

**《计算机图形学》第二小组第一次实验报告**

学 院： 软件学院

专 业： 软件工程

学生姓名： 余旺

学生姓名： 陈飞扬

学生姓名： 常高林

学生姓名： 农国凯

学生姓名： 张思宇

指导教师： 吴雨婷

**北京交通大学**

2025年4月

目 录

[1实验目的 4](#_Toc2084690130)

[2实验环境 4](#_Toc1611254951)

[2.1CMAKE 4](#_Toc1485908009)

[2.2GLFW 4](#_Toc30153442)

[2.3GLAD 4](#_Toc672276869)

[2.4GLM/EIGEN 4](#_Toc963463843)

[3实验原理 5](#_Toc13450739)

[3.1VAO和VBO 5](#_Toc1697889876)

[3.2着色器 5](#_Toc1861087728)

[3.3二维图形绘制 5](#_Toc2072519647)

[3.4二维图形变换 6](#_Toc1992896029)

[3.4.1缩放 6](#_Toc336499274)

[3.4.2反射 6](#_Toc1528288617)

[3.4.3切变 6](#_Toc245784515)

[3.4.4旋转 7](#_Toc1853019799)

[3.4.5平移 7](#_Toc424526380)

[3.5三维图形变换 7](#_Toc167360262)

[3.5.1缩放 7](#_Toc146642316)

[3.5.2旋转 8](#_Toc779182275)

[3.5.3平移 8](#_Toc1453873530)

[3.5.4正交投影 9](#_Toc1028615352)

[3.5.5透视投影 10](#_Toc183341305)

[4实验内容 12](#_Toc1517885765)

[4.1目录构建 12](#_Toc2081166412)

[4.2二维图形 12](#_Toc1955184303)

[4.2.1位移变换 13](#_Toc1949773317)

[4.2.2旋转变换 14](#_Toc1822865691)

[4.2.3缩放变换 14](#_Toc446346270)

[4.2.4反射变换 14](#_Toc31833340)

[4.2.5切变变换 15](#_Toc1848243203)

[4.2.6二维组合变化 15](#_Toc1257382213)

[4.3三维图形 15](#_Toc2116523471)

[4.3.1键盘控制 15](#_Toc1312014507)

[4.3.2缩放变换 19](#_Toc595806575)

[4.3.2旋转变换 19](#_Toc2146676913)

[4.3.2平移变换 19](#_Toc1984291376)

[4.3.3组合变换 19](#_Toc1559270418)

[4.3.4 摄像机移动 19](#_Toc12644004)

[5实验结果 20](#_Toc1534697604)

[5.1二维图像 20](#_Toc1272874498)

[5.1.1二维图形绘制 20](#_Toc2085163651)

[5.1.2二维图形变换 20](#_Toc1380109985)

[5.2三维图形绘制 23](#_Toc1609373772)

[5.2.1三维图形变换 24](#_Toc1465968621)

[5.2.2相机的移动 26](#_Toc1625894500)

[5.2.3透视投影 27](#_Toc1314909923)

[6实验总结和感想 28](#_Toc1890495001)

[7实验分工 29](#_Toc1793254763)

# 1实验目的

本次实验旨在通过计算机图形学编程实践，深化对几何变换理论体系及其工程实现的理解，培养理论推导与工程编码的双向转化能力。具体目标包括：

1.深入理解二维/三维几何变换（平移、旋转、缩放、反射、切变）及视图-投影变换的数学原理,并将其转化为实际代码。

2.对比齐次坐标系的数学推导（如组合变换的矩阵乘法顺序）与GLM库矩阵操作的实际差异。

3.学习并掌握相关图形API,如本次实验采取的OpenGL,通过多变换模式切换（键盘交互）和效果对比，建立变换参数与可视化结果的关联认知。

# 2实验环境

## 2.1CMAKE

CMake是一个跨平台的自动化构建系统,用于管理软件构建过程中的编译器等。CMake通过CMakeLists.txt配置文件来生成标准构建脚本。在本次实验中使用了CMake完成相关库的导入以及项目和库文件的编译等。

## 2.2GLFW

本次实验采取了GLFW,一个专门针对OpenGL的C语言库,它提供了一些渲染所需接口,允许用户自定义窗口参数、创建OpenGL上下文等。GLFW来自[[An OpenGL library | GLFW](https://www.glfw.org/)]。本实验中采取了GLFW提供的不同编译器版本以便成员能根据自身需求选择。

## 2.3GLAD

本次实验额外采用GLAD用于管理OpenGL扩展和核心API。为防止开发过程中直接从驱动程序加载所有函数造成复杂错误,引入GLAD作为加载库自动加载并管理函数指针,简化OpenGL初始化过程,降低开发难度。GLAD来自[[GLAD](https://glad.dav1d.de/)]。

## 2.4GLM/EIGEN

GLM是一个基于c++的数学库,本项目中使用GLM以便进行矩阵、向量、四元数等计算,同时GLM提供了一系列数学函数以及各种变换的实现,进一步提高项目开发效率,非常适用于进行图形相关开发。GLM可以在[[GLM](https://glm.g-truc.net/0.9.8/index.html)]获取。GLM已经对图形处理常用的变换操作做好了封装处理，但为了展示几种变换的内部细节，我们采用了线性代数EIGEN库,EIGEN是著名的纯头文件C++高性能线性代数库，他的导入十分简单，仅需引用相关头文件即可，同时EIGEN也具备较高的性能表现。

# 3实验原理

## 3.1VAO和VBO

VBO（顶点缓冲对象）  
VBO 是 OpenGL 中用于在 GPU 内存（显存）中高效存储顶点数据的对象。它允许开发者将顶点数据（如位置、颜色、纹理坐标等）一次性传输到 GPU，避免每次渲染时重复传输数据，从而显著提升性能。

VAO（顶点数组对象）  
VAO 是一个状态容器，用于保存顶点属性的配置（如顶点位置、法线、纹理坐标的绑定方式）。它记录了如何将 VBO 中的数据解析为顶点属性

## 3.2着色器

着色器是运行在 GPU 上的小程序，负责控制渲染管线的特定阶段。现代 OpenGL 使用可编程着色器替代固定功能管线，实现高度灵活的渲染效果。

**顶点着色器**

作为管线的第一阶段，负责处理输入的顶点数据。它接收来自顶点数组对象（VAO）的原始顶点属性（如位置、法线、纹理坐标），并通过矩阵变换（如模型、视图、投影矩阵）将顶点从局部空间转换到裁剪空间，最终输出顶点在屏幕上的位置（gl\_Position）。此外，顶点着色器还可计算光照、传递颜色或纹理坐标至后续阶段，甚至生成自定义变量（如varying变量）供片段着色器插值使用。例如，一个简单的顶点着色器可能将顶点颜色直接传递到片段阶段，或通过MVP矩阵实现3D场景的投影变换。

**片段着色器**

则在光栅化阶段后执行，负责计算每个像素（片段）的最终颜色。它接收顶点着色器输出的插值数据（如颜色、纹理坐标），并结合纹理采样、光照模型或后处理算法，决定片段的颜色值（如FragColor）。片段着色器是实现复杂视觉效果的核心，例如通过Phong光照模型计算表面反射光，或应用图像模糊、景深等后期效果。其输出将直接写入帧缓冲区，形成最终渲染图像。

## 3.3二维图形绘制

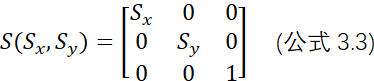
通过顶点缓冲对象（VBO）定义三角形顶点数据（位置、颜色、纹理坐标），使用着色器（Shader）进行渲染，结合元素缓冲对象（EBO）指定绘制顺序。

## 3.4二维图形变换

### 3.4.1缩放

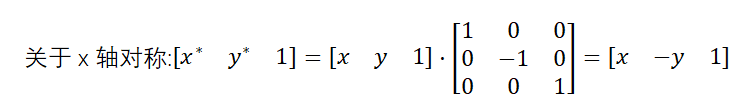
缩放变换是一种用于改变对象大小的几何变换。在二维空间中,缩放可以沿着X、Y轴进行,下方给出了变换矩阵,其中IMG_256

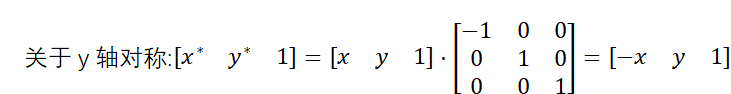
表示X、Y轴上的缩放因子

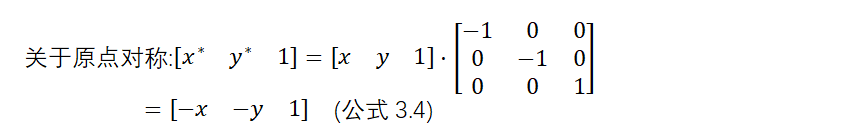


### 3.4.2反射

反射变换是一种将当前图形进行周对称几何变换的方式。在二维空间中,反射可以沿着X、Y轴以及原点进行,下方给出了变换过程。

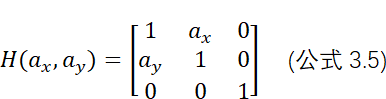






### 3.4.3切变

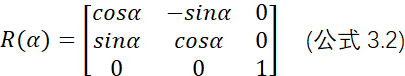
切变变换,也被称为剪切变换,是一种线性映射,只会改变物体形状而不会改变面积,下方将给出变换矩阵。



### 3.4.4旋转

旋转变化只改变图形方向,不改变大小或形状。一般旋转变化主要围绕原点进行。

下方给出了二维旋转变化所需的其次矩阵。



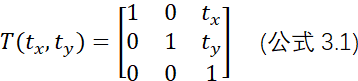
### 3.4.5平移

位移变换是一种基本几何变化,唯一不会改变几何体的大小、形状,只会改变几何体位置。

位移变换可以使用三维的位移向量来描述。将位移向量置入齐次坐标矩阵最后一列即可实现位移操作,下方公式给出了X,Y上位移所需的其次矩阵

IMG_256

分别表示X、Y轴上的位移量。

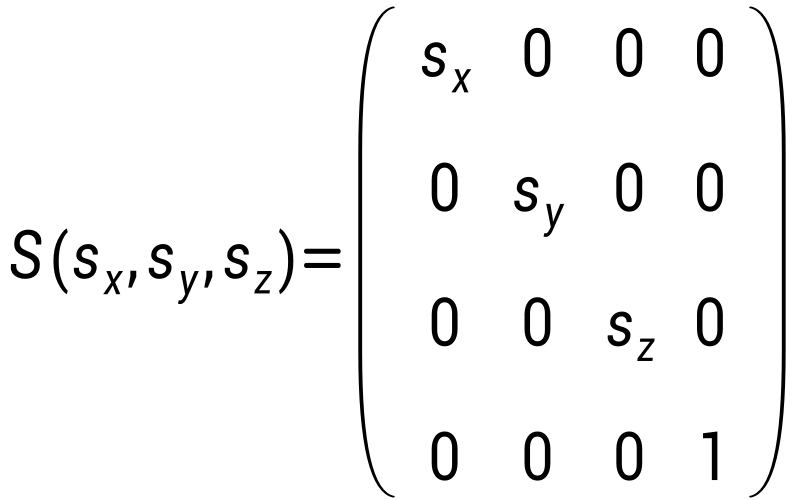


## 3.5三维图形变换

三维变换可以通过类比二维变换来进行，在三维空间中可以用（x,y,z,1)T来表示一个点，而用（x,y,z,0)T来表示一个向量

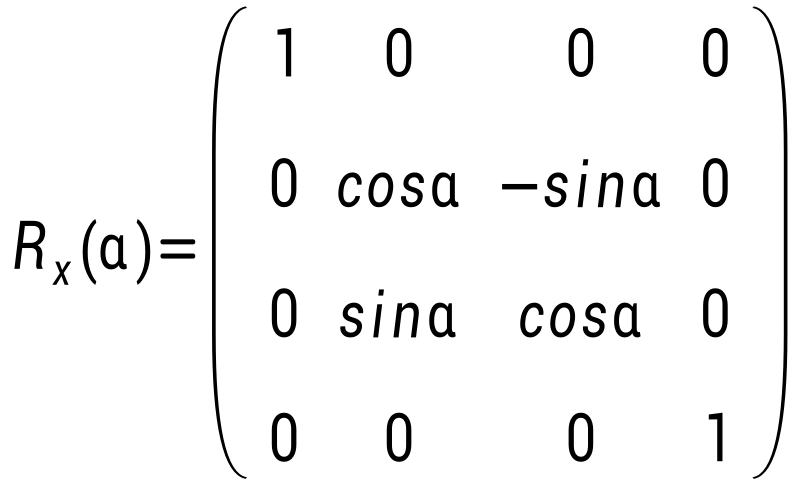
### 3.5.1缩放

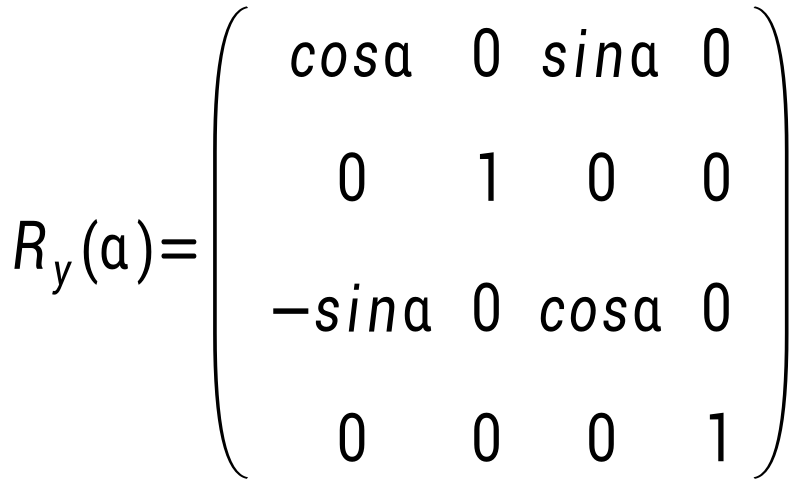
缩放变换描述的是将物体沿着某个轴线对物体的大小进行调整，在齐次坐标形式下的变换矩阵如下所示，其中Sx,Sy,SZ.分别表示沿着x,y,z的缩放因子。

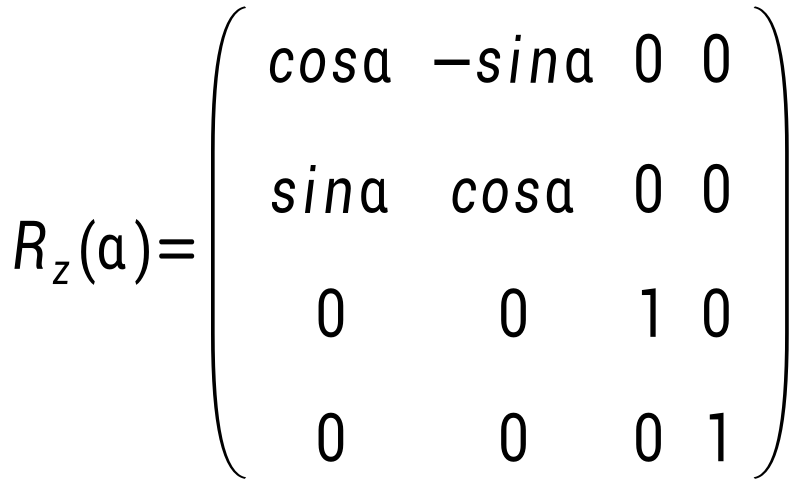


### 3.5.2旋转

和二维类似，旋转变换并不会改变物体的大小和形状，仅仅会改变物体的朝向，通常用θ来表示物体旋转的角度，根据物体旋转的旋转轴的不同，主要有三种旋转变换矩阵，绕x轴，绕y轴，绕z轴，表示如下

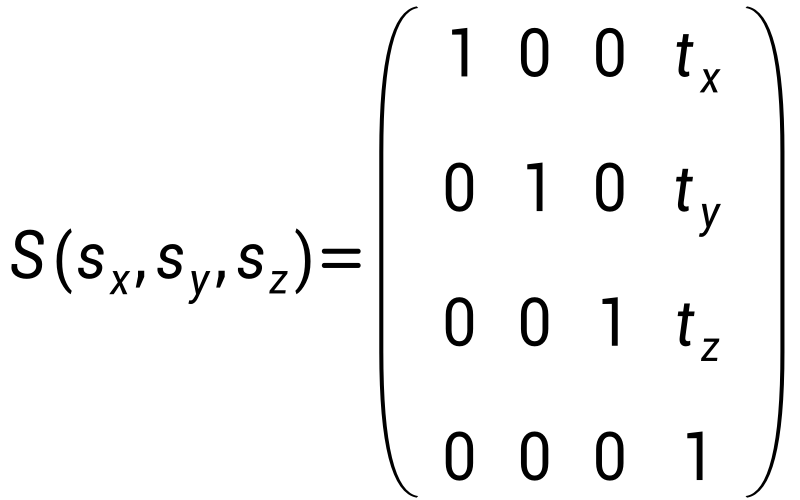






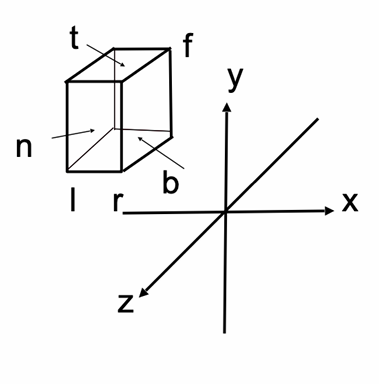
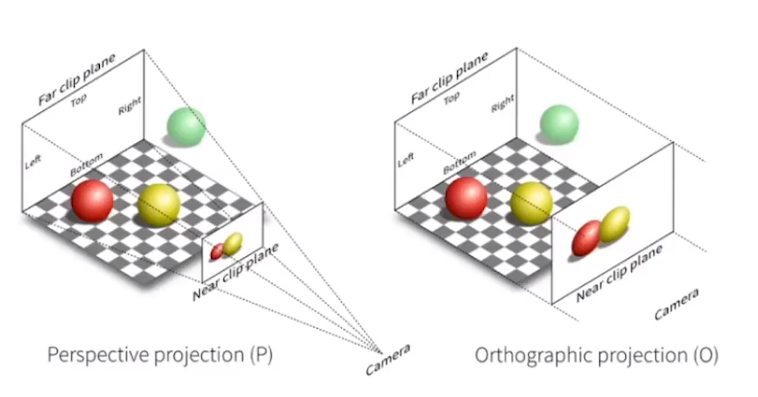
### 3.5.3平移

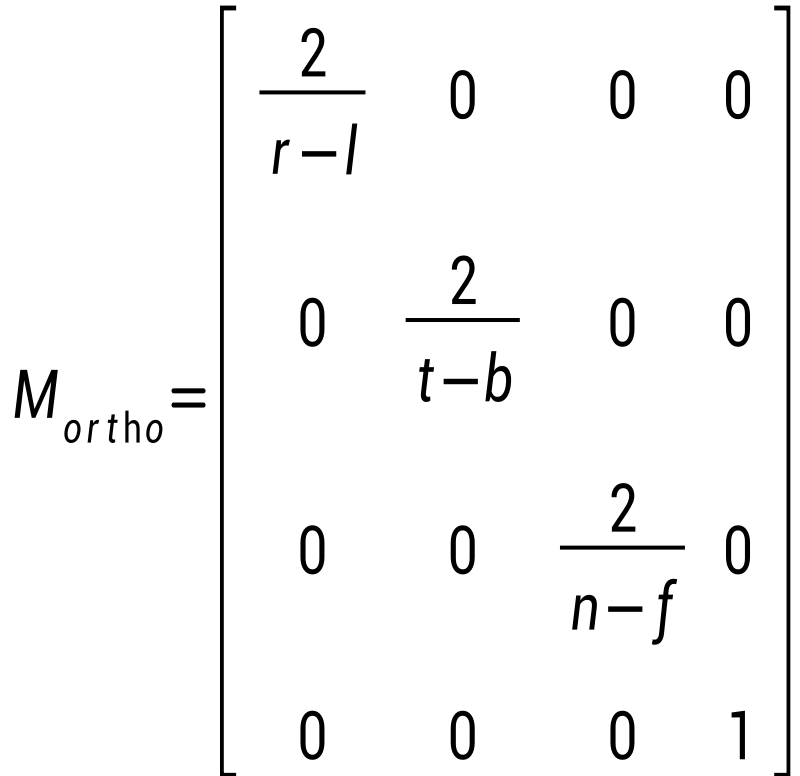
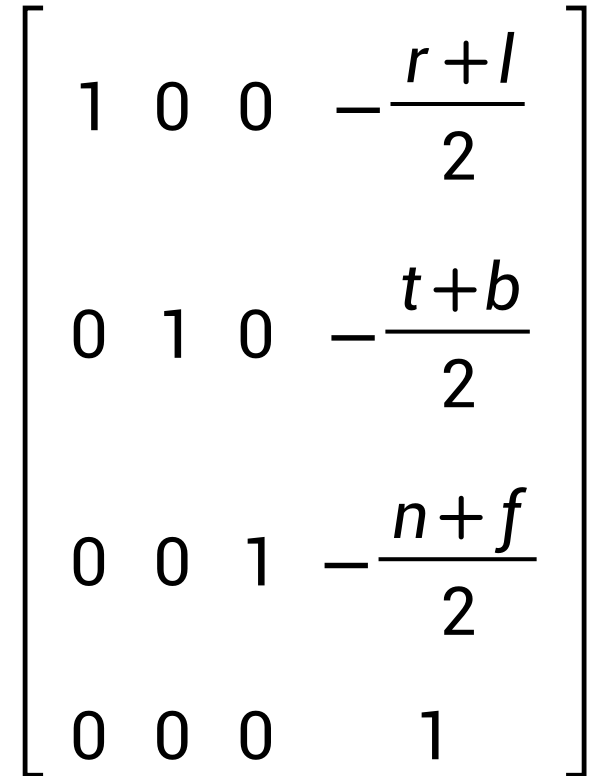
平移变换就相对更简单了，只是将图形沿着某个方向移动，其变换矩阵如下，其中tx,ty,tz分别表示沿着x轴。y轴和z轴移动的距离。



### 3.5.4正交投影

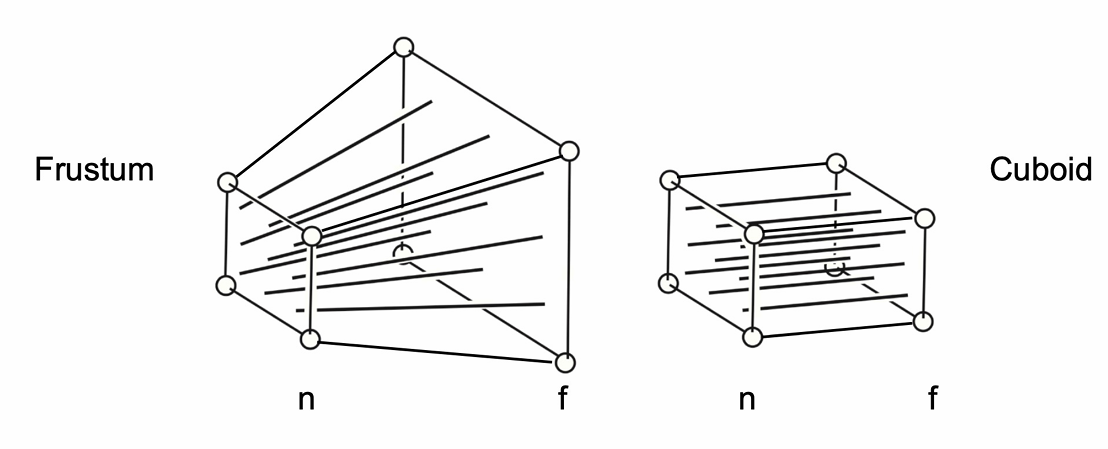
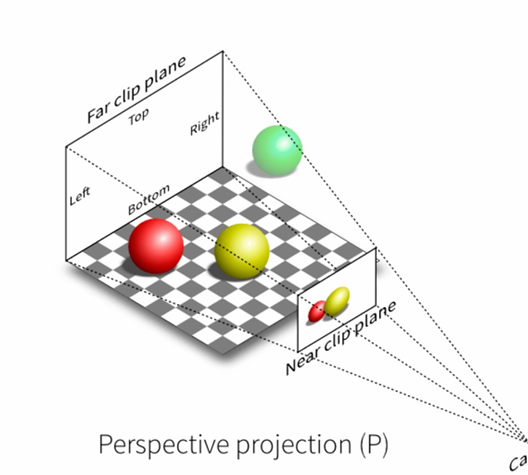
正交投影的特点是能够保持物体的平行，不会参数近大远小的视觉效果，计算正交投影矩阵前，首先需要定义这个平行于观察平面的正方体的六个面，也就是左（l)，右(r)，上(t)，下(b)，远(f)，近(n)，然后我们需要将物体从远端移动到坐标原点，然后再将其缩放到1\*1\*1大小下的立方体中，所有共有两个变换矩阵。

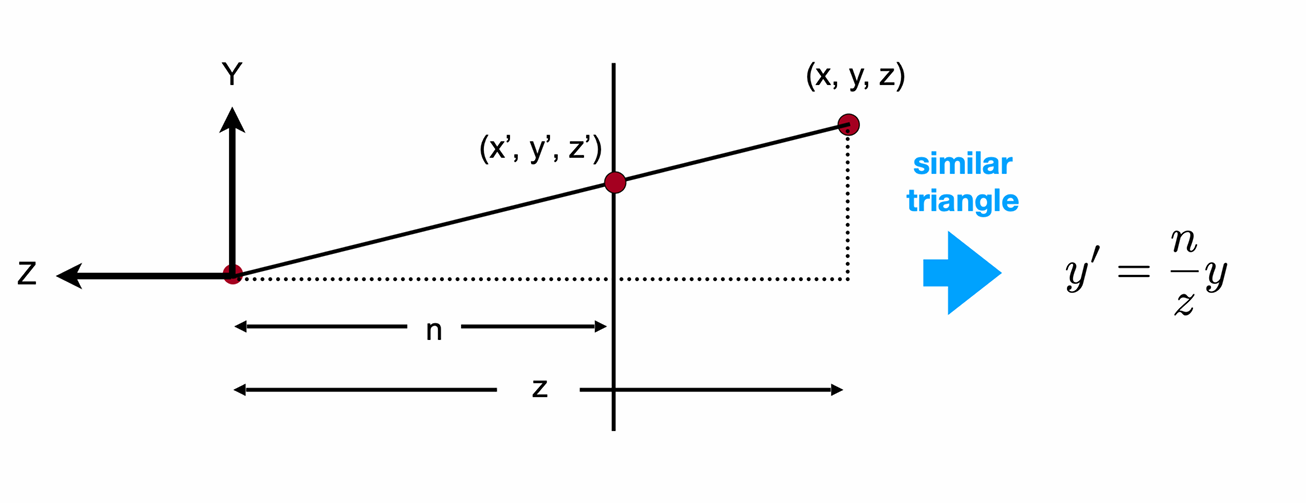


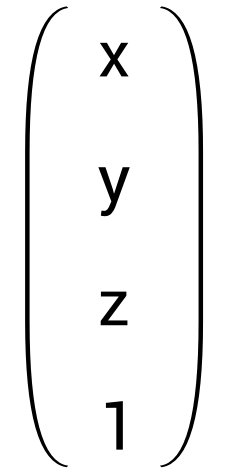
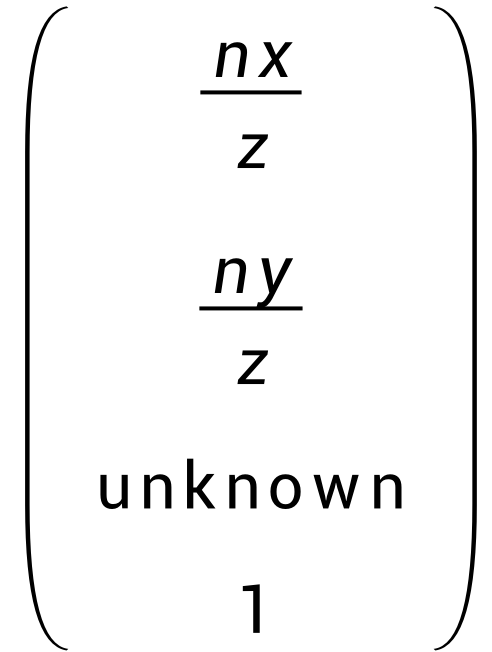
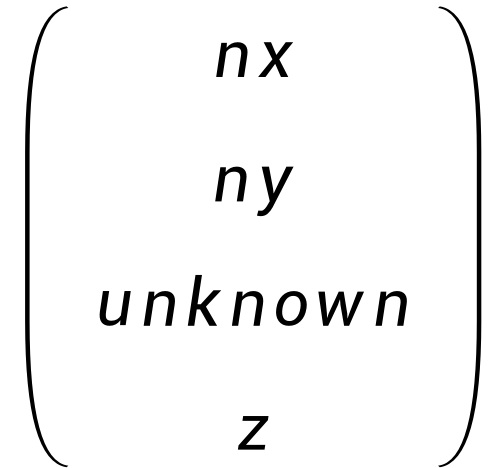
### 3.5.5透视投影

透视投影则更接近于人类观察世界的视角，也就是近大远小，具有真实的效果，我们用一个视椎体来表示这个可见空间，相对来说其变换过程复杂一些，思路是将透视投影“挤成”正交投影。



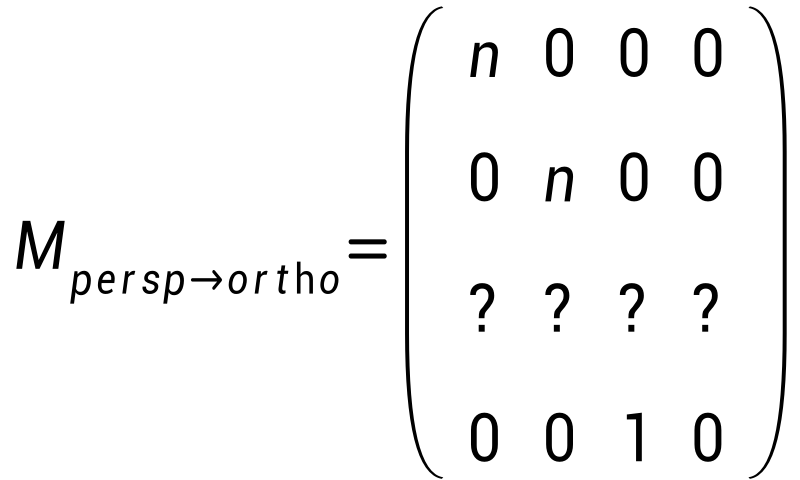


利用相似三角形，我们就可以得到相似比，从而计算变换矩阵。

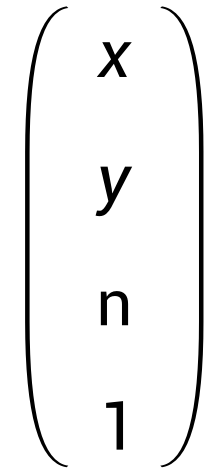
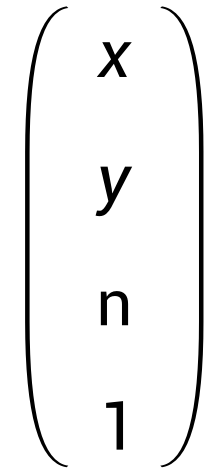
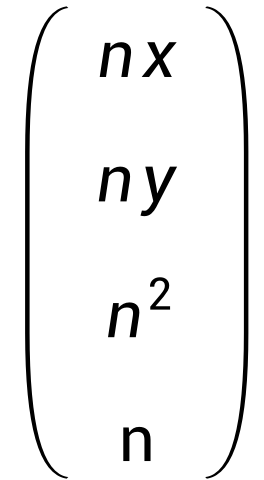
  同乘z 

sss

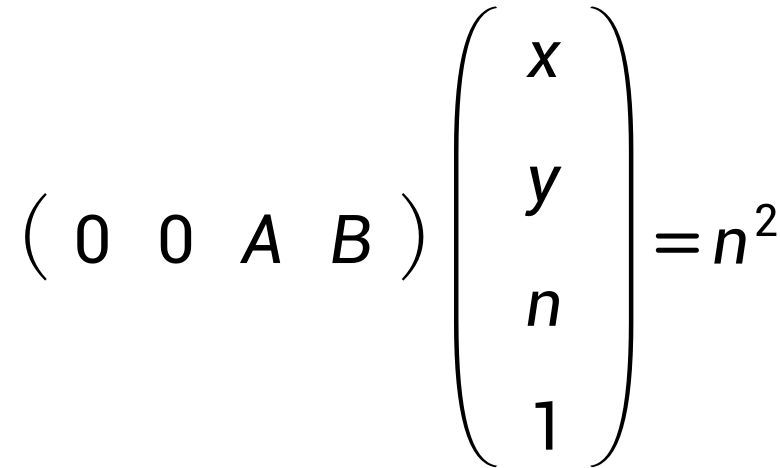
所以我们可以由最终的结果得到部分变换矩阵的值



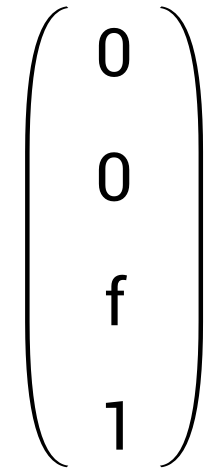
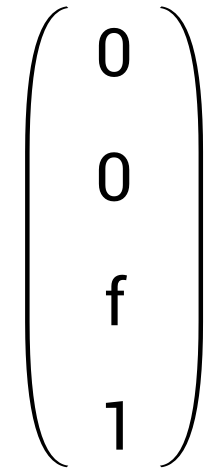
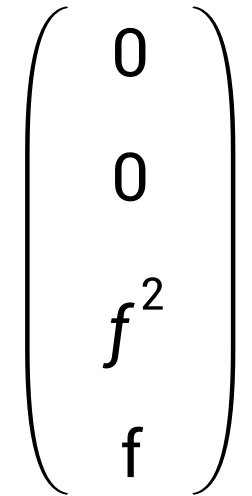
并且由于在近平面的点不会发生改变，同时在远平面的点的z值也不会改变，

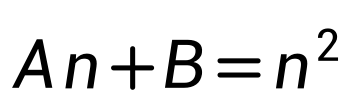
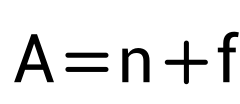
此时可以获得一个方程,有关第三行的未知数A,B，

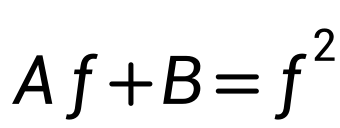
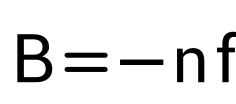


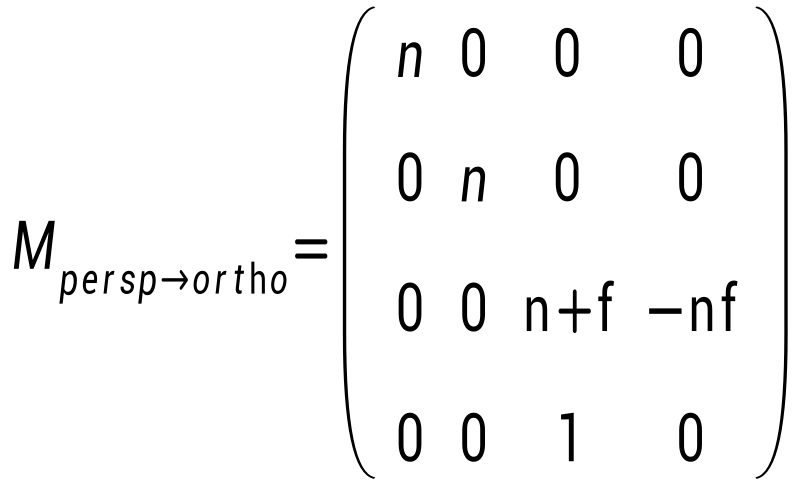
又根据远平面的点z值不变，

又得到一个方程，将两个方程联立



# 4实验内容

## 4.1目录构建

这里展示的目录结构以小组成员余旺同学的目录结构为例子，和另一名同学的不尽相同

**application**

在本次实验我们用于存放应用程序相关的代码和资源 ，包含创建窗口、处理用户输入、设置 OpenGL 上下文等与整体应用逻辑相关的代码。此外camera相关的代码放在这里

**assets**

存放资源文件，像纹理图片、模型文件（如.obj、.fbx 等格式，若项目涉及三维场景搭建 ）、着色器代码文件（.vert 顶点着色器、.frag 片段着色器 ）等。

**glframework**

可能是自定义的 OpenGL 框架代码。可能封装了 OpenGL 的基础操作，比如封装了创建和管理 OpenGL 缓冲区（顶点缓冲区、索引缓冲区 ）、纹理对象的函数，以及对 OpenGL 状态机（如深度测试、混合模式 ）进行设置的工具函数等，方便在项目中复用。

**out**

一般是构建输出目录 。编译项目后，生成的可执行文件、中间目标文件、链接库等会放在这里。例如，使用 CMake 构建基于 OpenGL 的项目时，可执行文件可能就输出到这个目录。

**thirdParty**

存放第三方库相关内容。在 OpenGL 项目中，可能会用到像 GLFW（用于创建窗口和处理输入 ）、GLEW（用于加载 OpenGL 函数指针 ）、ASSIMP（用于导入模型 ）等第三方库，它们的头文件、库文件（.lib 或.so 等 ）等会放在此目录。

**wrapper**

这主要存放错误处理相关代码

## 4.2二维图形

在二维图形绘制后,我们可以直接对transform矩阵进行操控,以实现不同的变换,由于OpenGL中只有三维概念,所以我们将Z坐标固定为0以实现二维变换。

在项目中我们采用监听键盘输入的方式展现变换,项目初始化时三角形将默认处于旋转状态。

void OnKey(int key, int action, int mods) {  
 if (action == GLFW\_PRESS) {  
 switch (key) {  
 case GLFW\_KEY\_1: transformType = 1; break; // 缩放  
 case GLFW\_KEY\_2: transformType = 2; break; // 反射  
 case GLFW\_KEY\_3: transformType = 3; break; // 切变  
 case GLFW\_KEY\_4: transformType = 4; break; // 平移  
 case GLFW\_KEY\_5: transformType = 5; break; // 组合变换  
 case GLFW\_KEY\_6: transformType = 6; break; // 复位  
 default: transformType = 0; // 持续旋转  
 }  
 }  
}

### 4.2.1位移变换

通过Onkey()函数监听用户键盘输入,当用户键入4时,在main()函数中将对用户输入进行响应。

while (app->update()) {  
 switch (transformType) {  
 case 1: doScaleTransform(); break;  
 case 2: doReflectionTransform(); break;  
 case 3: doShearTransform(); break;  
 case 4: doTranslationTransform(); break;  
 case 5: doCompositeTransform(); break;  
 case 6: transform = glm::mat4(1.0f); break;  
 default: doRotation(); // 默认持续旋转  
 }  
 render();  
 }

其中对应位移变换为

void doTranslationTransform() {  
 transform = glm::translate(glm::mat4(1.0f), glm::vec3(0.5f, 0.0f, 0.0f));  
}

在函数中我们直接对shader绑定的transform矩阵进行操作,用GLM库的translate方法生成变换矩阵,glm::mat4(1.0f)表示创建一个4\*4单位矩阵,glm::vec3(0.5f, 0.0f, 0.0f)表示位移向量,由此得到所需的变换矩阵并传入transform,由于transform已经与顶点着色器绑定,着色器将自动应用于绘制出的点,具体过程如下代码。

void main()  
{  
 vec4 position = vec4(aPos, 1.0);  
 position = transform \* position;  
 gl\_Position = position;  
 color = aColor;  
 uv = aUV;  
}

### 4.2.2旋转变换

同样监听用户输入进行相应,因4.2.1中已进行过程介绍,故此处简要讲解。

当用户输入对应数字后,将执行以下代码

float angle = 0.0f;  
void doRotation() {  
 angle += 2.0f;  
 //每一帧都会“重新”构建一个旋转矩阵  
 transform = glm::rotate(glm::mat4(1.0f), glm::radians(angle), glm::vec3(0.0, 0.0, 1.0));  
}

声明一个angle表示角度,并调用GLM的rotate以弧度制传入angle生成旋转矩阵,并由transform传递至着色器以应用。

### 4.2.3缩放变换

缩放变换实现代码如下

//缩放变换  
void doScaleTransform() {  
 transform = glm::scale(glm::mat4(1.0f), glm::vec3(0.1f, 0.5f, 1.0f));  
}

同样采用GLM库的方法生成变换矩阵并交由着色器实现变换。

### 4.2.4反射变换

反射变换实现如下

// 反射变换函数  
void doReflectionTransform() {  
 transform = glm::scale(glm::mat4(1.0f), glm::vec3(-1.0f, 1.0f, 1.0f));  
}

### 4.2.5切变变换

切变变换实现如下

void doShearTransform() {  
 glm::mat4 shearMat = glm::mat4(1.0f);  
 shearMat[1][0] = 0.5f; // X方向剪切  
 transform = shearMat;  
}

此处实现有所不同,此处先生成了4\*4的单位矩阵,然后人为调整矩阵进行x方向剪切,需要注意的是GLM库中的矩阵是烈主序,所以(1，0)实际上对应了x剪切

### 4.2.6二维组合变化

二维组合变化只需要按照自右向左的顺序进行矩阵变换即可,具体操作便是按照一定顺序调用相关GLM函数,最后赋值给transform,交由着色器处理,比如以下代码展示了先移动,再旋转,最后缩放为0.7f.

void doCompositeTransform() {  
 glm::mat4 trans = glm::translate(glm::mat4(1.0f), glm::vec3(0.3f, 0.0f, 0.0f));  
 glm::mat4 rot = glm::rotate(trans, glm::radians(45.0f), glm::vec3(0.0f, 0.0f, 1.0f));  
 transform = glm::scale(rot, glm::vec3(0.7f, 0.7f, 1.0f));  
}

## 4.3三维图形

### 4.3.1键盘控制

类似地，在三维部分，我们也使用键盘对物体进行控制。

void processInput(GLFWwindow \*window) {

    if (glfwGetKey(window, GLFW\_KEY\_ESCAPE) == GLFW\_PRESS)//esc键 程序退出

        glfwSetWindowShouldClose(window, true);

    if (glfwGetKey(window, GLFW\_KEY\_LEFT\_ALT) == GLFW\_PRESS) //左alt键 增加第二个纹理的显示比例

{

        mixValue += 0.01f;

if (mixValue >= 1.0f)

            mixValue = 1.0f;

    }

    if (glfwGetKey(window, GLFW\_KEY\_RIGHT\_ALT) == GLFW\_PRESS) //右alt键 降低第二个纹理的显示比例

{

        mixValue -= 0.01f;

        if (mixValue <= 0.0f)

            mixValue = 0.0f;

    }

    if (glfwGetKey(window, GLFW\_KEY\_UP) == GLFW\_PRESS) //上键 纹理向上移动

{

        offset\_v += 0.01f;

        if (offset\_v >= 1.0f)

            offset\_v = offset\_v - 1.0f;

    }

    if (glfwGetKey(window, GLFW\_KEY\_DOWN) == GLFW\_PRESS) //下键 纹理向下移动

{

        offset\_v -= 0.01f;

        if (offset\_v <= 0.0f)

            offset\_v = 1.0f - offset\_v;

    }

    if (glfwGetKey(window, GLFW\_KEY\_RIGHT) == GLFW\_PRESS) //右键 纹理向右移动

{

        offset\_u += 0.01f;

        if (offset\_u >= 1.0f)

            offset\_u = offset\_u - 1.0f;

    }

    if (glfwGetKey(window, GLFW\_KEY\_LEFT) == GLFW\_PRESS) //左键 纹理向左移动

{

        offset\_u -= 0.01f;

        if (offset\_u <= 0.0f)

            offset\_u = 1.0f - offset\_u;

    }

    if (glfwGetKey(window, GLFW\_KEY\_PAGE\_DOWN) == GLFW\_PRESS) //Page up键 放大纹理

{

        scaleValue += 0.01f;

        if (scaleValue >= 1.0f)

            scaleValue = 1.0f;

    }

    if (glfwGetKey(window, GLFW\_KEY\_PAGE\_UP) == GLFW\_PRESS) //Page down键 缩小纹理

{

        scaleValue -= 0.01f;

        if (scaleValue <= 0.0f)

            scaleValue = 0.0f;

    }

    if (glfwGetKey(window, GLFW\_KEY\_X) == GLFW\_PRESS) //X键 绕X轴转动

{

        angle\_X -= 5.f;

        if (angle\_X <= 0.0f) {

            angle\_X = 360 + angle\_Y;

        }

    }

    if (glfwGetKey(window, GLFW\_KEY\_Y) == GLFW\_PRESS) //Y键 绕Y轴转动

{

        angle\_Y -= 5.f;

        if (angle\_Y <= 0.0f) {

            angle\_Y = 360 + angle\_Y;

        }

    }

    if (glfwGetKey(window, GLFW\_KEY\_Z) == GLFW\_PRESS) //Z键 绕Z轴转动

{

        angle\_Z -= 5.f;

        if (angle\_Z <= 0.0f) {

            angle\_Z = 360 + angle\_Y;

        }

    }

    if (glfwGetKey(window, GLFW\_KEY\_KP\_5) == GLFW\_PRESS) //5键 沿Y轴正方向移动

{

        position[1] += 0.01;

    }

    if (glfwGetKey(window, GLFW\_KEY\_KP\_2) == GLFW\_PRESS) //2键 沿Y轴负方向移动

{

        position[1] -= 0.01;

    }

    if (glfwGetKey(window, GLFW\_KEY\_KP\_1) == GLFW\_PRESS) //1键 沿X轴正方向移动

{

        position[0] += 0.01;

    }

    if (glfwGetKey(window, GLFW\_KEY\_KP\_3) == GLFW\_PRESS) //3键 沿X轴负方向移动

{

        position[0] -= 0.01;

    }

    if (glfwGetKey(window, GLFW\_KEY\_W) == GLFW\_PRESS) //W键 向上移动摄像机

{

        camera.MoveForward(-0.1f);

}

    if (glfwGetKey(window, GLFW\_KEY\_S) == GLFW\_PRESS) //S键 向下移动摄像机

      {

camera.MoveForward(0.1f);

}

    if (glfwGetKey(window, GLFW\_KEY\_A) == GLFW\_PRESS) //A键 向左移动摄像机

{

          camera.MoveRight(-0.1f);

}

    if (glfwGetKey(window, GLFW\_KEY\_D) == GLFW\_PRESS) //D键 向右移动摄像机

{

          camera.MoveRight(0.1f);

}

if (glfwGetKey(window, GLFW\_KEY\_SPACE) == GLFW\_PRESS) //空格键 向上移动摄像机

{

        camera.MoveRise(+0.1f);

    }

    if (glfwGetKey(window, GLFW\_KEY\_LEFT\_CONTROL) == GLFW\_PRESS) //左ctrl 向下移动摄像机

{

        camera.MoveRise(-0.1f);

    }

}

### 4.3.2缩放变换

当用户按下Page up或Page down 键后，scaleValue的值被改变，通过shader1.setFloat("scaleValue", scaleValue)传递给着色器，然后通过公式vec2 scaledTexCoord = (TexCoord - 0.5) / scaleValue + 0.5 + offset，将纹理坐标移动回原点，然后调整缩放比例，再讲坐标移动回原来的位置。

### 4.3.2旋转变换

当用户按下X键或Y键或Z键后,对应angle的值减少15°，在GetXRotationMatrix4f（angle)中生成变换矩阵，最后与投影矩阵相乘得到最终的变换矩阵，并传递给着色器。

transMatrix = ProjectionMatrix \* modelMatrix4f;

glUniformMatrix4fv(transformLoc, 1, GL\_FALSE, transMatrix.data());

### 4.3.2平移变换

当用户按下上、下、左、右键时，将会改变position向量中的X/Y值，之后在GetMoveMatrix(position)应用修改后的position,然后和投影矩阵相乘就得到了最终的变换矩阵transMatrix = ProjectionMatrix \* modelMatrix4f; 最后在着色器中应用变换glUniformMatrix4fv(transformLoc, 1, GL\_FALSE, transMatrix.data());

### 4.3.3组合变换

将前述各种变换的变换矩阵进行矩阵乘法，最后和投影矩阵再相乘，得到最终变换矩阵，将变换矩阵应用于着色器，就可以实现组合变换

### 4.3.4 摄像机移动

当按下W/S/A/D键时，调用camera.MoveForward和MoveRight方法，将摄像机向对应方向移动0.1个单位，当按下Space和Ctrl时,将会使用MoveRise上下移动摄像机，鼠标控制通过 GLFW 的回调函数实现，用于调整摄像机的朝向，每次回调记录鼠标的x,y值从而计算位移，根据位移调整摄像头的水平和垂直旋转。

# 5实验结果

## 5.1二维图像

**5.1.1二维图形绘制**

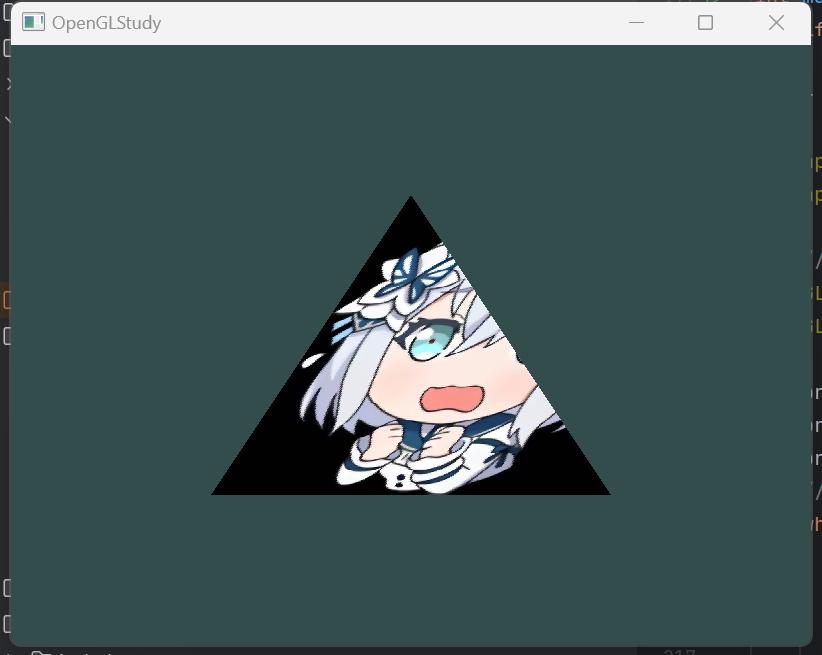


图5-1二维图像绘制

**5.1.2二维图形变换**

反射变化：通过glm::scale函数，将物体在 x 轴方向进行反射（乘以 -1），y 轴和 z 轴方向保持不变。

平移变换：glm::translate函数会生成一个平移矩阵，将物体在 x 轴方向向右平移 0.5 个单位。

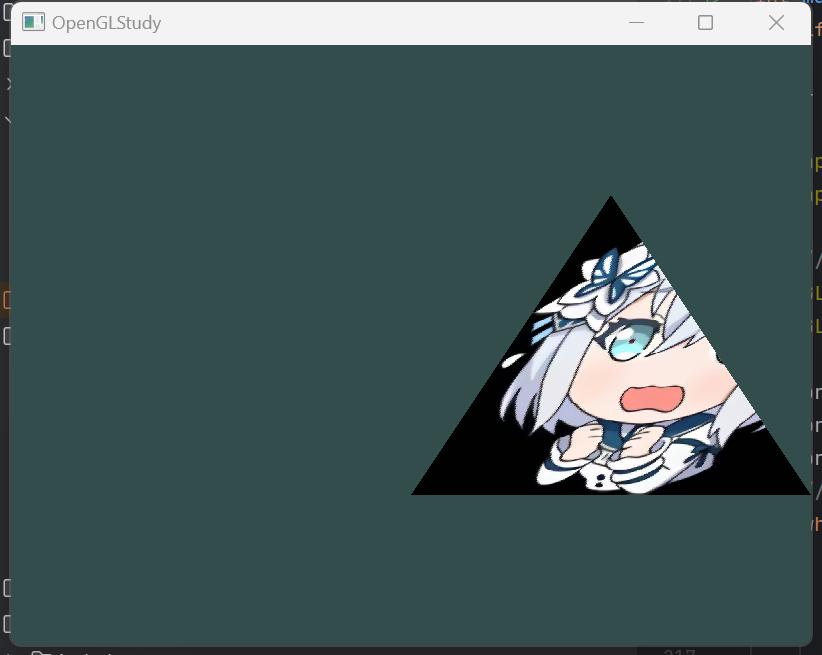


图5-2平移变换

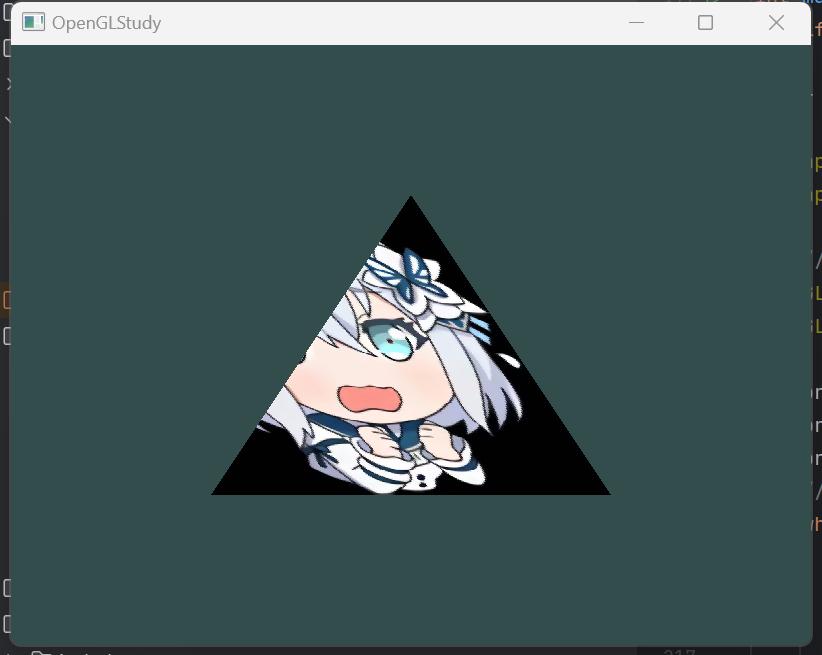


图5-3反射变换

**缩放变化**glm::scale函数会生成一个缩放矩阵，将物体在 x 轴方向缩小为原来的 0.1 倍，在 y 轴方向缩小为原来的 0.5 倍，z 轴方向保持不变。

**切边变换**：手动修改单位矩阵的元素shearMat[1][0]，实现 x 轴方向的剪切变换

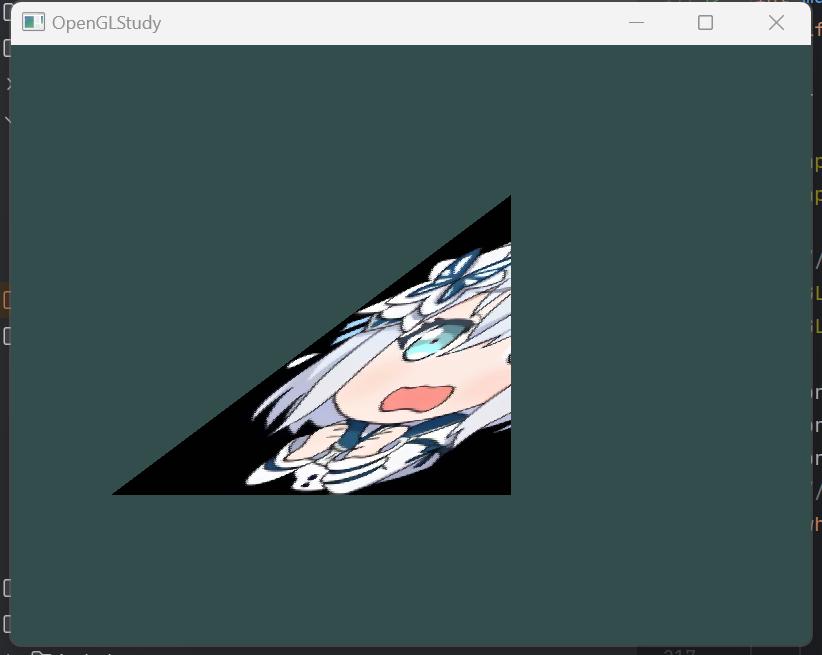


图5-4切边变换

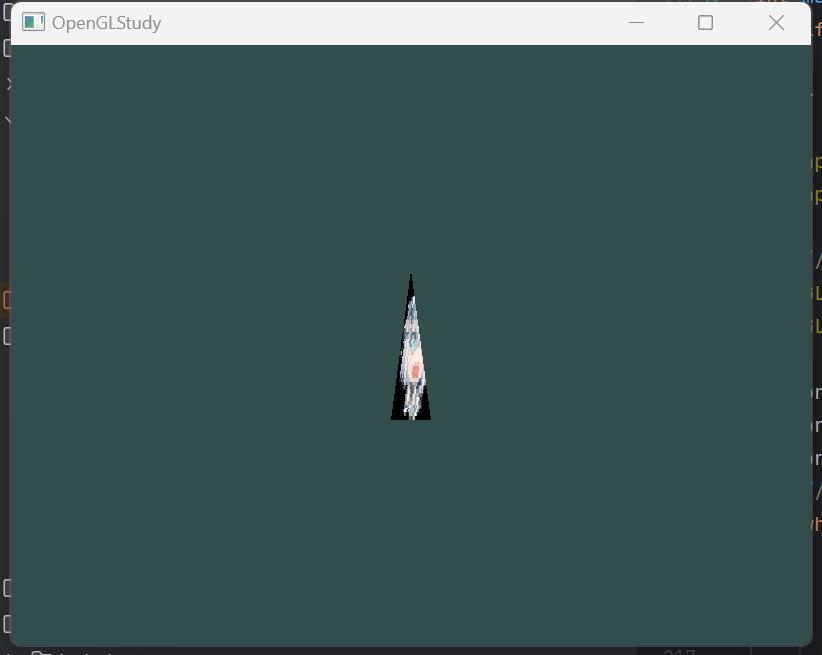


图5-5缩放变换

**旋转变换**：每帧angle都会增加 2 度，glm::rotate函数会生成一个绕z轴旋转angle弧度的旋转矩阵。

**组合变换**：先进行平移变换，将物体在 x轴方向向右平移 0.3 个单位。接着绕z轴旋转 45 度。最后在 x轴和 y轴方向进行缩放，缩放因子为 0.7。

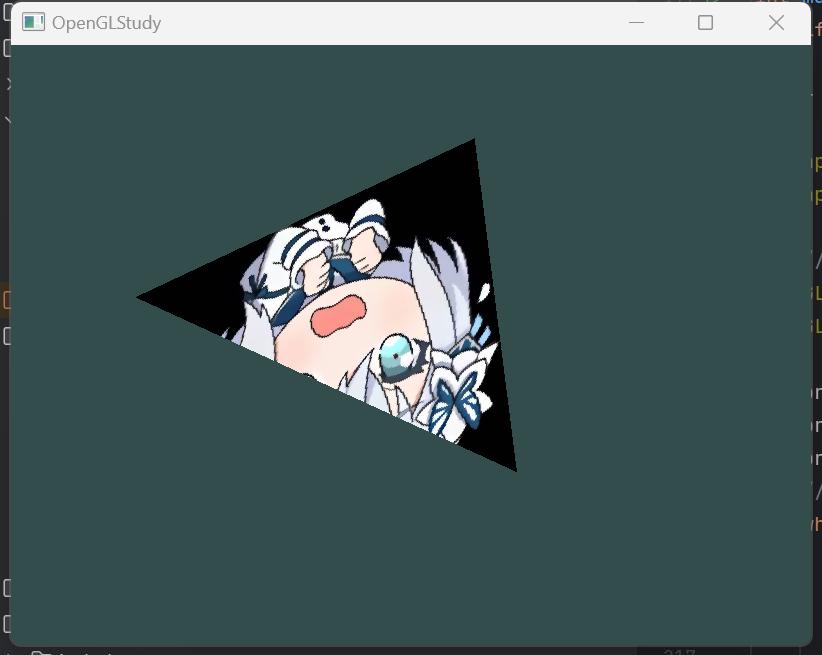


图5-6旋转变换

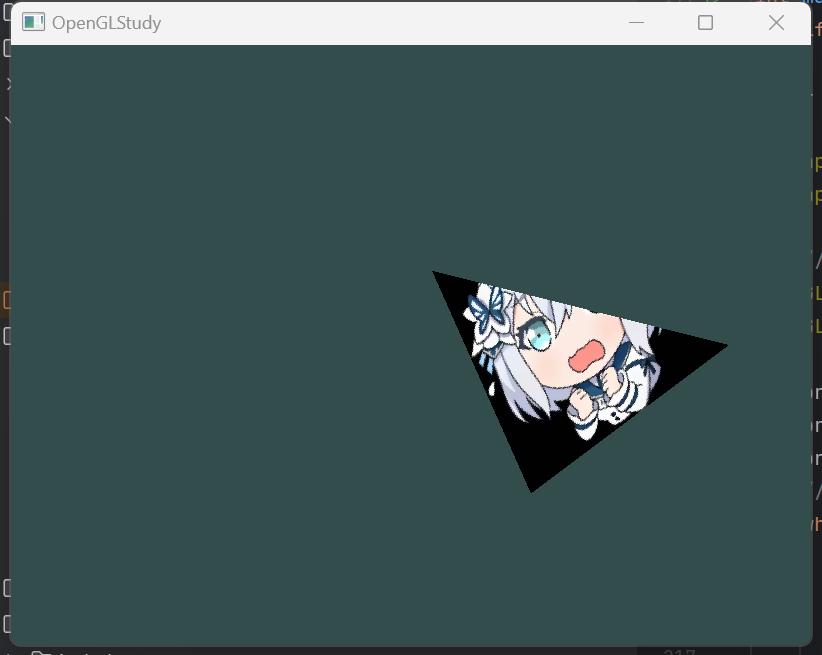
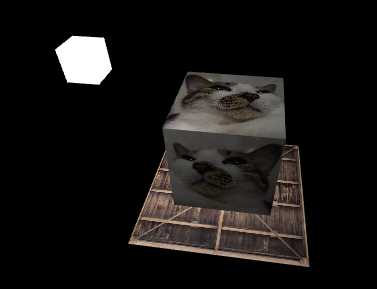


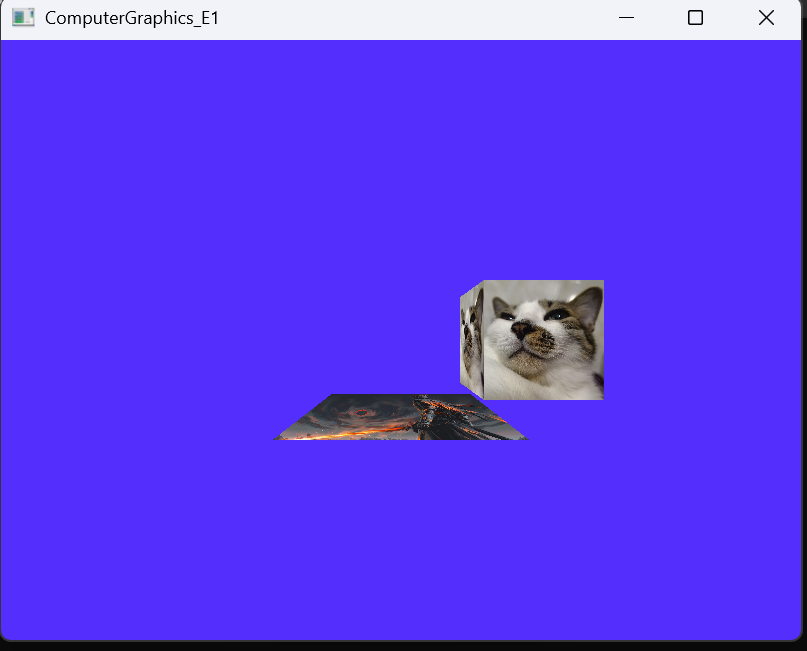
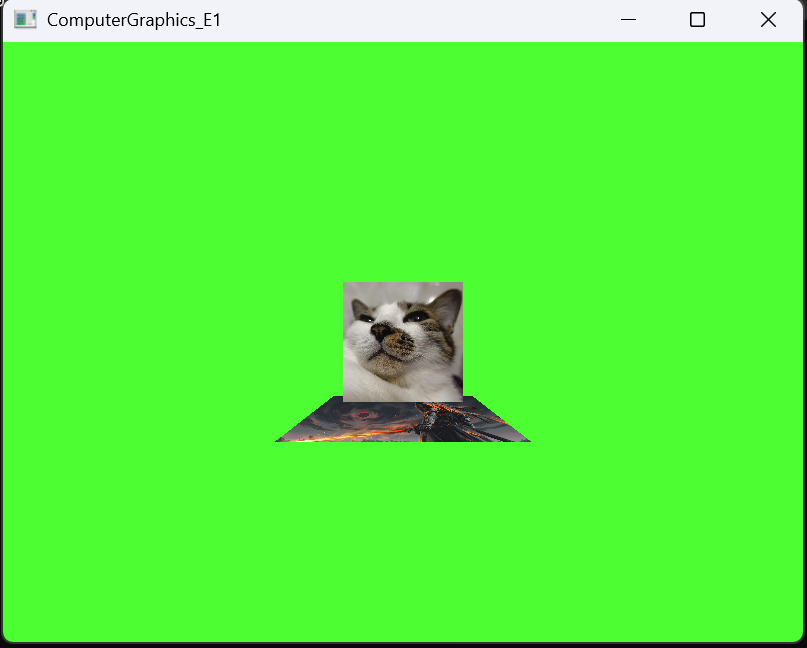
图5-7组合变换

## 5.2三维图形绘制

三维光照效果



### 5.2.1三维图形变换



平移变换



旋转变换

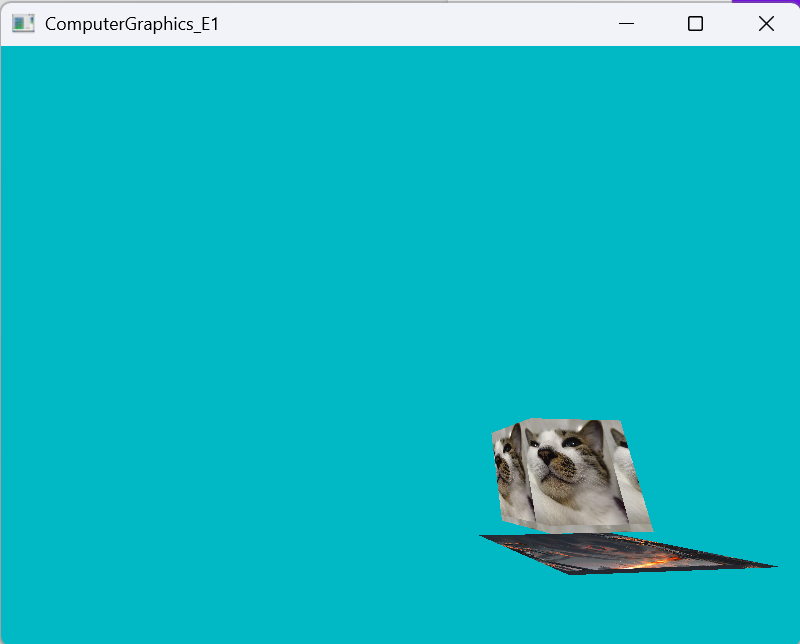
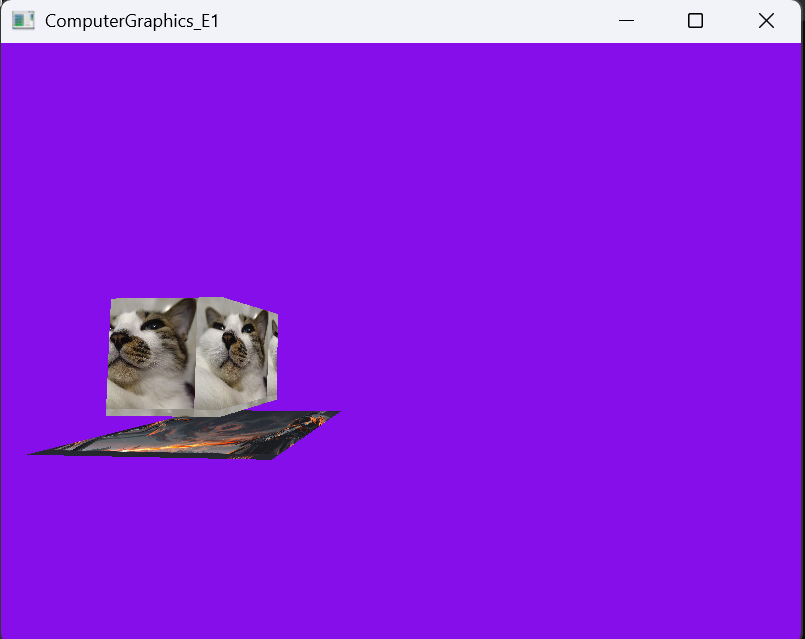


缩放变换

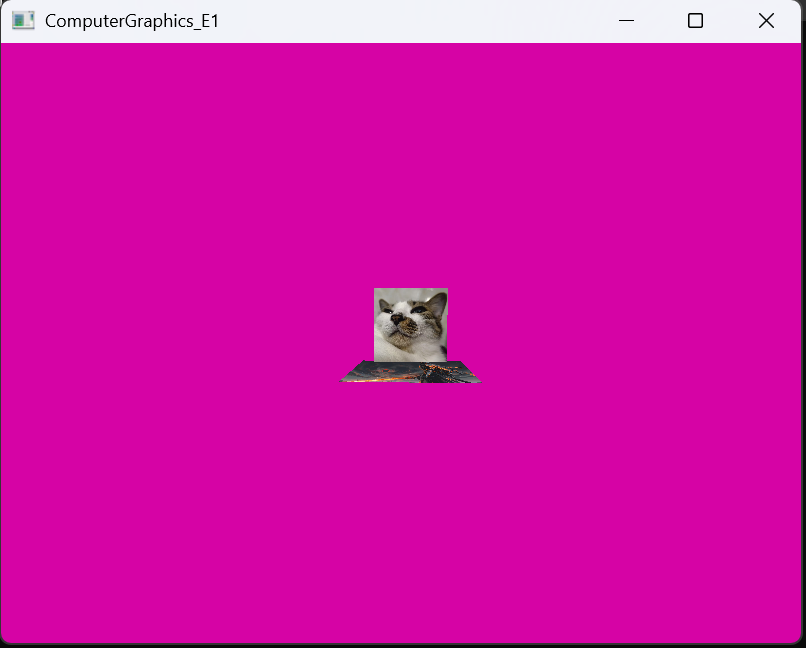
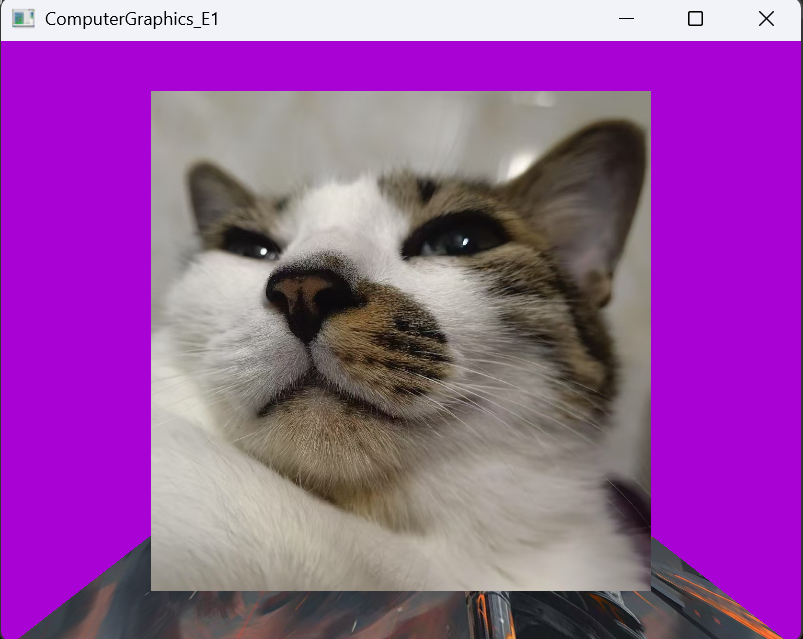


组合变换

### 5.2.2相机的移动

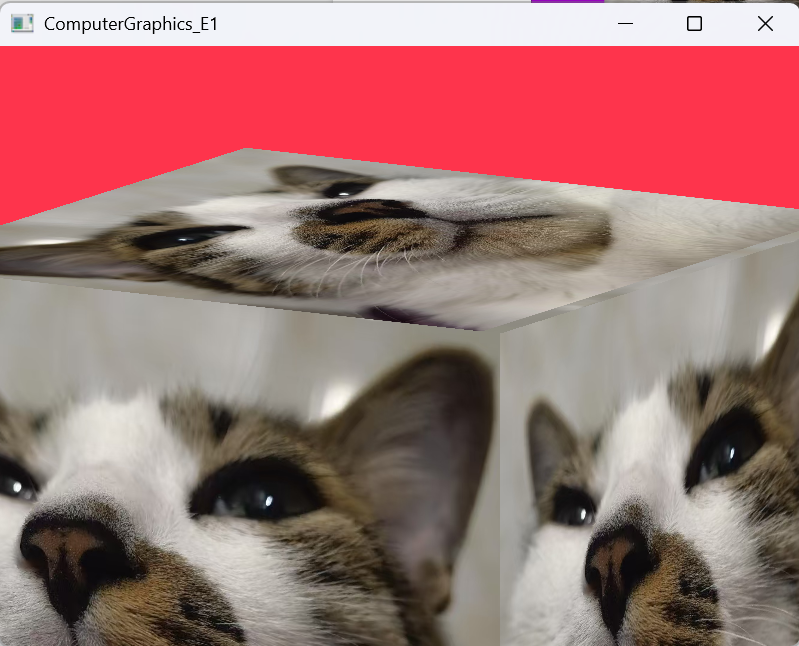


左右移动摄像机

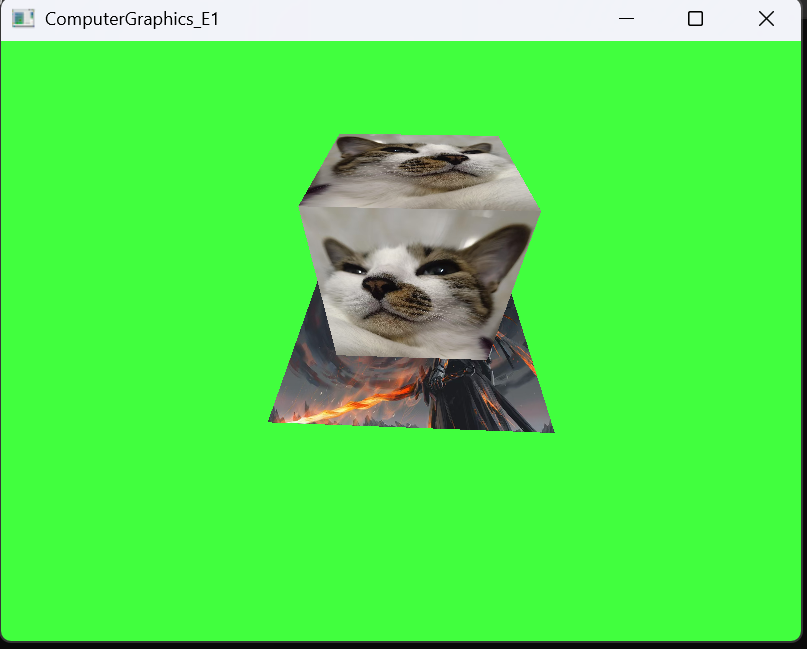


前后移动摄像机

### 5.2.3透视投影



正交投影



透视投影

# 6实验总结和感想

陈飞扬：这次实验是我第一次动手实践图形学，虽然对计算机和游戏接触的不少，但对于这一块还是很陌生，出现了不少问题，最主要的就是Cmake带来的一系列问题，从lib到解释器再到路径，配环境搞半天，队友配置好的依然无法编译通过，无奈只能转战python。在实操环节，通过写代码和报告，我将二维和三维的各种变换重新进行了推导，巩固了这部分知识，并且学习到了更多关于光照、材质等方面的知识

余旺：从未接触过图形学，说实话直接上手感觉难度太大了。既然这次主要负责二维图形的变换,那讲讲我的感受吧，最印象深刻的就是二维组合变换的探索。在变换的过程中没有按照预想的方式变换，我得出：1，glm变换函数的第一个参数(matrix)，是指:本次变换基于上次变换,2，glm旋转变换永远以本地坐标系的中心点为基,3，glm平移变换永远以缩放为基准，即缩放后的单位为缩放后为基准。还有个小问题是NDC坐标系下，一切按照NDC进行统一。在glBindVertexArray，glBindBuffer等操作时候，带给我的警示是有始就有终，绑定配套解绑，就像左括号和右括号的搭配，这里在写的时候存在漏掉怎么也找不到报错的问题。此外在尝试使用qt和vs后最终还是选择了VS+CMake,不是说更简便，或许是主流点，教程更多些，此次的学习中感觉整个搭建的过程是系统化工程化的学习下来了，感觉还不错，但是**真的真的好难**，包括后面三维和摄像机的书写等

常高林:在本次试验前，我便尝试过使用纯c++编写软光栅，很显然随着目标图像像素的提升带来了巨量的计算，在单线程的环境下渲染效率极其低下。而opengl的开发让我认识到了通过GPU加速运算带来的巨大算力，他使得实时的画面渲染成为可能。其中最让我感到惊叹的的是opengl作为久经考验的图形api的设计，他做到了具有很高的灵活性的同时，也极具易用性，我可以非常灵活的做到不同物体不同渲染器的切换。另一方面，opengl的状态机设计也暴露出一些不足，比如同一时刻只能绑定一种VBO对象以及同时只能对一种对象进行操作。显然这样的设计对多线程程序是不友好的，这也许就是Khronos group停止发展opengl转而开发vulkan的原因。当然对于初学者来说，opengl完善的文档及其易用性使其几乎是完美的入门级图形api。

张思宇:这次试验是我第一次尝试使用opengl制作二维/三维动画效果，与之前使用webgl&three.js实现类似效果对比，很明显感觉到更多底层的内容需要自己封装实现，加大了工作量但也强化了对底层知识的理解。同时强烈感觉到opengl对C++功底的要求还是有的，太久不写C++看组员写的C++代码有一点艰难，尤其是涉及到构造&左右值时。

同时又要吐槽一下C++老旧破败的包管理器，配置环境依旧需要手动下载对应包，解压缩，编译，手动加到Cmake中，尝试编译，发现和ＭＳＶＣ冲突，换版本，重新编译，感觉急需一个现代的包管理器拯救一下C++，隔壁Node包管理器都十几个了，谁来给C++写一个（。

农国凯：以前只有看过一点点图形学相关科普，第一次自己上手尝试才切实体验到了矩阵运算跟计算机图形显示相关关系，原来计算机当中向量的表示反应到矩阵运算上还是相对简单，不过实际的实操还是很有难度的，还有第一次切实体验到了uv的作用是什么，uv是怎么指导纹理的渲染的，平时只在各个地方听到，第一次切实见到。

# 7实验分工

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 余旺 | 软件2202 | 22301056 | 组长 | 23% |
| 陈飞扬 | 软件2202 | 22301060 | 组员 | 23% |
| 常高林 | 软件2202 | 22341129 | 组员 | 23% |
| 张思宇 | 软件2202 | 22301058 | 组员 | 13% |
| 农国凯 | 软件2204 | 22341063 | 组员 | 18% |

余旺 ：主要负责二维图形的变换，二维展示的键盘、鼠标控制，包括位移、旋转、缩放、反射、切变等。文档部分负责，剩下杂活都是了。

陈飞扬 :主要负责三维图形的绘制，旋转缩放平移变换等。

常高林：主要负责三维图形的键盘控制，鼠标控制，正交透视投影的切换，光照的等。

张思宇：负责文档校对排版，实验验证

农国凯 ：二维相关实验，二维图形绘制等，二维相关文档。