鲁班锁教学演示应用项目文档

三维计算机图形学及 3D 技术

作者：艾孜尔江·艾尔斯兰
时间：2019 年 12 月 12 日

1. 题目：

基于 OpenGL(使用 GLFW 和 GLAD 库)的鲁班锁教学演示应用

2. 项目梗概

根据计算机图形学及三维技术所学内容，利用标准库(GLFW、GLAD)和第三方库(Imgui、GLM 等)辅助，使用 OpenGL 编程语言完成从鲁班锁的模型渲染呈现到交互式计算机图形学相结合的鲁班锁教学演示程序。交互包含包含鼠标、键盘输入以及用户图形界面。

3. 项目意义

利用先进的三维计算机图形学技术展现中国传统文化元素，该项目作为一款基于 OpenGL 的图形学应用程序，除了用于探索和学习 OpenGL 相关知识及内容、拓展计算机图形学知识素养之外，还用于鲁班锁拆锁与装锁的教学演示，面向对象除了高校里学习计算机图形学的学生、各界研究和爱好计算机图形学的人士之外还面向普罗大众，进一步推广和普及中国传统文化精粹之一——鲁班锁，用科技感受古人之智！

4. 程序功能

Highlights:

1. 可以通过鼠标、键盘两种外设，对鲁班锁模型进行世界坐标系下的平移、旋转、缩放操作；
2. 可以对单个鲁班锁部件进行拾取操作，左键单击之后可以进入聚焦模式，独立对该部件进行交互操作，再次在该部件上进行单击操作可以退出聚焦模式回到之前的界面；
3. 为提示用户拾取结果，通过设置深度和模板测试表现了描边效果
4. 使用了较为真实的聚光灯效果，并可以随鼠标进行平移
5. 能够通过对用户友好的、易用的 UI 界面进行对于模型的交互，包含光照、贴图改变、动画(模型的拆分与组装效果)等设置；
6. 对于特殊材质，展示了半透明效果，方便用户看到模型部件的内部结构

5. 整体设计与详细流程

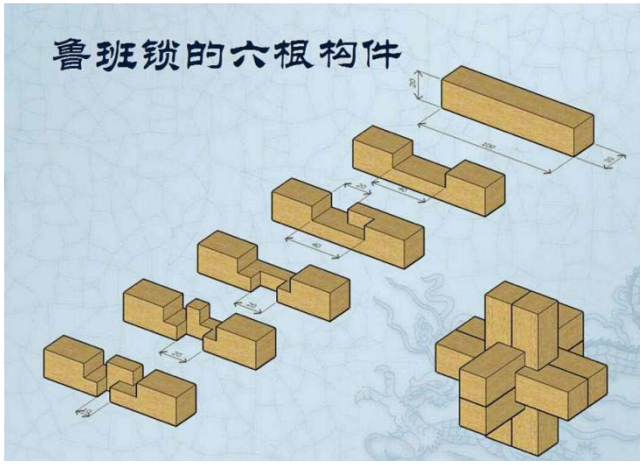
整体设计

从模型到贴图，从纹理到渲染，从零到一的建模呈现

本次项目是探究鲁班锁的结构，目的是弘扬中国传统文化。

由于鲁班锁模型的建模难度尚可，决定采取的非外部导入模型，而是在 OpenGL 里利用 VAO、VBO 与 EBO 进行绘制。

软件的具体使用如下：3ds Max（三维美术建模软件）、UV Layout（切 UV 的软件）、Photoshop（贴图绘制）、Substance Painter（三维纹理涂色）等。



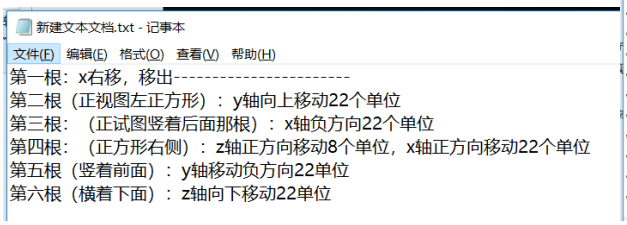
[图 1：鲁班锁的实际模型结构]

(附图：在三维涂色软件中生成的纹理贴图)

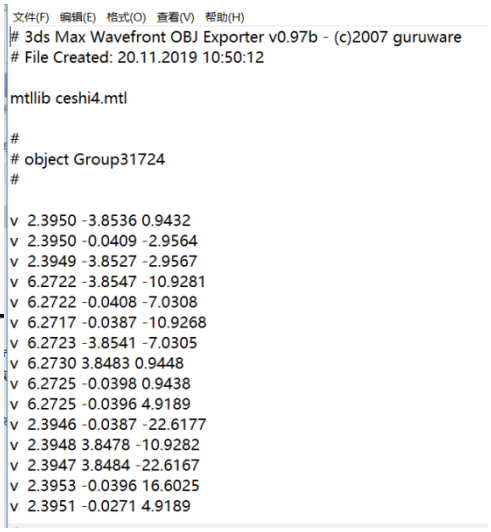
首先，使用 3ds Max 软件建立一个鲁班锁低模，之后手动进行拓扑，将模型面片绘制成全部为三角形。

之后切 uv，倒回到 3ds Max 里面贴上贴图，导出 obj 文件。

然后用 TXT 文本打开 obj 格式文件，手动读取 V---顶点坐标、Vt 纹理坐标、Vn----法线坐标、f-----面片索引，通过文件解析，将这些数据整理并导入到项目中。



（附图：动画数值设置）



（附图：三维建模软件导出的模型坐标）

交互设计、界面交互功能的实现（鼠标键盘的交互、ImGUI 窗体交互）、排错算法的实现

在实际的开发过程中，主要精力花在了交互功能开发与实现上，同时还实现了 ImGUI 界面的各类窗体及其具体功能，彻底舍弃原方案中的 Qt 框架，整个项目的窗体使用开源的 ImGUI 库来实现，有效地提升了工作效率。

另外，Shader 的基本撰写方式及其运用、纹理映射、投影变换等相关图形学知识也为衔接可交互界面与基本操作做了充分的铺垫。

在项目伊始，考虑到 OpenGL 自身在错误处理机制上的脆弱性，利用 C++语言中的 ASSERT 语句和 GLFW 库所提供的 glGetError()函数将排错功能封装起来，该算法有效解决了绘制图形的过程中容易因小错误引发的无法成功绘制的情况。

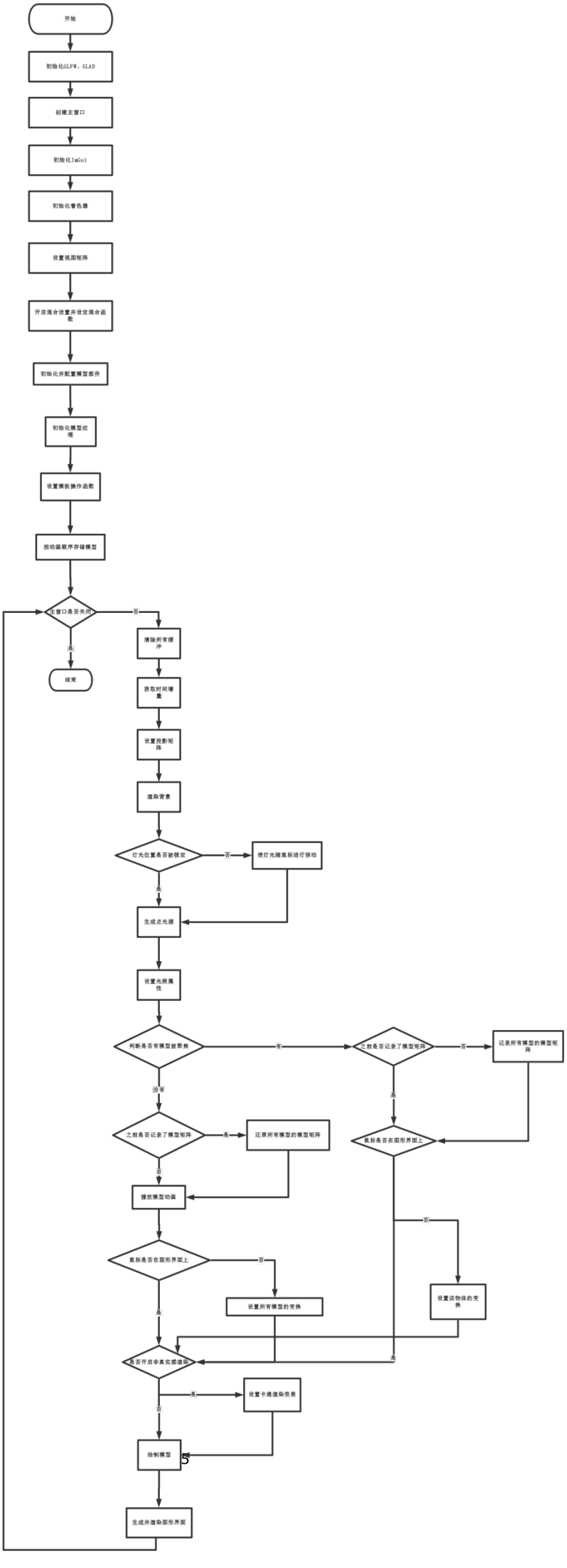
通过对于 glGetError()函数与其他函数的封装，就能避免 OpenGL 出错时的茫然，加上 ASSERT 语句对出错位置的断点，就能够在不设置任何断点的时候让程序停在出错的位置上。

同时，在项目的初始阶段，由于三维几何体（鲁班锁）的模型还没有被建出来，尝试着使用茶壶对交互功能进行大体的实现，重温大二上学期计算机图形学课程中所授的平移、旋转、缩放、光照等的交互部分，简单地实现了基础交互功能，同时还进一步实现了基于 GLUT 库的拾取问题。

但是毕竟 GLUT 库已经是上世纪的遗物，已经不在受到维护，在结构上与 GLFW 库有很多不同，因此最终决定利用层次包围盒来实现拾取，其基本思想是模型在屏幕上的坐标，同鼠标位置进行比较，通过点与直线相对位置。

在中期，如面对检测鼠标是否在窗口上这一问题进行了剖析并解决了通过观察者模式监听鼠标事件，并让窗口实现与 OpenGL 场景中物体的相分离，彻底避免了小窗体生成的程序对窗体内部物体的过度遮挡和阻碍。

总体流程图


开始
初始化GLFW、GLAD
创建主窗口
初始化ImGui
初始化着色器
设置视图矩阵
开启混合设置并设定混合函数
初始化并配置模型部件
初始化模型纹理
设置模板操作函数
按动画顺序存储模型
主窗口是否关闭
是
否
结束
清除所有缓冲
获取时间增量
设置投影矩阵
渲染背景
灯光位置是否被锁定
否
使灯光随鼠标进行移动
是
生成点光源
设置光照属性
判断是否有模型被聚焦
否
之前是否记录了模型矩阵
否
记录所有模型的模型矩阵
是
有
没有
之前是否记录了模型矩阵
是
还原所有模型的模型矩阵
否
鼠标是否在图形界面上
播放模型动画
否
鼠标是否在图形界面上
否
设置所有模型的变换
设置该物体的变换
是
是否开启非真实感渲染
是
设置卡通渲染效果
否
绘制模型
生成并渲染图形界面

详细设计

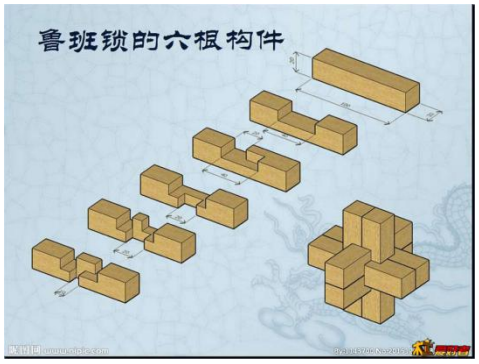
模型构建

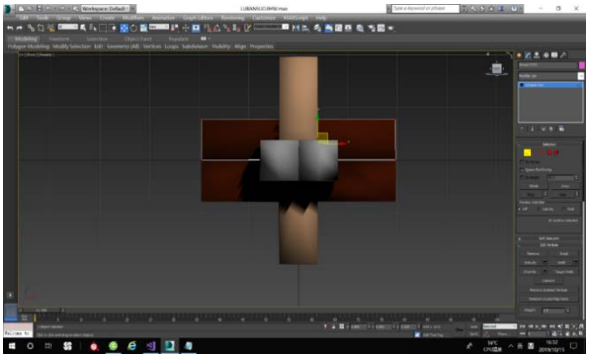
该项目是探究鲁班锁的内部结构，目的是弘扬中国传统文化。

鲁班锁，也叫八卦锁、孔明锁，相传是由战国时期的工匠大师鲁班制作而成，内部为榫卯结构。在古代，应用于房梁穿插的固定，也是广泛流传于中国民间的智力玩具。鲁班锁不用钉子和绳子，完全靠自身结构的连接支撑，就像一张纸对折一下就能够立得起来，展现了一种看似简单，却凝结着不平凡的智慧。

鲁班锁种类繁多，数不胜数，组合方式也各种各样，主要有 “别闷棍，”“六子联芳”，“莫奈何”，“难人木”等等。该项目根据最经典的“六子连芳”式鲁班锁，分析其内部结构，做出了这款鲁班锁展示程序，向优秀的中国传统文化致敬。

鲁班锁采取的并不是外部导入模型，而是在 opengl 里面建立模型。首先，用 3ds Max 软件建立一个鲁班锁低模，之后进行拓扑，通过 Editable Poly 组件转 Editable Mesh，将模型面片绘制成全部为三角形。





之后切 uv，倒回到 3D Max 里面贴上贴图，导出 obj 文件。然后用 txt 文本打开 obj 文件，读取 obj 文件数据。

导入数据的过程实际上是对 obj 文件数据进行解析的过程。Obj 文件中：v 代表顶点坐标、vt 代表纹理坐标、vn 代表法线坐标、f 代表面片的生成索引，部分数据如下图所示。

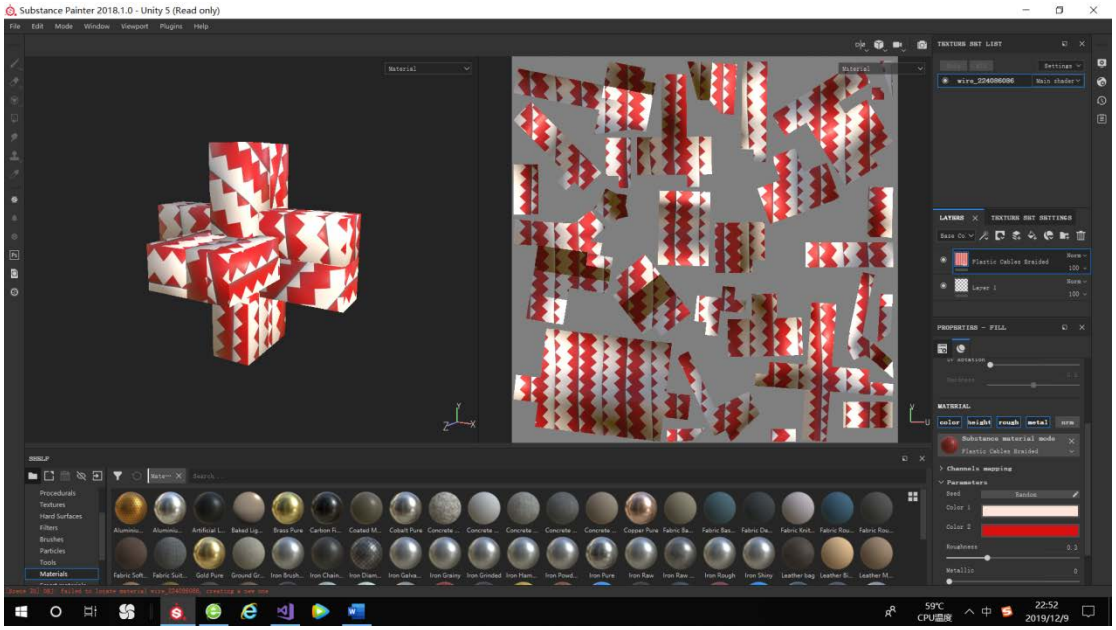
但是由于数据过于复杂，采取编写代码解析数据，通过 c#代码进行爬取数据，将数据整合。

代码核心思路如下：遍历 obj 文本，将 f（三角形面片）中所有数据由“/”隔开，每三个数据进行换行。将所有模型的 vn 数据所有装入一个数法线组，将所有模型的顶点数据装入另一个顶点数组。所有 vt 数据按序装入一个纹理数组。通过每一个 f 对应的数据，分别从顶点数组、法线数组、纹理数组中找到对应的值压入（v1，v2，v3，vt1，vt,，vn1，vn2，vn3）中。部分代码如下：

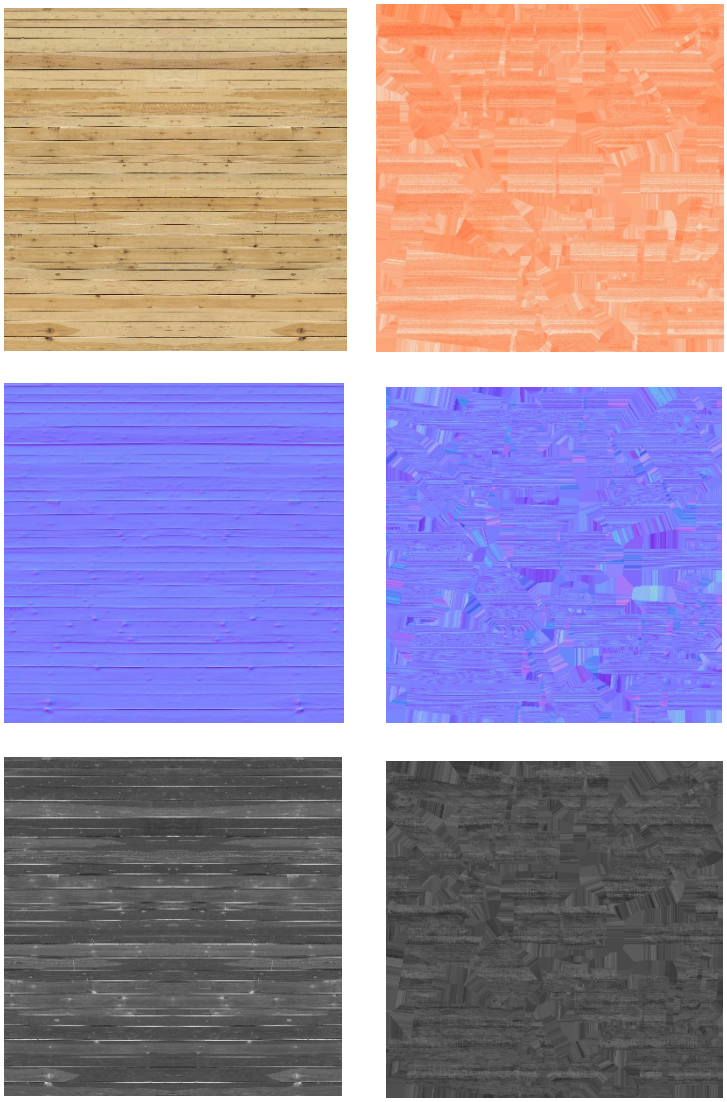
```
74
75
76          //用来存储新数据的数组
77          public static ArrayList vArray = new ArrayList();
78          public static ArrayList vtArray = new ArrayList();
79          public static ArrayList vnArray = new ArrayList();
80          public static ArrayList fArray = new ArrayList();
81          public static string finalString = "";
82
83          //将字符串转换成double的方法
84          static public double toDouble(string s)
85          {
86              return System.Convert.ToDouble(s);
87          }
88
89          static void Main(string[] args)
90          {
91              //读取所有的vt,并创建vt数组
92              StreamReader sr1 = null;
93              sr1 = File.OpenText(vtFilePath);
94              while (sr1.Peek() != -1)
95              {
96                  string dataText = sr1.ReadLine();
97                  string[] dataTextArray = System.Text.RegularExpressions.Regex.Split(dataText, @"\s{1,}");
98                  if (dataTextArray[0] == "vt")
99                  {
100                     VtPoint vt = new VtPoint(toDouble(dataTextArray[1]), toDouble(dataTextArray[2]));
101                     vtArray.Add(vt);
102                 }
103                 //创建v数组
104                 if (dataTextArray[0] == "v")
105                 {
106                     VPoint v = new VPoint(toDouble(dataTextArray[1]), toDouble(dataTextArray[2]), toDouble(dataTextArray[3]));
107                     vArray.Add(v);
108                 }
109                 //创建vn数组
110                 if (dataTextArray[0] == "vn")
111                 {
112                     VnPoint vn = new VnPoint(toDouble(dataTextArray[1]), toDouble(dataTextArray[2]), toDouble(dataTextArray[3]));
113                     vnArray.Add(vn);
114                 }
115             }
116
117             //单独读取f数组
118             StreamReader sr = null;
119             sr = File.OpenText(dataFilePath);
120             while (sr.Peek() != -1)
121             {
```

79 %

纹理贴图部分，首先在 Substance Painter 软件中将模型导入，在 Substance Painter 中绘制贴图，并进行烘焙，导出纹理贴图、光滑贴图、法线贴图。如下图所示：



部分纹理贴图如下：



构建模型：

鲁班锁由 6 个独立的榫卯组成。为了存储每一个榫卯的数据，构建了一个结构体 Model 来进行存储，存储顶点数据、数据索引，动画结点。

```
struct Model
{
public:
    mat4 model;//模型矩阵

    vector<vector<float>> verticesData;//顶点数据
    vector<vector<int>> dataIndices;//数据索引
    vector<vec3> animationPoint;//动画结点
```

其中，数据索引是存储数据信息，由 3,2,3 的数据结构组成，存进一维数组之中。第一个 3 是指每一个顶点的 x,y,z 坐标，2 是纹理的 x,y 坐标，最后一个 3 是每一个点对应的法线坐标。这样存储的 model 数组有 6 个。Model 结构体构造如下：

```
Model model1
    (
```

```
        {//顶点数据
-0.001,0.6544,0.1297,0.7179,0.8387,-0.0008,1,0.0006,
0.1278,0.6536,0.258,0.7077,0.7943,0.5789,0.5752,0.578,
0.1275,0.6543,0.1295,0.7349,0.8114,0.706,0.7082,0.0003,}
……
84, //数据索引
        {
            {//8个包围点
                vec3(-3.888 ,     -19.604, 7.742) ,
                vec3(3.835,     -19.6, 7.74) ,
                vec3(3.823,   -19.604, 0.026) ,
                vec3(-3.889,   -19.589, 0.027)
                      ……
            }
        },
        { vec3(0,-2.2f,0) }//动画结点
    );//正视图竖前面
```

之后数据索引也是用一维数组进行存储。在 model 结构体里面定义了一个 trianglesNumber 参量，用于存储三角形的面片数量。

例如第一个 Model，有 84 个面片，通过封装在 Model 结构体里面的初始化顶点数据函数进行压入数组。具体如下（注释）：

```
//初始化顶点数据
    Model(vector<float> vertices_data, int trianglesNumber, vector<vector<vec3>>
surroundPoints, vector<vec3>animationPoint)
//参数为：顶点数据，三角形面片数量，包围盒数据，动画结点
    {
        verticesData.push_back(vertices_data);//将顶点压入数组

        vector<int> indices_data;
        for (int i = 0; i < (trianglesNumber * 3); i++)
//将面片数量*3，作为索引压入数组，用一个for循环执行。因为每一个面片都是由三个点构成，所以第一个三角形面片的索引为0,1,2，之后的接在后面，总数为：trianglesNumber*3
        {
            indices_data.push_back(i);
        }
        dataIndices.push_back(indices_data);
        InitVBOandVAO();
        InitEBO();
```

同理，包围盒和动画数据也是如此压入数组。

通过重载函数 Model，将背景数据也存放至数组。函数本身基本无变化，更改了参量，具体如下：（因为背景本身只是一个面片。不涉及包围盒和动画）

```
    Model(vector<float> vertices_data, int trianglesNumber)//顶点数据，
    {
        verticesData.push_back(vertices_data);
```

```
        vector<int> indices_data;
        for (int i = 0; i < (trianglesNumber * 3); i++)
        {
            indices_data.push_back(i);
        }
        dataIndices.push_back(indices_data);
        InitVBOandVAO();
        InitEBO();
    }
```

之后根据结构体中的数据信息，生成 VAO,VBO。生成 VAO,VBO 则被封装到自定义的 InitVAOandVBO()函数中。VAO 是指顶点数组对象，VBO 是顶点缓冲对象。通过 glGenBuffers()生成 VBO 和 glGenVertexArrays()生成 VAO。

```
void InitVBOandVAO()
    {
        /*将接收的顶点数据存储下来*/
        vertices_data = new float[verticesData[0].size()];
        for (int i = 0; i < verticesData[0].size(); i++)
        {
            vertices_data[i] = verticesData[0][i];
        }
        glGenBuffers(1, &VBO);//生成VBO
        BindObject(ObjectType::VBO);//绑定VBO
        glBufferData(GL_ARRAY_BUFFER, verticesData[0].size() * sizeof(float),
vertices_data, GL_STATIC_DRAW);//配置顶点数据
        glGenVertexArrays(1, &VAO);//生成VAO
        BindObject(ObjectType::VAO);//绑定 VAO
```

之后是索引缓冲对象 EBO 的处理。索引缓冲即对面片的构建，通过索引找顶点绘制。也是通过 glGenBuffers()函数进行生成，glBufferData()函数配置索引数据。

```
void InitEBO()
    {
        /*将接收的索引数据存储下来*/
        indices_data = new int[dataIndices[0].size()];
        for (int i = 0; i < dataIndices[0].size(); i++)
        {
            indices_data[i] = dataIndices[0][i];
        }
        glGenBuffers(1, &EBO);//生成EBO
        BindObject(ObjectType::EBO);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, dataIndices[0].size() * sizeof(int),
indices_data, GL_STATIC_DRAW);//配置索引数据
    }
```

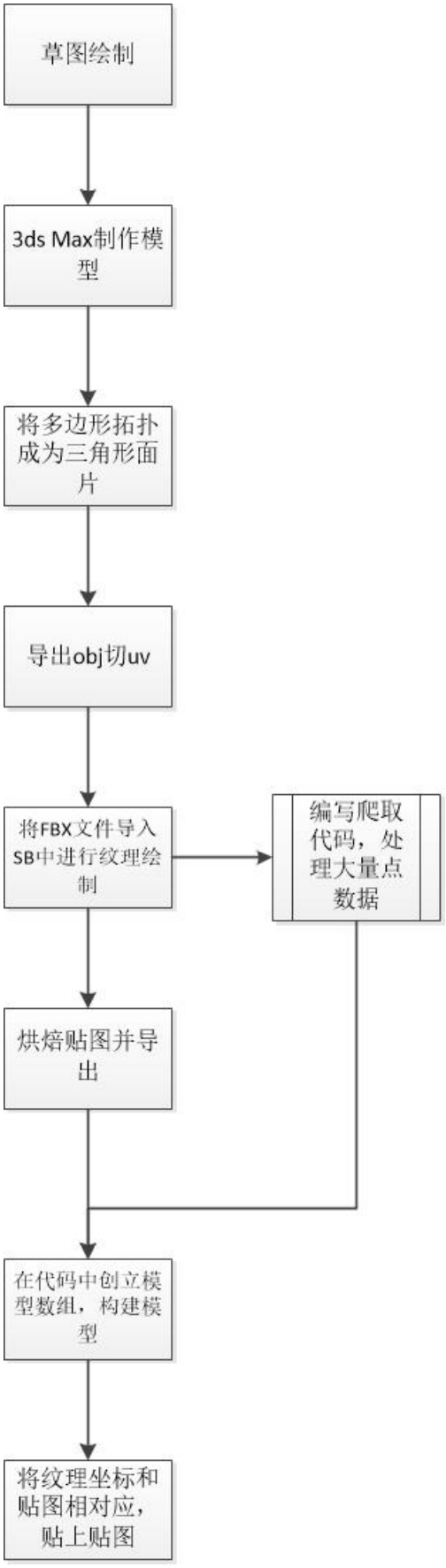
之后进行绑定 VAO、VBO，EBO，将 glBindBuffer()函数封装进自定的 BindObject()之中。

配置模型：

模型配置定义了一个 ConfigModel 函数，也封装在了 struct model 结构体之中。函数用于配置模型、其中第一个参量代表的是 shader 的序号，第二个参量是读取的数据个数，第三个参量是顶点数据类型，第四个指的是数据是否被标准化，第五个指的是步长，即每隔 8 个读取数据，sizeof 指的是是初始数据的起始位置。如下：

```
void ConfigModel(Model model)//配置模型函数
{
    model.ConfigAttribute(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)0);
    model.ConfigAttribute(1, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(3 * sizeof(float)));
    model.ConfigAttribute(2, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(5 * sizeof(float)));
    model.UnBindObject();
}
```

动画移动方向和数据通过在 3ds Max 中观察动画记录在文本中，具体如下：

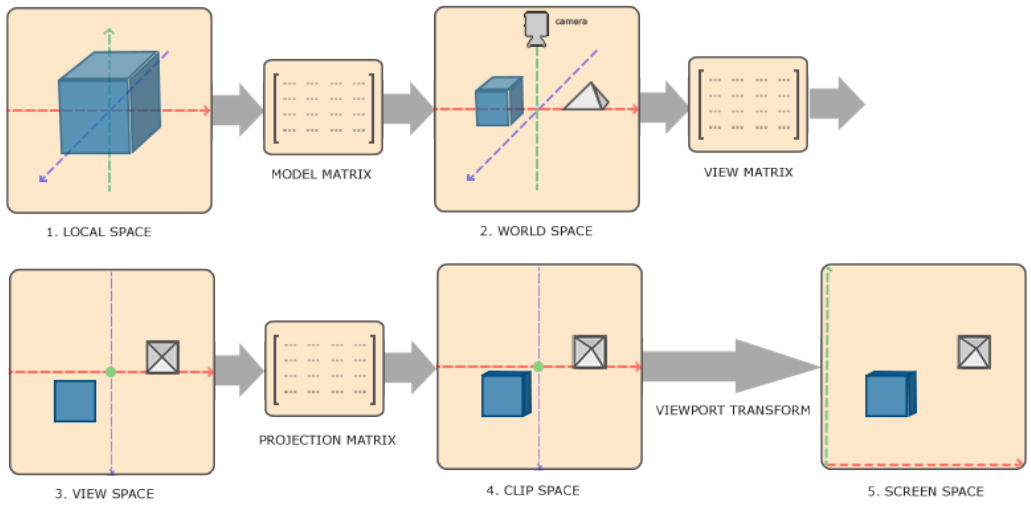

草图绘制
3ds Max制作模型
将多边形拓扑成为三角形面片
导出obj切uv
将FBX文件导入SB中进行纹理绘制
编写爬取代码，处理大量点数据
烘焙贴图并导出
在代码中创立模型数组，构建模型
将纹理坐标和贴图相对应，贴上贴图

模型交互

首先，考虑到用户最经常做的动作——改变窗口大小，通过 glViewport(0, 0, width, height)函数就完成了窗口中的物体根据用户对窗口大小的改变而同步发生相应的缩放这一功能，为方便最后的代码的融合，将其嵌套在一个函数中，这样一来想要实现这一功能的时候就可以直接通过调用所撰写的这个函数来解决视口变换与物体缩放变换绒布的问题，同时也达到了用户对视口地变换和视口内的物体变换相统一地目的。

对于鼠标的信息，分别设置了 lastmousePosition（定义上一个鼠标的位置）、mousePositionDelta（定义鼠标位置的增量）、isLeftMouseButtonRepeat（定义鼠标左键长按与点击事件——鼠标右键也采取同样的方式）、isLeftMouseButtonRelease（定义鼠标左键是否抬起）这几个全局变量，从框架结构上将鼠标交互画了一个外轮廓，之后再具体地撰写函数，对于光标位置的回调函数，通过撰写 CursorPosCallback()函数，并在这个函数里面撰写三个变量以接收具体的窗口指针变量、光标在窗口中的 X 坐标和 Y 坐标的坐标，根据先前定义的鼠标信息全局变量运用 OpenGL 官方的二维函数将 vec2()将光标的位置记录下来实时返还给调用方。

在这些基本的鼠标交互框架之后，根据具体的实例进行了物体的平移、旋转、缩放等变换。在通过交互来实现物体的平移旋转和缩放这一功能的时候，借助课程中所学的内容，通过物体与世界二者之间的坐标变换来具体实现这一层逻辑。为了将坐标从一个坐标系变换到另一个坐标系，用到了几个变换矩阵，最重要的几个分别是模型(Model)、观察(View)、投影(Projection)三个矩阵。的顶点坐标起始于局部空间(Local Space)，在这里它称为局部坐标(Local Coordinate)，它在之后会变为世界坐标(World Coordinate)，观察坐标(View Coordinate)，裁剪坐标(Clip Coordinate)，并最后以屏幕坐标(Screen Coordinate)的形式结束。下面的这张图展示了整个流程以及各个变换过程中本质上做了哪些工作：



在这里需要提一下摄像机移动的相关原理，在实现摄像机移动的过程中的移动速度直接利用了一个常量。理论上这样做并没什么打问题，但是实际情况下根据处理器的能力不同，有些电脑可能会比其他电脑每秒绘制更多帧，也就是以更高的频率调用 processInput 函数。结果就是，根据计算机的配置的不同，组内有些人可能移动很快，而有些人会移动很慢。这是在测试过程中所发现的一个设备方面的痛点和难点。当最终测试的时候，必须要确保它在所有硬件上移动速度都一样。因此最终还是选择直接使用一个常量来移动相机。图形程序和游戏通常会跟踪一个时间差(Deltatime)变量，它储存了渲染上一帧所用的时间。把所有速度都去乘以 deltaTime 值。结果就是，如果的 deltaTime 很大，就意味着上一帧的渲染花费了更多时间，所以这一帧的速度需要变得更高来平衡渲染所花去的时间。使用这种方法时，无论电脑快还是慢，摄像机的速度都会相应平衡，这样组内每个人的体验就都一样了，圆满解决了出现的上述问题。

对于鼠标输入模块，在移动物体的时候所选取的方式是利用偏航角和俯仰角是通过鼠标（或手

柄）移动获得的，水平的移动影响偏航角，竖直的移动影响俯仰角。它的原理就是，储存上一帧鼠标的位置，在当前帧中当前计算鼠标位置与上一帧的位置相差多少。如果水平/竖直差别越大那么俯仰角或偏航角就改变越大，也就是摄像机需要移动更多的距离。

首先要告诉 GLFW，它应该隐藏光标，并捕捉它。捕捉光标表示的是，如果焦点在程序上，也就是说，如果正在操作这个程序，Windows 中拥有焦点的程序标题栏通常是有颜色的那个，而失去焦点的程序标题栏则是灰色的，这时候，光标应该停留在窗口中，除非程序失去焦点或者退出。用了一个简单地配置调用来完成——glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);在调用这个函数之后，无论怎么去移动鼠标，光标都不会显示了，它也不会离开窗口。对于 FPS 摄像机系统来说非常完美。

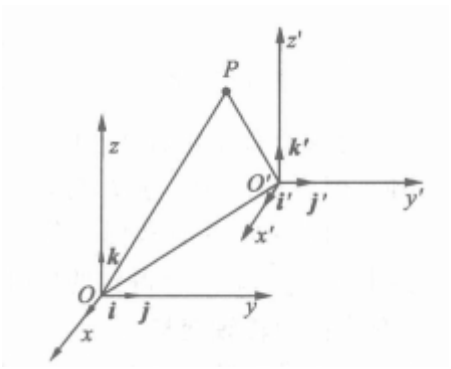
为了计算俯仰角和偏航角，还让 GLFW 监听鼠标移动事件。和键盘输入相似的是，用了回调函数来完成。

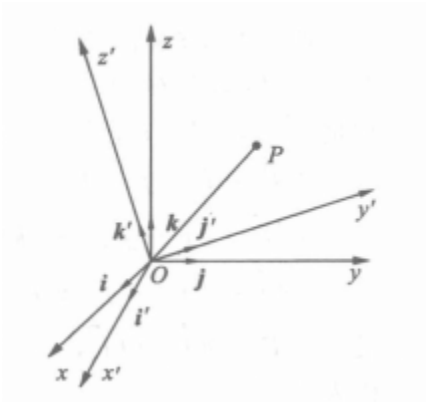
对于键盘交互，根据 GLFW 官方提供的函数举例，实现了自主撰写键盘交互函数的过程。在主函数中利用各类回调函数就能够完成交互部分的最佳体验，这种体验无论是对于用户来说抑或是对于真正了解交互设计本质的程序员们来说都是一个友好的互动方式。

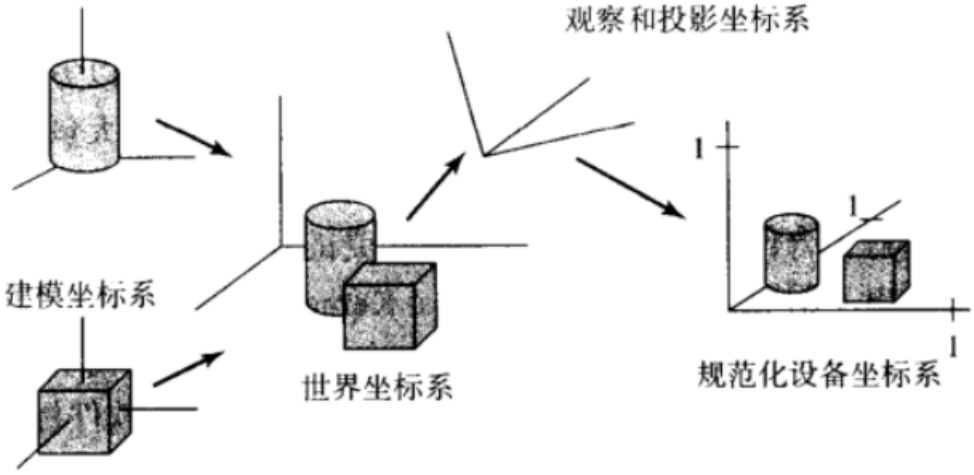
补充：

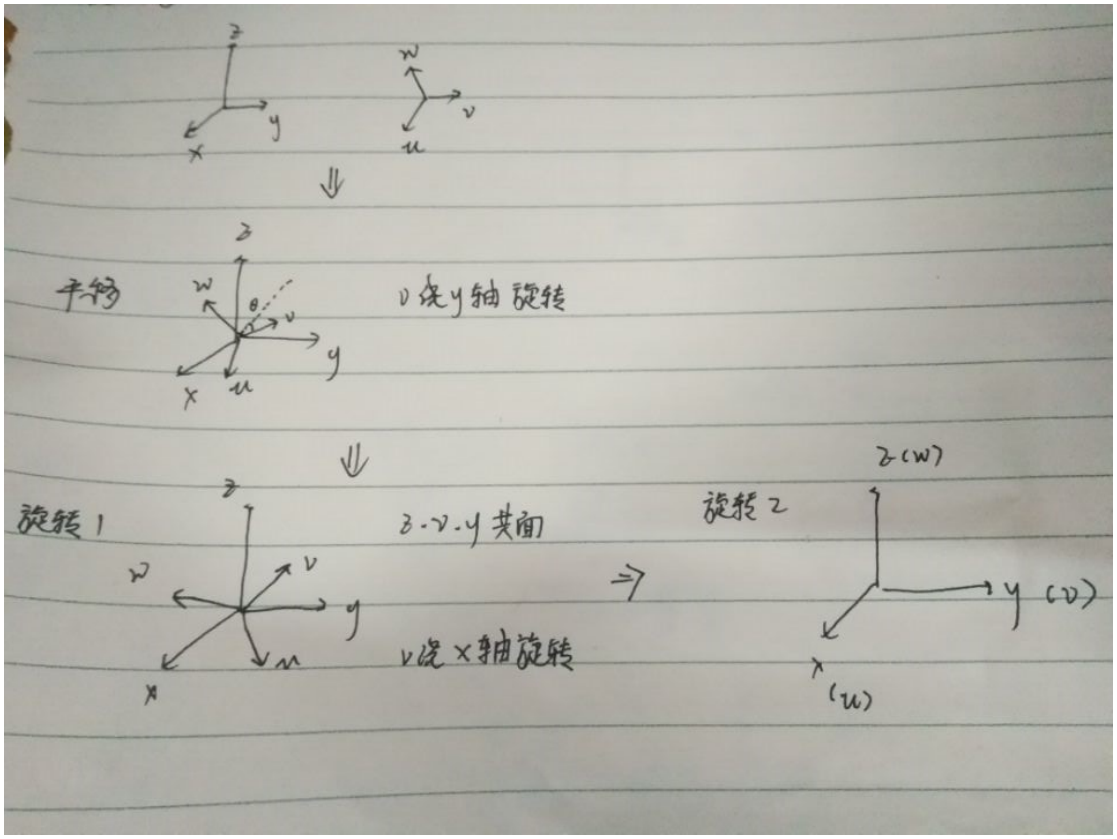
```
/*设置变换属性*/
        void SetTransform(vec3 translateDelta, vec2 RotateAngle, vec3 scaleDelta)
        {
            //进行世界坐标系下缩放
            model = translate(mat4(1), -translateSum) * model;//平移变换
            model = scale(mat4(1), scaleDelta) * model;//缩放变换
            model = translate(mat4(1), translateSum) * model;//逆平移变换

            //进行世界坐标系下旋转
            model = translate(mat4(1), -translateSum) * model;//平移变换
            model = rotate(mat4(1), radians(RotateAngle.x), vec3(0, 1, 0)) *
rotate(mat4(1), radians(RotateAngle.y), vec3(-1, 0, 0)) * model;//将模型矩阵平移到世界坐标原点在旋转
            model = translate(mat4(1), translateSum) * model;//逆平移变换

            //平移
            model = translate(mat4(1), translateDelta) * model;//纯粹的平移变换
        }
```

使用矩阵左乘的形式对物体进行平移，否则当物体旋转完之后就会出现世界坐标和物体自身坐标不一致的情况，translate(mat4(1),-translateSum)＊model 中的 mat4(1)实际上是指单位矩阵，尔后的 translateSum 是它的平移变量，这个平移变量可以根据用户移动的状况来定，下图呈现了本项目的原理：


z'
P
z
k'
O'
i' j'
y'
x'
k
O
i j
y
x


z'
z
P
y'
k'
k
j'
i
O
j
y
i'
x
x'


观察和投影坐标系
1
1
1
建模坐标系
世界坐标系
规范化设备坐标系



[的前期思考]

translateKeyDelta *= 10;　　rotateKeyDelta *= 10;　　scaleKeyDelta *= 10;这几个为鼠标交互事件中的系数值，奖这个系数存入矩阵中并将其传给平移变换，就能够相对快速的对物体进行平移。

鼠标拾取算法的实现：

0. 每个鲁班锁部件包含一个三维数组，作为其包围点，共 8 个

这 8 个包围点即作为包围物体的包围盒的八个顶点

1. 首先在创建鲁班锁时，通过构造函数传入初始的包围点(实例化包围盒对象)

```
//初始化包围点
SurroundBox(vector<vec3> points)
{
    for (int i = 0; i < 8; i++)
    {
        初始包围点[i] = points[i];
    }
}
```

2. 在每次循环时，更新包围点的位置并得到在屏幕上的坐标

```
//更新包围点
void UpdateSurroundPoint(mat4 模型矩阵, mat4 视图矩阵, mat4 透视矩阵, Window 窗口)
{
    for (int i = 0; i < 8; i++)
    {
```

```
                vec4 临时点 = (透视矩阵* 视图矩阵 * 模型矩阵) * vec4(初始化的包围点[i], 1);//从世界坐标到裁剪坐标
                if (临时点.w != 0)
                {
                    临时点 /= 临时点.w;
                }//进行标准齐次除法得到 NDC
                包围点[i] = vec2(临时点.x * (窗口.窗口宽度/ 2) + (窗口.windowWidth / 2), tempPoint.y * -(窗口.windowHeight / 2) + (窗口.windowHeight / 2));//从视口坐标到屏幕坐标
            }
        }
```

3. 得到屏幕上的八个点之后，首先获取其中的四个极端点(横向与纵向最值的四个点)

```
    //获取极端点
    void GetExtramePoint()
    {
        //初始化极端点
        for (int i = 0; i < 4; i++)
        {
            极端点[i] = 包围点[0];
        }

        //通过比较计算极端点
        for (int i = 1; i < 8; i++)
        {
            if (极端点[x 最小].x > 包围点[i].x)
            {
                极端点[x 最小] = 包围点[i];
            }
            if (极端点[x 最大].x < 包围点[i].x)
            {
                极端点[x 最大] = 包围点[i];
            }
            if (极端点[y 最小].y > 包围点[i].y)
            {
                极端点[y 最小] = 包围点[i];
            }
            if (极端点[y 最大].y < 包围点[i].y)
            {
                极端点[y 最大] = 包围点[i];
            }
        }
    }
```

4. 在得到上述四个点的基础之上，从包围点中得到剩余的四个点

```
    //获取剩余点
    void GetLeftPoint()
```

```
{
    //记录包围点中除去极端点的点
    for (size_t i = 0; i < 8; i++)//遍历包围点
    {
        包围点和极端点是否一致 = false;//假设是新的包围点
        for (int j = 0; j < 4; j++)//遍历极端点
        {
            if (包围点[i] == 极端点[j])//如果某个包围点和某个极端点一致
            {
                包围点和极端点是否一致 = true;//设定包围点与极端点一致
                break;
            }
        }
        if (包围点和极端点不一致)//如果与四个极端点都不一样
        {
            临时点.push_back(包围点[i]);//记录此点
        }
    }

    //初始化剩余点
    for (size_t i = 0; i < 4; i++)
    {
        剩余点[i] = 临时点[0];
    }

    //通过比较计算剩余点
    for (int i = 1; i < 4; i++)
    {
        if (剩余点[x最小].x > 临时点[i].x)
        {
            剩余点[x最小] = 临时点[i];
        }
        if (剩余点[x最大].x < 临时点[i].x)
        {
            剩余点[x最大] = 临时点[i];
        }
        if (剩余点[y最小].y > 临时点[i].y)
        {
            剩余点[y最小] = 临时点[i];
        }
        if (剩余点[y最大].y < 临时点[i].y)
        {
            剩余点[y最大] = 临时点[i];
        }
    }
```

```
}
```

5.最后判断鼠标是否在物体上，即是否在由极端点或者剩余点构成的平面上

```
//鼠标是否在包围盒上
bool IsMouseSelect()
{
    bool 是否在极端点区域内 =
        (鼠标位置是否在 x、y 最小值极端点所构成直线的上方&&
        鼠标位置是否在 x、y 最大值极端点所构成直线的下方&&
        鼠标位置是否在 x 最小值,y 最大值极端点所构成直线的下方&&
        鼠标位置是否在 y 最小值,x 最大值极端点所构成直线的上方);//是否在极端点构成的区域内

    bool 是否在剩余点区域内 =
        (鼠标位置是否在 x、y 最小值剩余点所构成直线的上方&&
        鼠标位置是否在 x、y 最大值剩余点所构成直线的下方&&
        鼠标位置是否在 x 最小值,y 最大值剩余点所构成直线的下方&&
        鼠标位置是否在 y 最小值,x 最大值剩余点所构成直线的上方);//是否在剩余点构成的区域内
    return (是否在极端点区域内|| 是否在剩余点区域内);//取区域的并集
}
```

补充：点与直线关系的判定算法：

基本原理即根据点和直线的关系判别式进行判断：

对于直线 Ax+By+C=0，将点坐标带入 Ax+By+C，若结果大于 0 说明点在直线下方，否则为上方或线上

又因为 y=kx+b 中，k=$\frac{y2-y1}{x2-x1}$，b=$\frac{y1x2-y2x1}{x2-x1}$，可得(y2-y1)x+(x1-x2)y+x2y1-x1y2=0

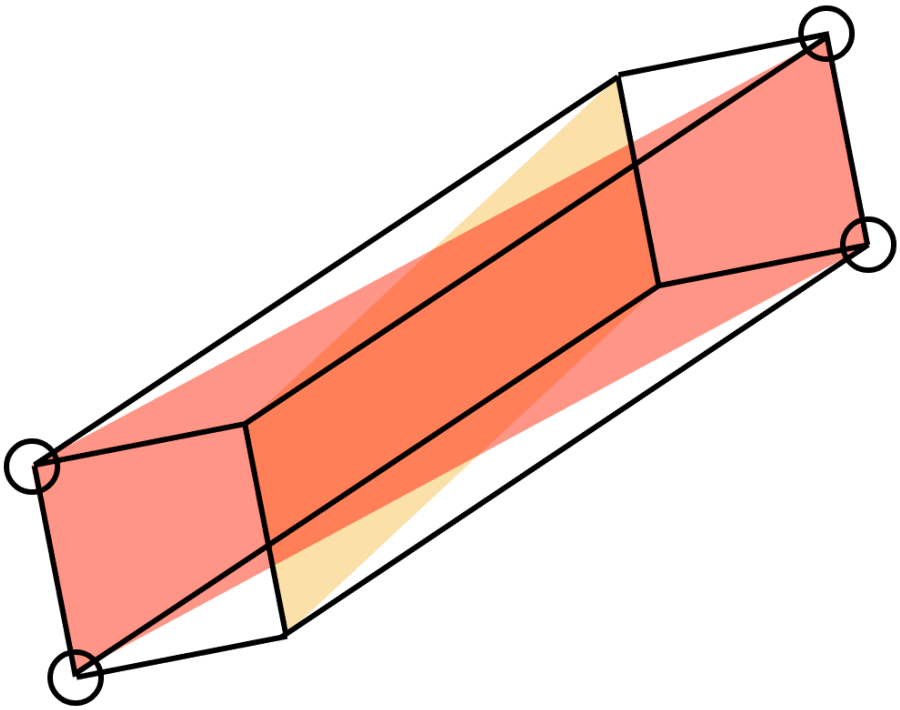
即 A=(y2-y1)；B=(x1-x2)；C=x2y1-x1y2

参考：https://blog.csdn.net/madbunny/article/details/43955883

实现伪代码：

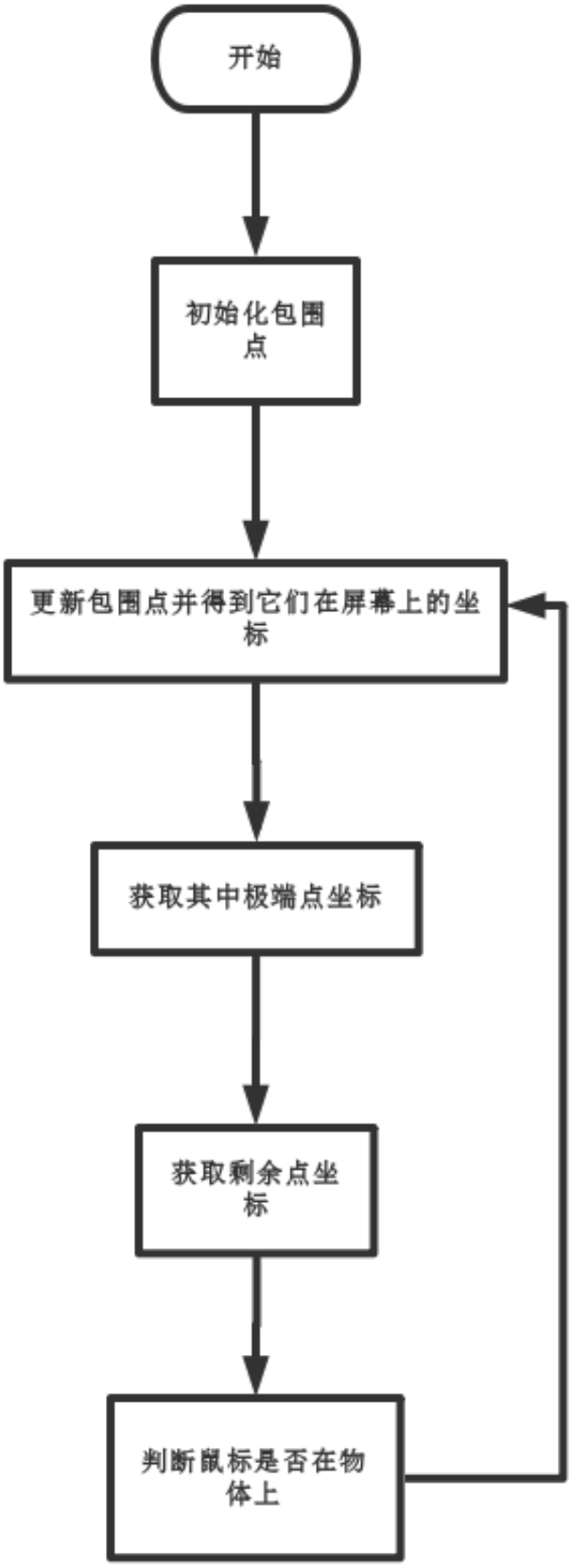
```
bool 点是否在直线下方(vec2 需要判断的点, vec2 线段左端点, vec2 线段右端点)
{
    float A = (线段右端点.y) - (线段左端点.y);
    float B = (线段左端点.x) - (线段右端点.x);
    float C = (线段右端点.x * 线段左端点.y) - (线段左端点.x * 线段右端点.y);
    return ((A * 需要判断的点.x + B * 需要判断的点.y + C) > 0);
}
```

补充：



(附图：红色加黄色区域为鼠标的可响应区域，当鼠标位于这些区域时认为物体被选中；圆圈标注的为极端点)

流程图：


开始
初始化包围点
更新包围点并得到它们在屏幕上的坐标
获取其中极端点坐标
获取剩余点坐标
判断鼠标是否在物体上

动画算法实现：

0. 动画的本质是物体的位移，难点在于如何根据给定的时间结点确定物体的位置，首先再构建模型时便记录每个物体的动画关键结点，中间进行补帧操作；应注意每个部件的第一个动画点是手动设定的 vec3(0,0,0)

1. 首先初始化部件的移动顺序，即根据给定的顺序记录模型

```
//初始化动画顺序
void InitAnimationSequence(vector<Model*> 模型, vector<int> 动画顺序)
{
    for (int i = 0; i < 模型数组包含数量; i++)
    {
        按照动画顺序存储的模型数组.push_back(模型[动画顺序[i]]);//根据顺序记录所有模型
    }
}
```

2. 在渲染循环中，首先要进行的操作是根据给出的时间结点，取得当前应当移动的物体，并确保之前和之后的部件处于正确位置

基本思路为在确保时间结点不超过 1 的前提下，通过循环得到当前应当移动的部件，即设定将动画时长限定在 0-1 的前提下，按部件输入等分时间段，再判定给定的时间点处于哪个时间段内

得到该索引后，在将该索引之前的部件设置为达到其内部的动画时间的末尾状态；索引之前的部件设置为达到其内部的动画时间的初始状态，以保证非连续播放动画时物体位置的正确性

根据得到的索引，计算对于单个物体来说的动画结点，即通过当前的动画结点减去索引值乘上 1/部件总量，再乘上 1/部件总量得到

```
//根据给出的动画时间点播放动画
void PlayAnimation(float 动画时间结点, Shader 模型着色器)
{
    if (动画时间结点< 1)//确保时间不会超过 1
    {
        int 动画顺序索引= 0;
        for (int i = 0; i < 模型动画序列.size(); i++)
        {
            //获取动画顺序索引
            if (动画时间结点 >= (动画顺序索引 + 1) * (1.0f / 模型动画序列.size()))
            {
                ++动画顺序索引;
            }
        }
        for (int i = 0; i < 模型动画序列.size(); i++)
        {
            if (i < 动画顺序索引)
            {
                模型动画序列[i]->设置动画(0.9999f);
            }
```

```
                else if (i > 动画顺序索引)
                {
                    模型动画序列[i]->设置动画(0);
                }
            }
            动画时间结点 -= (动画顺序索引) * (1.0f / 模型动画序列.size());//将时间确定到单个物体对应的动画时间
            模型动画序列[动画顺序索引]->设置动画(动画时间结点/ (1.0f / 模型动画序列.size()));//播放对应物体的动画
        }
    }
```

3. **根据计算后的动画时间结点，确定部件的位置**

用第 2 步骤类似的循环增加返回，获得给定的时间结点对应的动画索引(应当播放哪段动画)，之后获得目标位置为

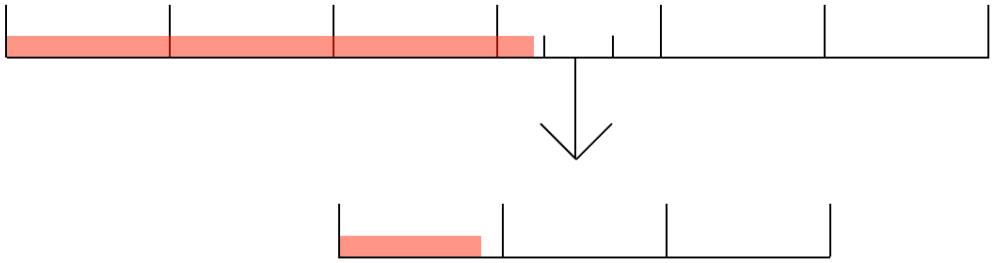
$$\square\square\square\text{动画}\square\square\square\square\square\square\square\square\text{画结点}+\frac{\text{动画时间结点}-(\frac{\text{上一个动画索引}}{\text{动画点数量}-1})}{\frac{1}{\text{动画点数量}-1}}\text{之后}$$

使物体按照自身坐标系进行平移，平移向量为目标位置-物体当前位置

最后记录物体的位置方便下次循环使用

```
//设置动画
void SetAnimation(float 动画时间结点)
{
    int 动画索引 = 0;
    //确定动画索引
    while (动画时间结点 >= ((动画索引) * (1.0f / (动画关键点.size() - 1))))//如果动画时间大于某个动画结点对应的时间
    {
        ++动画索引;//动画索引自增
    }
    vec3 目标位置 =
        (
        (
            (动画时间结点 - ((动画索引 - 1) * (1.0f / (动画关键点.size() - 1)))) / (1.0f / (动画关键点.size() - 1))
            )
            * (动画关键点[动画索引] - 动画关键点[动画索引- 1])
            )
        + 动画关键点[动画索引- 1];//获取移动插值
    模型矩阵 = 平移(模型矩阵, 目标位置 - 物体位置);//获取物体目标位置
    物体位置 = 目标位置;//记录目标位置
}
```

补充：



（附图：红色为动画进度，上方为所有模型的整体动画进度，下方为单个模型的动画进度，首先确定为第几个模型的动画，在确定该模型的第几段动画）

图形绘制以及描边效果实现：

0. 初始化纹理

首先使用 stb_image.h 文件中的 stbi_set_flip_vertically_on_load(true);函数，设定加载图片时，反转 y 轴以正确显示

之后对纹理进行生成，利用 glGenTextures(1, &textureID)生成纹理，再利用 stbi_load(path, &width, &height, &nrComponents, 0) 获取到图片的通道信息；之后利用 glBindTexture(GL_TEXTURE_2D, textureID);绑定图片到纹理;再使用 glTexImage2D 函数生成纹理；最后使用 glGenerateMipmap 生成多级渐远纹理

注意需要设置 Mipmap 的环绕与过滤模式，本项目中使用 glTexParameteri 函数进行设置；最终得到生成漫反射和镜面光纹理并得到其 ID

1. 在渲染循环中，每次利用 glActiveTexture 和 glBindTexture 函数激活并绑定纹理

2. 设置对于模板值的操作

利用 glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);进行设定，即当深度测试不通过或模板与深度测试都通过时，将对应的模板值设定为参考值(本项目设定为 1)

3. 进行模板设置

首先判断是否有进入了聚焦模式，没有则对每个模型进行 MVP 矩阵的变换；再判断鼠标是否在模型上，即使用上文中提及的鼠标拾取算法

如果鼠标不在部件上，则关闭模板写入并关闭模板测试，再进行模型的绘制；如果鼠标在某个部件上，则开启模板写入与模板测试，并设置总是渲染模型，渲染后设置模板值为 1；最后记录该物体的索引值以确认其需要被描边

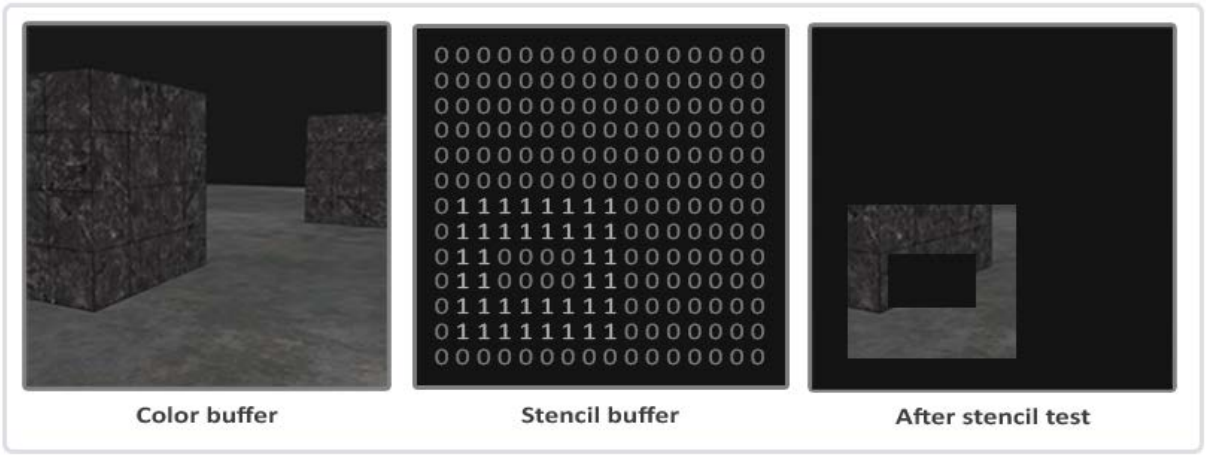
另外，项目中设定了当鼠标选中了物体并点击鼠标左键后，会记录该物体的索引为聚焦模式使用

4. 渲染描边

经过第 3 步之后，如果有物体需要被描边，则开启模板测试、关闭深度测试，并设定模板值不为 1 的是否才渲染物体，再利用描边着色器进行该物体的绘制，并将该物体进行一定程度的放大，以作为原来模型的边缘存在

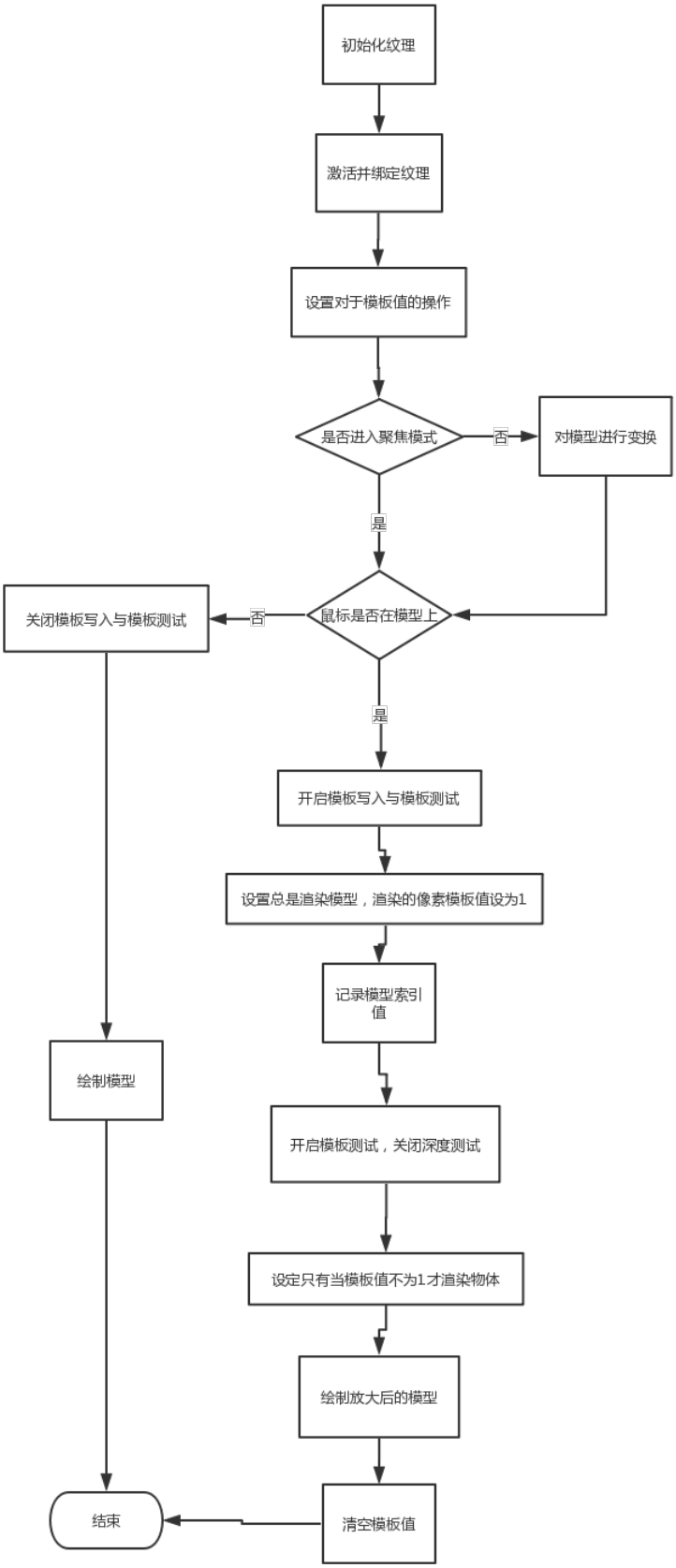
绘制描边后，需要再次清空模板值

补充：



（附图：在本项目中，描边效果即为设定了当模板值为 0 的区域才可以被渲染）

流程图：


初始化纹理
激活并绑定纹理
设置对于模板值的操作
是否进入聚焦模式
否
对模型进行变换
是
关闭模板写入与模板测试
否
鼠标是否在模型上
是
开启模板写入与模板测试
设置总是渲染模型，渲染的像素模板值设为1
记录模型索引值
绘制模型
开启模板测试，关闭深度测试
设定只有当模板值不为1才渲染物体
绘制放大后的模型
结束
清空模板值

着色器实现(真实感渲染与简单的卡通的渲染)：

由于另外的着色器比较简单且基本为重复内容，因此着重说明模型着色器(参考代码：https://learnopengl.com/code_viewer_gh.php?code=src/2.lighting/5.4.light_casters_spot_soft/5.4.light_casters.fs;https://blog.csdn.net/allenjiao/article/details/81031396,项目中的 shader_s.h 文件同样参考于网站：learnopengl.com)

顶点：

首先通过 MVP 矩阵，得到归一化裁剪空间坐标

gl_Position = (投影矩阵 * 视图矩阵 * 模型矩阵) * vec4(顶点坐标输入, 1.0);

输出顶点

片段位置 = 模型矩阵 * 顶点位置输入

输出纹理坐标

片段纹理 = 纹理坐标输入;

输出经过法线矩阵处理后的法线方向

片段法线方向 = 模型矩阵的逆矩阵再转置 * 法线方向输入;

片段：

首先获得标准化的法线方向(使用 normalize 函数)

再获得由片段指向光源的方向向量

下面进行光源的设置：

首先是最基本的环境光，是环境光色和纹理基色的混合

之后是漫反射光，为漫反射系数(灯的方向和法线方向点乘结果，角度越大值越小(非负))、漫反射光以及纹理基色的混合

最后是镜面光，较为复杂：首先获得从片段指向视图的方向；再获得由光源指向片段向量的，关于片段法向量反射之后的方向向量；基于以上获得镜面光分量：视线方向和反射方向的点乘结果(非负)的光滑度次方；最后，得到镜面光为镜面光分量、镜面光颜色以及镜面纹理基色的混合

以上是设置模型对与光照的响应，下面则“创建”出一个点光源

首先确定光锥：光锥角度为灯的照射方向和灯指向片段方向的点乘结果；在得到内外光切之差；最后通过以上的量，得到钳制在 0-1 之间的，光锥角度与的点乘值与外光切之差，即为光强，并使其影响漫反射和镜面反射

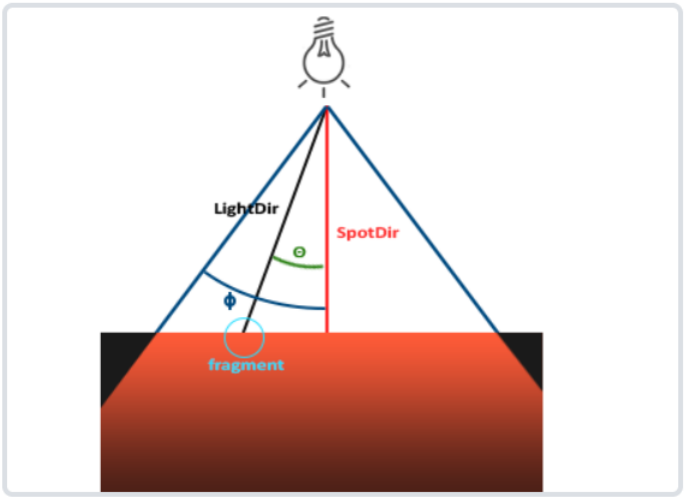
为使光线更加自然，使用衰减公式：$$F_{att} = \frac{1.0}{K_c + K_l * d + K_q * d^2}$$，其中 Kc,Kl,Kq 为三个系数，d 为片段与光线之间的距离

首先通过让光的位置向量减去片段位置向量，获得片段和灯之间的距离；在此基础上，根据传入的三个参数，得到衰减量并使它影响环境光、漫反射光以及镜面光

另外，NPR 渲染也在此着色器中体现：原理为根据灯光方向与法线方向之间的点乘结果，使不同结果的区域渲染成不同的纯色

其他的背景着色器和描边着色器使用了纹理、颜色的基本操作，不再赘述

补充：



- `LightDir`：从片段指向光源的向量。
- `SpotDir`：聚光所指向的方向。
- `Phi` ϕ：指定了聚光半径的切光角。落在这个角度之外的物体都不会被这个聚光所照亮。
- `Theta` θ：`LightDir`向量和`SpotDir`向量之间的夹角。在聚光内部的话θ值应该比ϕ值小。

(附图：聚光灯属性说明)

$$I = \frac{\theta - \gamma}{\epsilon}$$

这里ϵ(Epsilon)是内（ϕ）和外圆锥（γ）之间的余弦值差（$\epsilon = \phi - \gamma$）。最终的I值就是在当前片段聚光的强度。

（附图：光强计算公式）

图形用户界面：

本项目中，使用了第三方库 ImGUI 作为图形界面的部分

使用原因：ImGUI 相对于其他图形库更加轻量，不需要下载额外的软件或安装插件，将环境配置成功之后即可使用；另外，该图形库采用的使即时渲染模式，运行效率高并且可以达到 UI 设计中的易用性、简便性和美观性；项目初始立项时，本打算采用 QT 作为图形界面，但由于其体量较为庞大以及较难与 OpenGL 项目本身进行简单便捷的结合，因此最终放弃使用

Imgui 中编写难度最大的地方在于非界面部分的，有关联性的属性间逻辑联系，如内外光切的范围限定(内关切不能超过外光切)；NPR 颜色范围占比的限定(总和为 100%)等；另外，也通过查询相关资料，实现了中文界面的显示、窗口位置锁定等效果

项目中的核心部分使用伪代码：

(在渲染循环之前)

```
//初始化 Imgui
void InitImgui(Window 主窗口)
{
    检查 ImGui 版本;
    创建 ImGui 上下文;
    获取 ImGui 的输入输出系统;
    设置 ImGui 的主题颜色;
```

```
        设置 ImGui 使用中文字体(“选择中文字体所在的路径”);
        加载 GLFW(主窗口);
        设定 OpenGL 版本("#version 330");
    }
    (在渲染循环中)
//生成 Imgui 主窗口
void GenImguiMainWindow(Shader 模型着色器, Shader 背景着色器)
{
    生成 ImGui 窗口(u8"欢迎使用鲁班锁教学演示程序！",窗体不可移动);//u8 使用是为了使用中
文
    锁定窗口在左上角;
    换行;
    if (开启“简介”菜单)
    {
        换行并缩进;
        使用随窗口换行的文字进行提示;
    }
    换行;
    if (开启“交互说明”菜单)
    {
        {
            换行并缩进;
            if (开启树型布局)
            {
                换行并缩进;
                if (开启树型布局(“鼠标”))
                {
                    ……
                }
                if (开启树型布局(“键盘”))
                {
                    ……
                }
            }
        }
    }
    换行;
    if (开启“投影设置”菜单)
    {
        为透视投影和正交投影设定单选项，当使用透视投影时，提供滑动条来改变 FOV;
    }
    换行;
    if (开启“背景设置”菜单)
    {
        if (设置输入框，提示用户输入一个图片路径，按下回车后可以改变背景的纹理)
```

```
        {
            改变背景的纹理(纹理路径);
        }
    }
    换行;
    if (开启“模型设置”菜单)
    {
        if (开启一个下拉菜单，让用户选择使用何种贴图)
        {
            改变模型贴图;
        }
        提供勾选框，设定是否使用 NPR 渲染;
        if (使用 NPR 渲染)
        {
            通过四组滑动条和颜色设定器，改变 NPR 渲染的效果;
        }
    }
    换行;
    if (开启“灯光设置”菜单)
    {
        换行并缩进;
        为灯光深度设置滑动条;
        if (!isUseNPR)
        {
            为光的内关切和外光切设定两个滑动条;
            为环境光、漫反射光、镜面光以及光泽度设定颜色条与滑动条;
        }
        为是否将灯光锁定到屏幕中央设定勾选框;
        if (锁定灯光位置)
        {
            设定灯光位置为定值;
        }
    }
    if (开启“动画设置”菜单)
    {
        if (设定“重置”按钮)
        {
            动画时间结点回到初始值;
        }
        设定当动画播放时创建“暂停”按钮，否则为“播放”按钮;
        根据之前的设置改变动画播放时间结点;
        创建滑动条来显示和设定动画进度;
    }
}
```

半透明效果实现

（参考：

https://learnopengl-cn.github.io/04%20Advanced%20OpenGL/03%20Blending/）

1. 首先需要利用 `glEnable(GL_BLEND);`函数，启用混合

2. 调用 `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);`函数，进行混合设置，即使用源颜色向量和 alpha 作为源因子;1-alpha 作为目标因子

补充：

OpenGL中的混合是通过下面这个方程来实现的：

$$\bar{C}_{result} = \bar{C}_{source} * F_{source} + \bar{C}_{destination} * F_{destination}$$

- $\bar{C}_{source}$：源颜色向量。这是源自纹理的颜色向量。
- $\bar{C}_{destination}$：目标颜色向量。这是当前储存在颜色缓冲中的颜色向量。
- F_{source}：源因子值。指定了alpha值对源颜色的影响。
- $F_{destination}$：目标因子值。指定了alpha值对目标颜色的影响。

（附图：混合说明）

6. 运行结果与说明

实验环境的配置：

1. 基于 OpenGL 3.0 官方库——GLFW 和 GLAD；
2. 第三方库——ImGui；
3. 第三方库——glm.h(用于数学运算)
4. 第三方库——stb_image.h

7. 结语

由于时间限制，部分功能还有待完善，如更加精确的拾取功能；更加丰富的描边效果；阴影贴图，法线、视差贴图的使用等等。希望在之后的学习中能够继续努力，将所学到的知识进行实际运用并努力尝试实现更为丰富的效果。

8. 参考文献

1. **基于 GLFW 库的 OpenGL 最新版本使用指南及其教程：**
   https://learnopengl-cn.github.io/

2. *Cherno OpenGL Video Series*:
   https://www.bilibili.com/video/av68903616?from=search&seid=14679403914757051509

3. 《基于 GLFW 的 OpenGL 编程入门_艾孜尔江制作》视频教程 ：
   https://www.bilibili.com/video/av70143533/

4. OpenGL 编程指南 9：裁剪平面 glPushMatrix 和 glPopMatrix 矩阵栈顶操作：
   https://blog.csdn.net/shenziheng1/article/details/66968854

5. GLFW Input Documentation：
   https://www.glfw.org/docs/latest/input_guide.html#input_mouse_button

6. 《一周学习光线追踪——层次包围盒 BVH》：
   https://zhuanlan.zhihu.com/p/36439822

7. NPR 介绍：en.wikipedia.org/wiki/Non-photorealistic_rendering