    return glm::lookAt(Position, Position + Front, Up);
}
```

最后，采用透视投影，计算矩阵的投影矩阵。

投影变换矩阵的计算

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

将三个矩阵相乘，可以得到模型的 MVP 矩阵，供 vertex shader 进行顶点变换。

（3） 光照模型

采用 phong 模型作为光照模型。模型的光照分为环境光，漫反射和高光。在场景中设置两个光源，一个是左上角固定的点光源，另一个是随摄像机一同移动的手电筒光源，最终模型表面的颜色来自两个光源的叠加。

首先是 phong 模型的 GLSL 实现，光照的各成分对应的代码如下。

光照强度的计算

```
    // obtain normal from normal map in range [0,1]
    vec3 normal = texture(texture_normal1, fs_in.TexCoords).rgb;
    // transform normal vector to range [-1,1]
    normal = normalize(normal * 2.0 - 1.0);   // this normal is in tangent space

    // get diffuse color
    vec3 color = texture(texture_diffuse1, fs_in.TexCoords).rgb;
    // ambient
    vec3 ambient = 0.1 * color;
    // diffuse
    vec3 lightDir = normalize(fs_in.TangentLightPos - fs_in.TangentFragPos);
    float diff = max(dot(lightDir, normal), 0.0);
    vec3 diffuse = diff * color;
    // specular
    vec3 viewDir = normalize(fs_in.TangentViewPos - fs_in.TangentFragPos);
    vec3 reflectDir = reflect(-lightDir, normal);
    vec3 halfwayDir = normalize(lightDir + viewDir);
    float spec = pow(max(dot(normal, halfwayDir), 0.0), 32.0);
    vec3 specular = light.specular * spec;
```

为光照添加根据距离的衰减，使用公式（2-1）进行衰减因子的计算，并将最

终的光照乘以该因子。

$$F_{att} = \frac{1.0}{K_c + K_l * d + K_q * d^2} \quad (2-1)$$

上式的常数项通常保持在 1.0，主要是为了确保分母永远不会小于 1，否则会在某些距离上提高光照强度。线性项与距离值相乘，以线性方式减少强度。平方项与距离的平方相乘，为光源设置了一个二次衰减。当距离很小时，平方项与线性项相比不太重要，但随着距离的增加，平方项变得非常大。

由于平方项的存在，光强会首先以线性方式衰减，直到距离足够大，使平方项超过线性项，然后光强会急剧减少，shader 中衰减项计算实现如下。

光照衰减强度的计算

```
// attenuation
float distance    = length(light.position - fs_in.FragPos);
float attenuation = 1.0 / (light.constant + light.linear * distance + light.quadratic * (distance * distance));
ambient  *= attenuation;
diffuse  *= attenuation;
specular *= attenuation;
```

最后是手电筒光源的额外处理，如图 2.2 所示。在上述点光源模型上，对光线角度θ做特殊处理即可。比较简单的方式是将入射光线角度大于$\emptyset$的反射光线衰减为 0。但是这样实现的手电筒光圈边缘十分锐利，和现实中不同。因此在$\emptyset$附近需要做平滑处理，使得手电筒光圈变柔和，使用公式（2-2）计算手电筒的衰减强度。

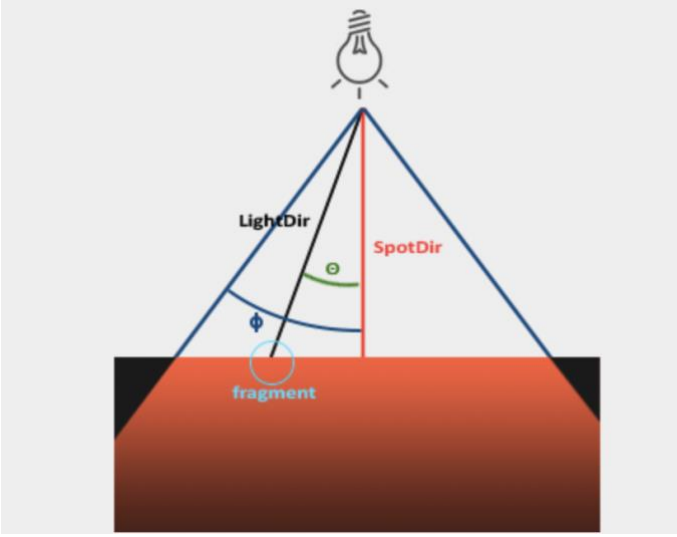

图 2.2 手电筒光照模型

柔和的基本思路是在原来的内层光圈（$\emptyset$）的基础上添加外层光圈（γ），内层光圈内的光线不衰减，内外层之间的光线强度向外逐渐衰减，在外层光圈处衰减至 0。

$$I = Clamp(\frac{\theta - \gamma}{\emptyset - \gamma}) \quad (2-2)$$

其中的$Clamp$函数定义如公式（2-3）所示。

$$Clamp(x) = \begin{cases} 1\ (x > 1) \\ x\ (1 \geq x > 0) \\ 0\ (otherwise) \end{cases} \qquad (2-3)$$

对应的 GLSL 函数实现如下。

光照衰减强度的计算

```
// spotlight (soft edges)
vec3 rlightDir = normalize(light.position - fs_in.FragPos);
float theta = dot(rlightDir, normalize(-light.direction));
float epsilon = (light.cutOff - light.outerCutOff);
float intensity = clamp((theta - light.outerCutOff) / epsilon, 0.0, 1.0);
diffuse  *= intensity;
specular *= intensity;
```

最后，将点光源和手电筒光源叠加即可得到最终的光照。

（4） glfw

在主程序中记录大小模型的位置，在 render loop 的每一次循环中会读取当前时间，并根据当前时间和上次循环的时间差来更新模型的位置，角度和摄像机位置，防止因为不同平台的计算能力不同导致的模型和相机移动或旋转的速率不同。

在渲染循环中同时读取用户键盘输入，进行更改模型大小，修改旋转速度等操作，更新对应的参数。在随后的渲染过程中读取这些参数，计算 MVP 矩阵，并进行渲染。

（5） 凹凸纹理

在导入模型贴图时，同时加载了图形的法线贴图。在之前的光照模型中，默认模型同一面上的所有 fragment 的法向量均相同，为该面的法向量，因此该纹理是“光滑”的，没有凹凸不平的质感，特别是对于 phong 光照的高光项来说。

因此导入法线贴图，法线贴图与贴图类似，但是纹理坐标映射的不是像素值，而是该点的法向量方向。注意这里的法向量是相对贴图的法向量，但是贴图贴在模型上后，本身会有一个初始方向，因此要进行 TBN 变换，将顶点位置，入射方向，和相机位置变换进正切空间，在正切空间中进行光线计算，TBN 矩阵的计算如下所示。

TBN 变换的实现

```
mat3 normalMatrix = transpose(inverse(mat3(model)));
vec3 T = normalize(normalMatrix * aTangent);
vec3 N = normalize(normalMatrix * aNormal);
T = normalize(T - dot(T, N) * N);
vec3 B = cross(N, T);

mat3 TBN = transpose(mat3(T, B, N));
vs_out.TangentLightPos = TBN * light.position;
vs_out.TangentViewPos  = TBN * viewPos;
```

```
vs_out.TangentFragPos    = TBN * vs_out.FragPos;
vs_out.TangentPointPos = TBN * point.position;
```

至此，可以观察到模型表面反光不再是光滑的贴图，而是具有凹凸不平的纹理。

2.3 实验结果

实现如图所示场景，一个印章围绕另一个印章旋转，WASD 键可以在场景中自由移动。U 和 J 可以调整大模型大小，H 和 K 调整小模型公转速度，M 和 N 调整模型自转的速度。可以观察到图 2.3 和图 2.4 中表面凹凸不平的反射高光。场景中除了左上角固定白色点光源，还有来自摄像头的手电筒光源，二者叠加得到模型表面的颜色。

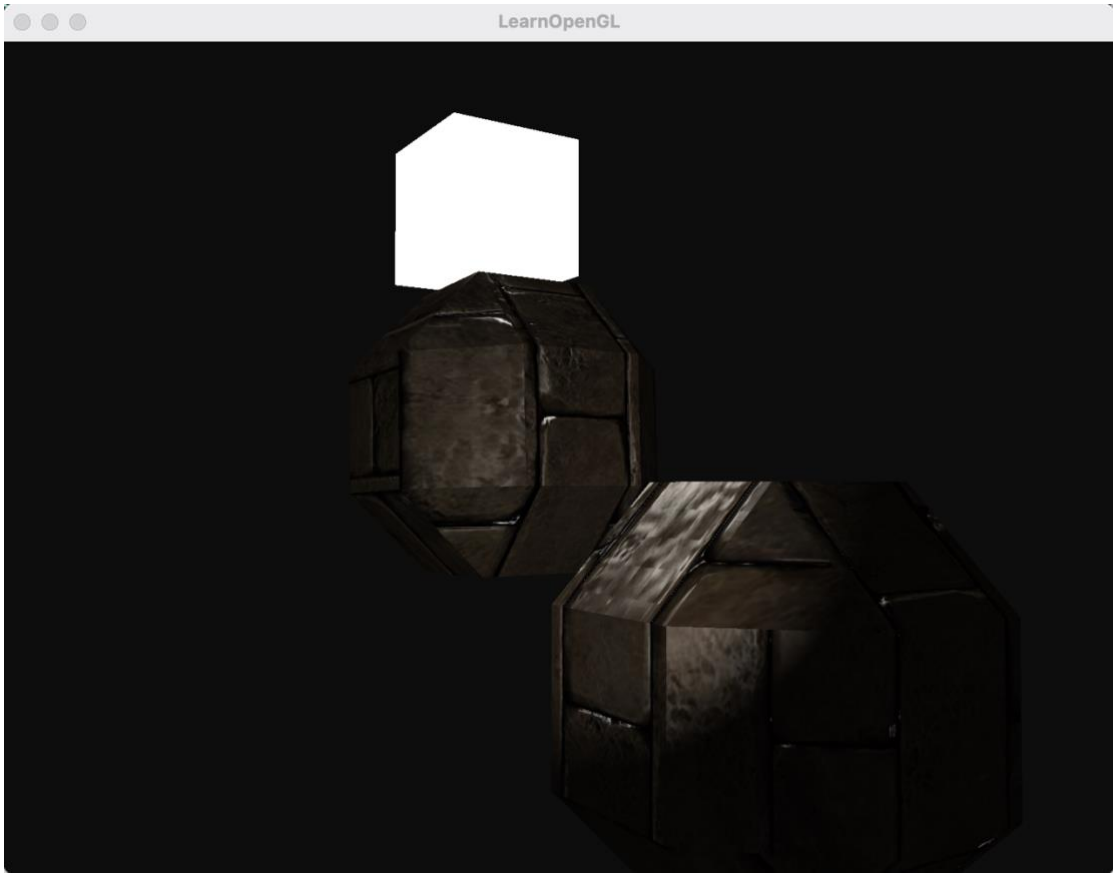

图 2.3 运行效果（1）

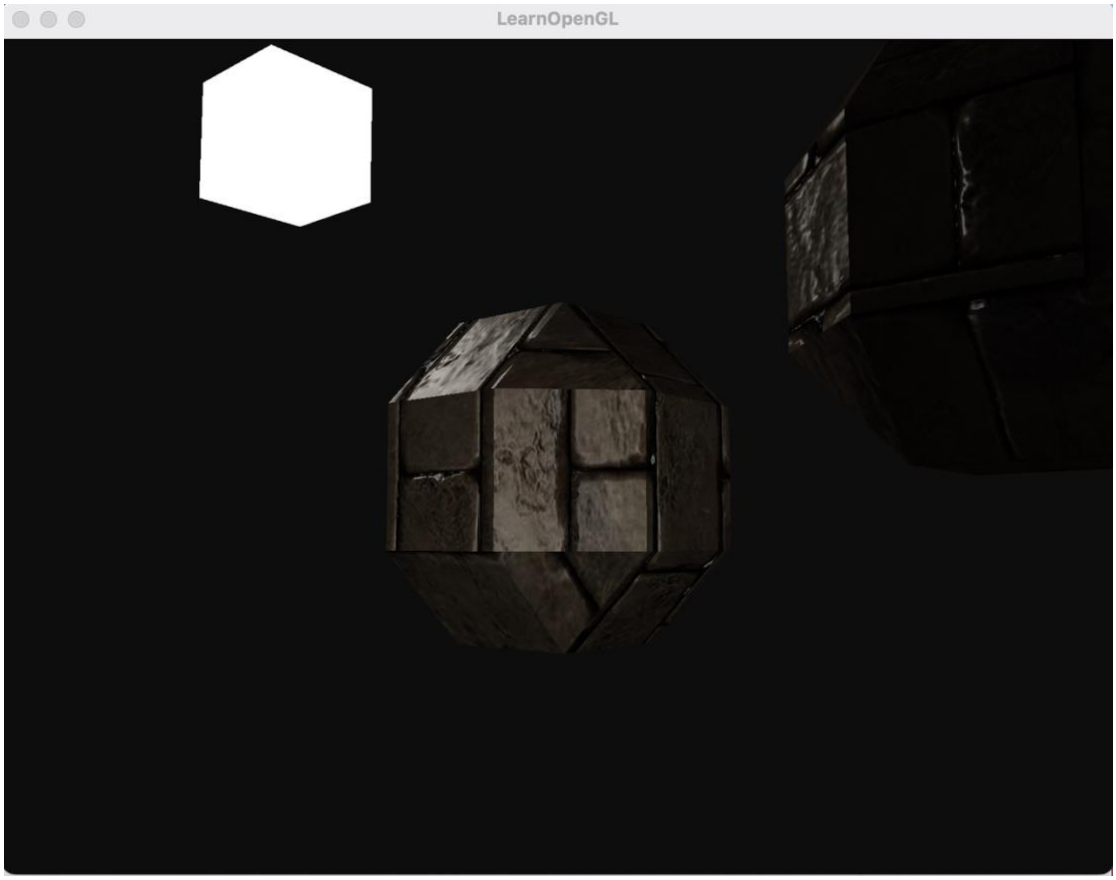

图 2.4 运行效果（2）

2.4 心得体会

我选修的计算机图形学课程是一门非常有意思的课程。在这门课程中，我学习了图形学的基础知识，例如三维图形的表示方法、图形变换、光照模型、渲染算法等。这些知识对于我理解和掌握图形学非常重要。

在课程的实践练习中，我使用了 opengl 库来绘制独孤信印章模型。我通过法线贴图实现了模型凹凸纹理。并使用点光源和跟随镜头的摄像头光源来照亮模型。在绘制过程中，我不仅学会了如何使用 opengl 库和 assimp 库，还学会了如何处理模型数据、如何进行图形变换、如何计算光照和如何渲染图形等。

在这门课程的学习中，我觉得最有意思的是能够将所学知识应用到实际的绘图中，看到自己绘制的图形逐渐变得逼真，感觉非常的兴奋。我也很感激这门课程的老师，在讲授基础知识的同时，也给出了大量的实践练习，帮助我们加深对知识的理解。

在这门课程的学习过程中，我还发现了许多我以前不知道的知识点，例如法线贴图的原理和实现方法，以及物理渲染的原理和实现方法。我觉得这些知识点非常有意思，对于我以后的学习和工作都有很大的帮助。

同时，我也觉得在这门课程的学习过程中，我的编程能力得到了很大的提升。我学会了如何使用 opengl 库和 assimp 库，以及如何处理模型数据、如何进行图

形变换、如何计算光照和如何渲染图形等。在这门课程中，我们还学习了 GLSL 语言和渲染管线。GLSL 是一种高级着色语言，可以用来编写 shader。Shader 是图形学中非常重要的一种组件，它可以在图形渲染过程中实现各种效果。例如，我们可以使用 shader 实现光照、阴影、反射等效果。通过利用 GLSL 编写 shader 实现 phong 模型也加强了我对光照模型的理解。

在学习 GLSL 和 shader 的过程中，我们还了解了渲染管线的工作原理。渲染管线是图形学中一种非常重要的概念，它描述了图形从输入到输出的整个流程。例如，我们可以使用渲染管线来实现三维图形的变换、光照计算和图形渲染等。

总的来说，我觉得 GLSL 和 shader 是图形学中非常有趣的知识点，它们可以帮助我们实现各种高级效果，并且学习这些知识点对于我以后的学习和工作都有很大的帮助。

2.5 源代码

这个项目代码全部代码托管在 GitHub 上，这里给出主程序 main.cpp 的代码作为示例。本项目在 MacOS 12.6 上完成的开发和测试，并可以直接移植到 windows 平台下构建和运行。仓库中已经给出了 windows 平台下预编译的可执行文件供检查和展示，可以直接下载并运行，也可以使用 cmake 直接自行构建。

项目的 github 仓库地址：

https://github.com/wojiaowenzhong233/hust_computer_graphics_homework。

程序 1：main.cpp

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <algo/shader.h>
#include <algo/camera.h>
#include <algo/model.h>
#include <algo/cube.h>
#include <iostream>

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);

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

// camera
```

```
Camera camera(glm::vec3(0.0f, 0.0f, 3.0f));
glm::vec3 pointPosition(-5.0f, 5.0f, 0.0f);

float lastX = SCR_WIDTH / 2.0f;
float lastY = SCR_HEIGHT / 2.0f;
bool firstMouse = true;

// timing
float deltaTime = 0.0f;
float lastFrame = 0.0f;

float current_scale = 1.0f;
float current_omega = glm::radians(50.0f);
float current_theta = 0.0f;

float current_fai = 0.0f;
float dfai_dt = glm::radians(30.f);

const auto axis = glm::normalize(glm::vec3 {-1, 1, 0});

struct draw_context {
    Model &model;
    Shader &shader;
    const glm::mat4 &view;
    const glm::mat4 &projection;
    glm::vec3 position;
    float scale;
    float theta;
};

void set_light(Shader& lightingShader);
void draw_model(const draw_context &context);

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);
#endif
```

```
    // glfw window creation
    // --------------------
    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT,
"LearnOpenGL", nullptr, nullptr);
    if (window == nullptr)
    {
        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;
    }

    // tell stb_image.h to flip loaded texture's on the y-axis (before loading model).
    stbi_set_flip_vertically_on_load(true);

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

    // build and compile shaders
    // -------------------------
    Shader ourShader("colors.vs", "colors.fs");
    Shader pointShader("light_cube.vs", "light_cube.fs");

    // load models
    // -----------
    Model ourModel("model/model.obj");
    cube point(pointPosition);

    // render loop
    // -----------
```

```
    while (!glfwWindowShouldClose(window))
    {
        // per-frame time logic
        // --------------------
        auto currentFrame = static_cast<float>(glfwGetTime());
        deltaTime = currentFrame - lastFrame;
        lastFrame = currentFrame;
        current_theta += current_omega * deltaTime;
        current_fai     += dfai_dt * deltaTime;

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

        // render
        // ------
        glClearColor(0.05f, 0.05f, 0.05f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        const glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom),
(float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
        const glm::mat4 view = camera.GetViewMatrix();
        const auto r = 3.0f;

        draw_context context{ourModel, ourShader, view, projection};

        context.position = glm::vec3(0.0f);
        context.theta       = current_fai;
        context.scale       = current_scale;
        draw_model(context);

        const auto position = glm::rotate(glm::mat4(1.0f), current_theta, axis) *
glm::vec4(0, 0, 1, 1);
//          context.position = glm::vec3(r * cos(current_theta), r * sin(current_theta), 0.0f);
        context.position = position * r;
        context.theta       = current_fai;
        context.scale       = 0.5f;
        draw_model(context);

        point.Draw(pointShader, view, projection);

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

```
    // 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)
{
    const auto stride = 1.0f;
    const auto rstride = 2.0f;

    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);
    if (glfwGetKey(window, GLFW_KEY_U) == GLFW_PRESS) {
        current_scale = min(2.0f, current_scale + stride * deltaTime);
    }
    if (glfwGetKey(window, GLFW_KEY_J) == GLFW_PRESS) {
        current_scale = max(0.3f, current_scale - stride * deltaTime);
    }
    if (glfwGetKey(window, GLFW_KEY_H) == GLFW_PRESS) {
        current_omega = min(4.0f, current_omega + rstride * deltaTime);
    }
    if (glfwGetKey(window, GLFW_KEY_K) == GLFW_PRESS) {
        current_omega = max(-4.0f, current_omega - rstride * deltaTime);
    }
    if (glfwGetKey(window, GLFW_KEY_M) == GLFW_PRESS) {
        dfai_dt = min(4.0f, dfai_dt + rstride * deltaTime);
    }
    if (glfwGetKey(window, GLFW_KEY_N) == GLFW_PRESS) {
        dfai_dt = max(-4.0f, dfai_dt - rstride * 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 xposIn, double yposIn)
{
    auto xpos = static_cast<float>(xposIn);
    auto ypos = static_cast<float>(yposIn);

    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(static_cast<float>(yoffset));
}

void set_light(Shader& lightingShader) {
    lightingShader.setVec3("light.position", camera.Position);
    lightingShader.setVec3("light.direction", camera.Front);
    lightingShader.setFloat("light.cutOff", glm::cos(glm::radians(12.5f)));
    lightingShader.setFloat("light.outerCutOff", glm::cos(glm::radians(17.5f)));
    lightingShader.setVec3("viewPos", camera.Position);
```

```
        // light properties
        lightingShader.setVec3("light.ambient", 0.1f, 0.1f, 0.1f);
        // we configure the diffuse intensity slightly higher; the right lighting conditions differ with each lighting method and environment.
        // each environment and lighting type requires some tweaking to get the best out of your environment.
        lightingShader.setVec3("light.diffuse", 0.8f, 0.8f, 0.8f);
        lightingShader.setVec3("light.specular", 0.5f, 0.5f, 0.5f);
        lightingShader.setFloat("light.constant", 1.0f);
        lightingShader.setFloat("light.linear", 0.09f);
        lightingShader.setFloat("light.quadratic", 0.032f);


        lightingShader.setVec3("point.position", pointPosition);
        lightingShader.setVec3("point.ambient", 0.05f, 0.05f, 0.05f);
        lightingShader.setVec3("point.diffuse", 0.8f, 0.8f, 0.8f);
        lightingShader.setVec3("point.specular", 1.0f, 1.0f, 1.0f);
        lightingShader.setFloat("point.constant", 1.0f);
        lightingShader.setFloat("point.linear", 0.045f);
        lightingShader.setFloat("point.quadratic", 0.0075f);
}

void draw_model(const draw_context &context) {
        // don't forget to enable shader before setting uniforms
        context.shader.use();
        set_light(context.shader);

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

        // render the loaded model
        glm::mat4 model = glm::mat4(1.0f);
        model = glm::translate(model, context.position); // translate it down so it's at the center of the scene
        model = glm::scale(model, glm::vec3(context.scale));    // it's a bit too big for our scene, so scale it down
        model = glm::rotate(model, context.theta, glm::vec3(0.0f, 1.0f, 0.0f));
        context.shader.setMat4("model", model);

        context.model.Draw(context.shader);
}
```

3 课后作业

从以下二项中选择一项完成。其中选择 A 满分 40 分，选择 B 满分 35 分。

选择 A：

利用 OpenGL 框架，设计一个独孤信印章模型动画。要求如下：

（1）构建独孤信印章模型，该模型可以通过指定的外接球半径控制大小。

（2）增加纹理（凹凸纹理），可以使用木纹纹理或者其它图片纹理。

（3）构建观察空间，使用三维投影对模型进行观察，要求模型能够进行自转，通过键盘可以控制模型的大小和转速。

（4）增加光照处理，光源可以设在左上方。

（5）在上面的基础上，构建两个印章模型，一大一小，大印章自转，小印章绕着大印章旋转，要求处理好消隐。

选择 B：

利用 OpenGL 框架，设计一个日地月运动模型动画，要求如下：

（1）运动关系正确，相对速度合理，且地球绕太阳，月亮绕地球的轨道不能在一个平面内。

（2）地球绕太阳，月亮绕地球可以使用简单圆或者椭圆轨道。

（3）对球体纹理的处理，至少地球应该有纹理贴图。

（4）增加光照处理，光源设在太阳上面。

（5）为了提高太阳的显示效果，可以在侧后增加一个专门照射太阳的灯。