软光栅化渲染器说明文档

Application：SoftRender_Ver.miHoYo

Developer：Zhen Liu @ NWPU

1 环境及依赖

开发环境：VS 2022 + QT 5.12.3

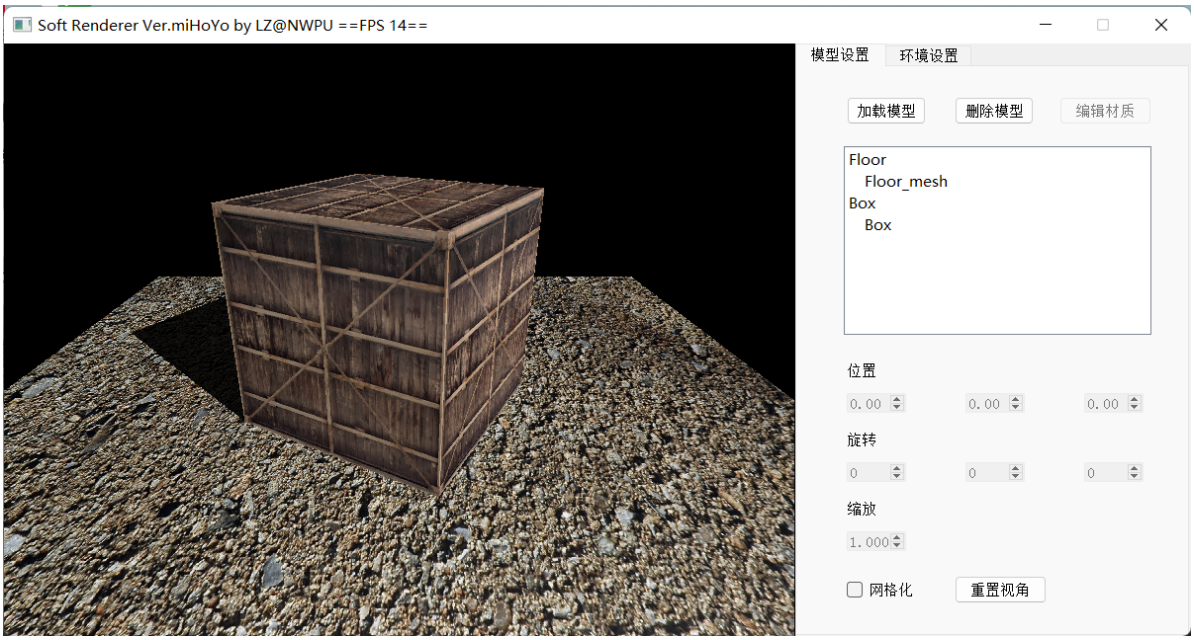
第三方库：glm 数学库

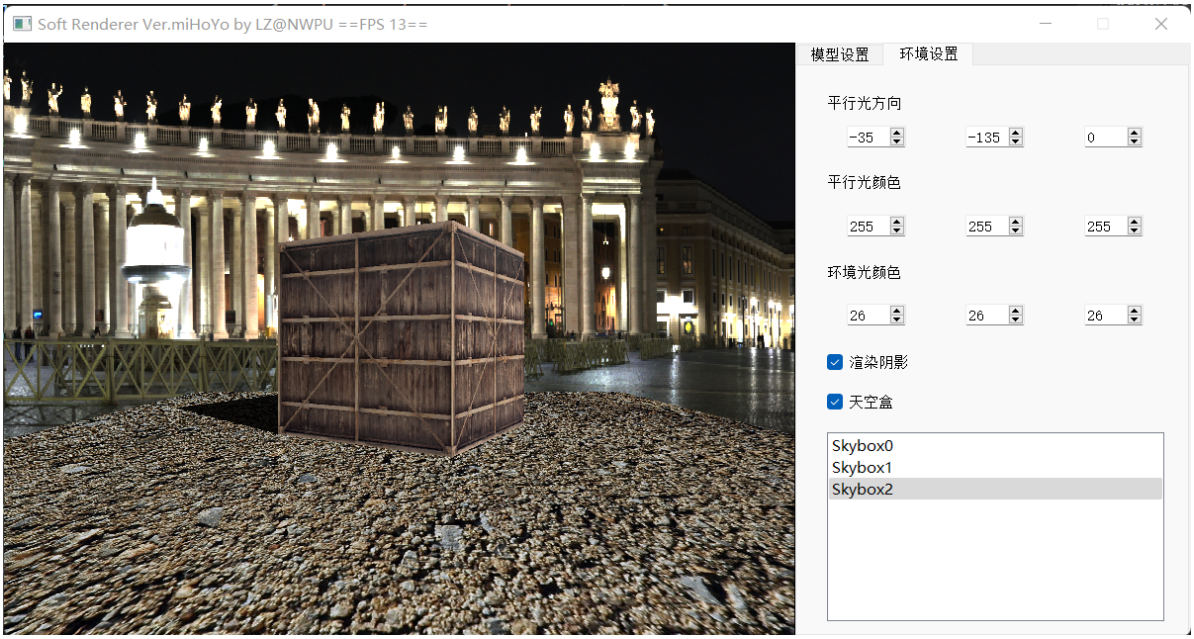
编译说明：需要在 VS 中安装 Qt Visual Studio Tools 插件并配置 QT 路径，之后正常编译即可

运行说明：src/x64/Release 文件夹下已经包含必要的 QT 动态库，可直接运行程序

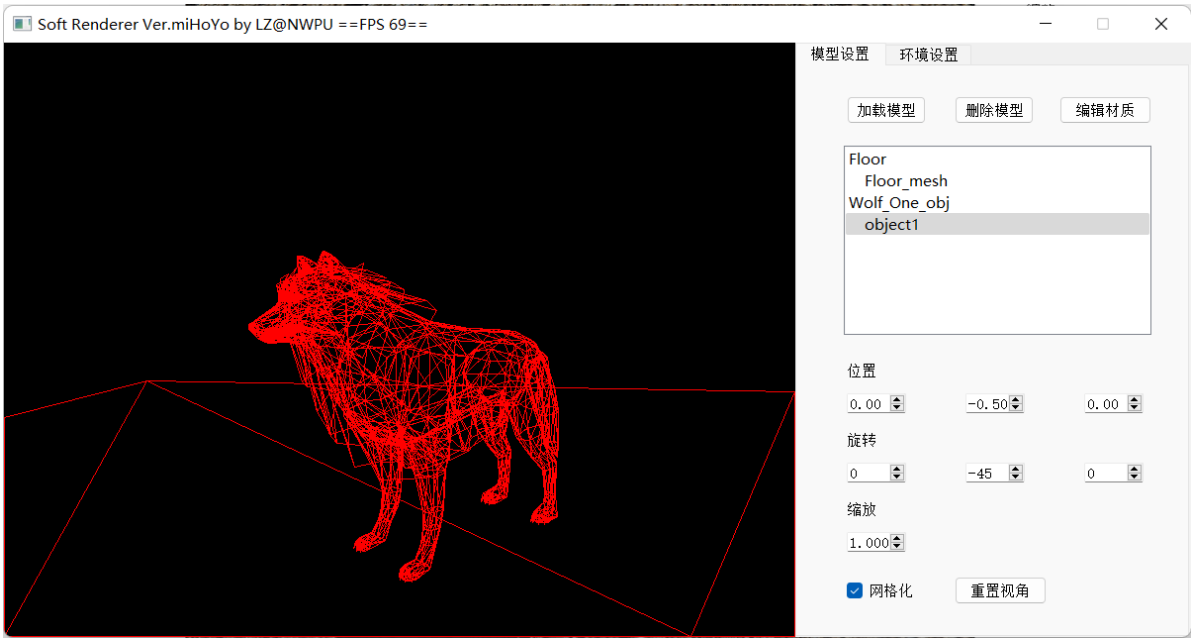
2 特性

- 基于 QT 的可交互界面
- 支持加载 .obj 模型和主纹理，支持模型、材质、环境等参数设置
- 完整模拟渲染管线基本流程
- 实现了基础的材质、纹理、光照、阴影、天空盒以及图元裁剪、光栅化等算法




Soft Renderer Ver.miHoYo by LZ@NWPU ==FPS 13==
模型设置
环境设置
平行光方向
-35
-135
0
平行光颜色
255
255
255
环境光颜色
26
26
26
渲染阴影
天空盒
Skybox0
Skybox1
Skybox2


Soft Renderer Ver.miHoYo by LZ@NWPU ==FPS 19==
模型设置
环境设置
加载模型
删除模型
编辑材质
Floor
Floor_mesh
Wolf_One_obj
object1
位置
0.00
-0.50
0.00
旋转
0
-45
0
缩放
1.000
网格化
重置视角


Soft Renderer Ver.miHoYo by LZ@NWPU ==FPS 69==
模型设置
环境设置
加载模型
删除模型
编辑材质
Floor
Floor_mesh
Wolf_One_obj
object1
位置
0.00
-0.50
0.00
旋转
0
-45
0
缩放
1.000
网格化
重置视角

3 实现思路

3.1 UI 框架及操作说明

程序的入口为 `main.cpp` 中的主函数，其中调用了程序的主窗口，程序主窗口在 `RenderWidget` 类中实现，该类会开始渲染循环 `RenderLoop` 并处理鼠标、键盘和绘制等事件，其中：

- **鼠标点击**事件处理定义在 `RenderWidget.cpp` 中的 `RenderWidget::mouseMoveEvent` 函数中，通过鼠标拖动可以改变相机的俯仰角和方向角，目前不支持滚转角变化；
- **鼠标滚轮**事件处理定义在 `RenderWidget.cpp` 中的 `RenderWidget::wheelEvent` 函数中，鼠标滚轮可以拉近或拉远视角，通过改变模型的缩放系数实现；
- **键盘输入**事件处理定义在 `RenderWidget.cpp` 中的 `RenderWidget::keyPressEvent`、`RenderWidget::keyReleaseEvent` 和 `RenderWidget::DealInput()` 三个函数中，每隔 100 ms 接收一次键盘输入，通过 W、A、S、D 可以改变相机水平位置，通过 Q、E 可以改变相机所在位置的垂直高度；
- **绘制事件**处理定义在 `RenderWidget.cpp` 中的 `RenderWidget::Vsync` 和 `RenderWidget::paintEvent` 函数中，每一帧绘制完成后，交换前后缓冲并向主窗口发出刷新信号，此时主窗口接收当前 FrameBuffer 中的颜色缓冲数据，并将其转化为 `QImage` 对象以绘制在界面的画布上。

主窗口还包含模型设置和环境设置两个标签页，分别实现在 `ModelTab.cpp` 和 `EnvTab.cpp` 中，功能都是接收输入并改变渲染时的一些参数和状态。选中模型还可以打开材质编辑面板，实现在 `MaterialDialog.cpp` 中，具体可参见代码及注释。

3.2 渲染流程

在主窗口 `RenderWidget` 类的构造函数中除了初始化 UI 外，还初始化了起始的渲染状态并调用了渲染主循环：

```
RenderWidget::RenderWidget(QWidget* parent)
    : QWidget(parent), fps(0), firstMouseMove(true), mdTab(nullptr)
{
    ...

    // 初始化状态机
    sys = StateMachine::GetInstance();
    sys->Init(800, 600, 60);

    renderLoop = new RenderLoop(nullptr);
    renderThread = new QThread(this);

    // 渲染线程和函数绑定
    renderLoop->moveToThread(renderThread);
    connect(renderThread, &QThread::finished, renderLoop,
&RenderLoop::deleteLater);
    connect(renderThread, &QThread::started, renderLoop,
&RenderLoop::MainLoop);
    connect(renderLoop, &RenderLoop::Vsync, this, &RenderWidget::Vsync);

    renderThread->start();

    ...
}
```

其中 `StateMachine` 是一个全局对象，类似于 OpenGL 中的上下文（context），负责记录整个渲染系统的状态，这些状态包括帧缓冲、相机状态、灯光参数、场景及模型数据、渲染模式、剔除模式、深度写入开关、颜色写入开关等等，以及改变这些状态的相关方法，初始化状态机时会对视口大小、帧缓冲、相机、光源等以及负责绘制的类 `Graphics` 的对象进行初始化，并且会构造默认的地板平面。具体实现在 `StateMachine.cpp` 中。

`RenderLoop` 类实现了渲染主循环，类似于 Vulkan 中的 Pipeline，负责设定每一个 Pass 渲染时的各种状态参数和渲染目标（Render Target），并按顺序调用不同的 Pass 对场景进行渲染，渲染主循环实现在 `RenderLoop.cpp` 中的 `RenderLoop::MainLoop()` 函数中：

```
void RenderLoop::MainLoop()
{
    // 初始模型
    Model* wBox = new Model(Mesh::CreateBox(glm::vec3(0, 0, 0), 0.5));
    Texture2D* mt = new Texture2D("../assets/Textures/container.jpg");
    wBox->meshes[0]->material->SetTexture(mt);
    wBox->yaw = 60;
    wBox->name = "Box";
    wBox->meshes[0]->name = "Box";
    sys->AddModel(wBox);

    // 渲染循环
    while (!shouldClosed) {
        /***** Shadow Pass *****/
        if (sys->drawShadow) {
            // 参数设置，2048 ShadowMap 分辨率
            sys->SetViewPortMatrix(2048, 2048);
            sys->writeColor = false;
            sys->writeDepth = true;
            sys->faceCullMode = Front;
            // 逐光源生成 ShadowMap
            for (int i = 0; i < sys->dirLights.size(); i++) {
                sys->dirLights[i]->SetShadowMap(nullptr);
                // 设定 RT
                FrameBuffer* SBO = new FrameBuffer(2048, 2048);
                sys->graphics->SetRenderTarget(SBO);
                SBO->ClearDepth(1.0f);
                // 绘制 ShadowMap
                sys->DrawShadow(sys->dirLights[i]);
                // 存储 ShadowMap
                sys->dirLights[i]->SetShadowMap(new Texture2D(SBO-
>depthBuffer));
                delete SBO;
            }
        }
        else {
            for (int i = 0; i < sys->dirLights.size(); i++) {
                sys->dirLights[i]->SetShadowMap(nullptr);
            }
        }
        /***** Forward Render Pass *****/
        // 参数设置
        sys->SetViewPortMatrix(sys->width, sys->height);
        sys->writeColor = true;
```

```
        sys->writeDepth = true;
        sys->faceCullMode = Back;
        // 设定 RT
        sys->graphics->SetRenderTarget(sys->backBuffer);
        sys->ClearColor(0, 0, 0);
        sys->ClearDepth(1.0f);
        // 绘制场景
        sys->DrawScene();
        // 双重缓冲交换
        sys->SwapBuffer();
        // 发送屏幕刷新信号，在画布上显示当前帧
        emit Vsync(sys->GetFrame(), 0, 0);
        sys->fps++;
    }
}
```

上面说到 `StateMachine` 中会初始化负责绘制的类 `Graphics` 的对象，该类中实现了全部渲染算法，负责执行状态机发送的 "Drawcall"，这里的 Drawcall 即为上面代码中一系列以 Draw 开头的函数，这些函数内部都会调用 `Graphics` 中的方法进行绘制。

以 `sys->DrawScene()` 为例：

```
// 绘制场景
void StateMachine::DrawScene()
{
    mutex.lock();
    // 缩放系数矩阵
    glm::mat4 globelScale = glm::scale(glm::mat4(1.0f),
glm::vec3(modelScale, modelScale, modelScale));
    // View矩阵
    glm::mat4 viewMatrix = camera->GetViewMatrix();
    // 投影矩阵
    glm::mat4 projectionMatrix = camera->GetProjectionMatrix();
    // 三角形数量，调试用
    triangle = 0;

    // 每个模型逐光源绘制
    for (int i = 0; i < models.size(); i++) {
        for (int j = 0; j < dirLights.size(); j++) {
            Uniform u(globelScale * models[i]->GetModelMatrix(), viewMatrix,
projectionMatrix);
            u.cameraPos = glm::vec4(camera->position, 1.0f);
            u.ambient = ambient;
            u.dirLight = dirLights[j];
            graphics->DrawModel(*models[i], u, 2);
        }
    }
    // 绘制 sky box
    if (drawSkyBox) {
        faceCullMode = Front;
        writeDepth = false;
        Uniform u(glm::mat4(1.0f), viewMatrix, projectionMatrix);
        u.cubemap = skyboxMap[currentSkybox];
        graphics->DrawModel(*skyBox, u, 2);
        faceCullMode = Back;
```

```
    }
    mutex.unlock();
}
```

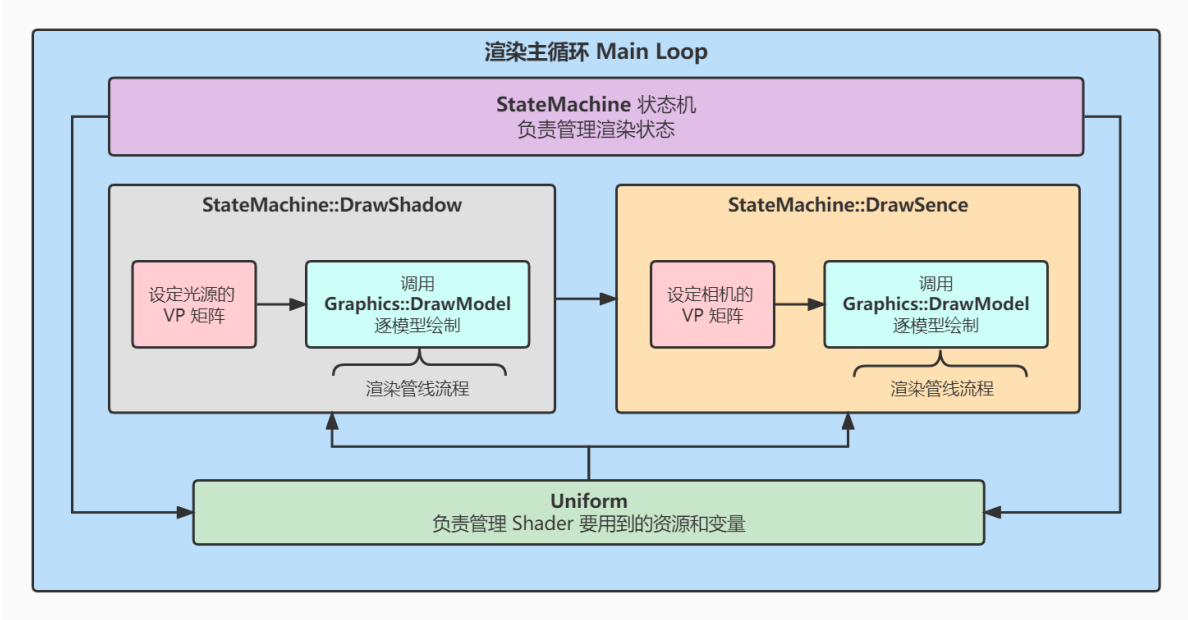
该函数中的工作就是获取 MVP 矩阵，并初始化一个 `Uniform` 对象，然后发送 Drawcall，`Uniform` 对象的功能类似于 Vulkan 中的 Pipeline Layout，其中存放了 Shader 中可以用到的所有资源及参数，包括各种全局变量和纹理等，`Uniform` 定义在 `ShaderBase.h` 中。

`Graphics` 对象会从模型开始渲染，即 `Graphics::DrawModel` 函数，该函数接收三个参数：模型 `Model` 对象、`Uniform` 对象和 pass 索引，该函数会绘制给定的 Model，Model 由 Mesh 组成，因此会逐 Mesh 绘制，并将 Mesh 对应的材质参数填充到 `Uniform` 对象中，以便 Shader 访问，材质 `Material` 还绑定了 Shader，该函数会根据给定的 pass 索引找到对应的 Shader 进行此次绘制。

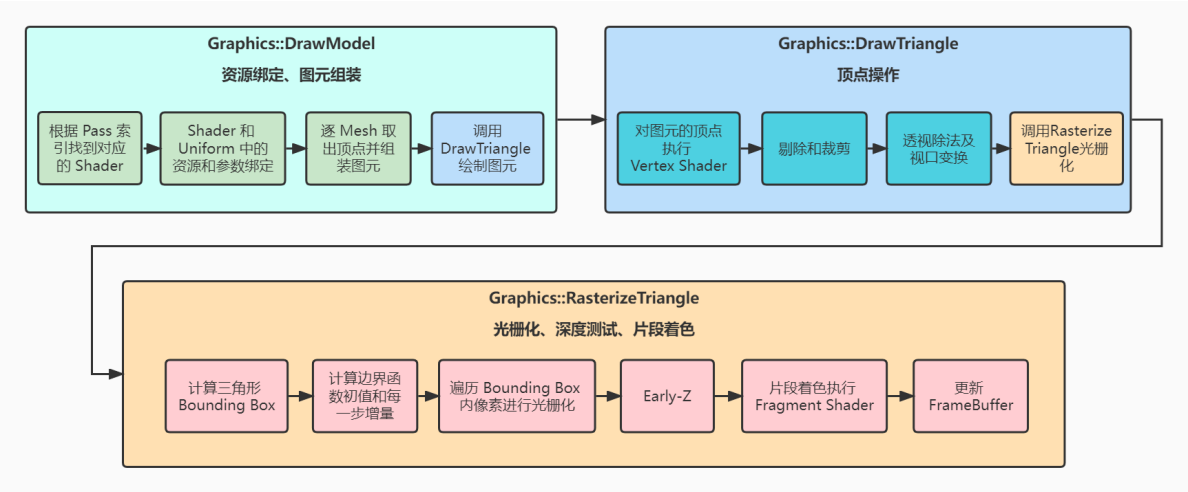
对于 Mesh，由顶点缓冲 VBO 和索引缓冲 EBO 组成，逐 Mesh 绘制即为从 Mesh 的顶点缓冲和索引缓冲中组装三角形图元，然后调用 `Graphics::DrawTriangle` 逐图元进行绘制。

`Graphics::DrawTriangle` 函数接收三个顶点，首先对顶点执行 Vertex Shader，之后进行剔除以丢弃不需要渲染的图元，然后对部分在裁剪空间立方体内的图元进行裁剪生成新的顶点和图元，最后对所有顶点组成的图元调用 `Graphics::RasterizeTriangle` 函数进行光栅化，在光栅化函数中会进行 Early-Z 深度测试，并为通过测试的片段执行 Fragment Shader，最后根据是否开启颜色写入和深度写入来更新 FrameBuffer。

整个渲染流程中各对象的关系如下图所示：



其中 `Graphics::DrawModel` 是渲染管线流程的入口，渲染管线流程如下图所示：



3.3 其他实现细节

按照管线流程顺序，本节介绍其中一些关键算法的实现思路。

3.3.1 图元裁剪算法

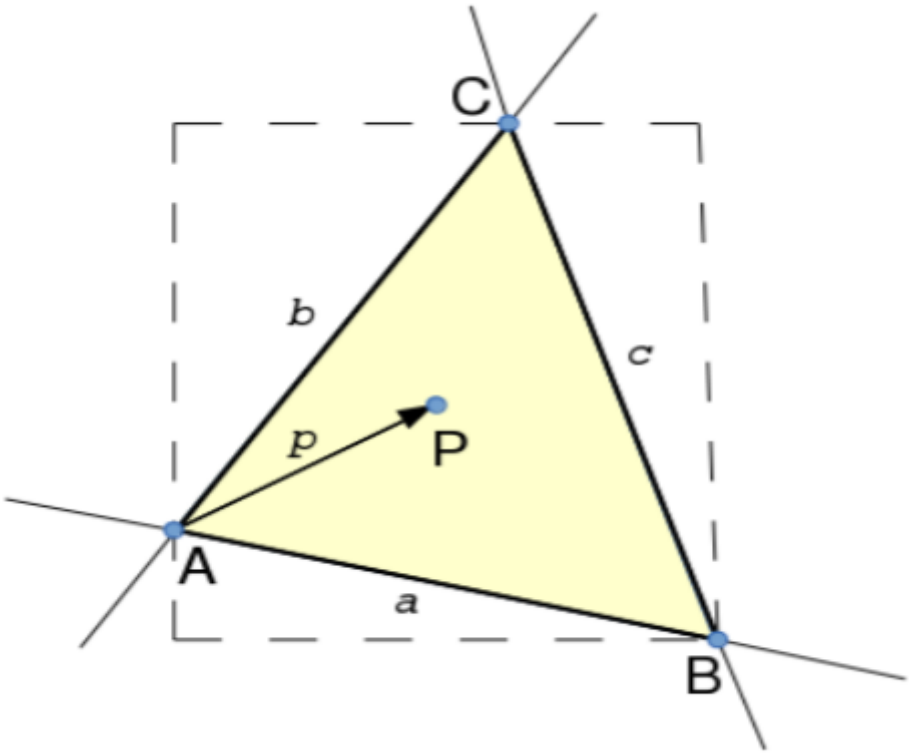
图元裁剪算法在顶点变换到裁剪空间之后，透视除法之前执行，基本流程是：

- 判断图元是否完全在裁剪立方体外部，对于完全在外部的图元直接丢弃
- 判断图元是否完全在裁剪立方体内部，对于完全在内部的图元无需裁剪
- 对部分在内部的图元执行裁剪算法，遍历六个裁剪平面，对两两顶点组成的直线进行裁剪
- 如果两顶点在裁剪平面两侧，则求直线和平面的交点（通过点到平面的距离插值得到）作为新的顶点加入结果顶点集
- 每一个平面的裁剪结果（顶点集）作为下一次裁剪的输入，直到遍历完六个裁剪平面

具体实现在 `Clip.cpp` 的 `Clip::SutherlandHodgeman` 方法中。

3.3.2 光栅化算法

使用了边界函数光栅化算法，边界函数即根据向量叉乘判断点是否在三角形内部，如下图：



则点 P 相对于边 AB 的边界函数可以表示为：

$$F_{AB}(P) = AB \times AP = (B_x - A_x)(P_y - A_y) - (B_y - A_y)(P_x - A_x)$$

其他两边同理，对于逆时针顶点顺序的三角形，三个边界函数都大于 0 即为点在三角形内部。上面的边界函数展开并整理可以改写为：

$$F_{AB}(P) = (A_y - B_y)P_x + (B_x - A_x)P_y + (A_xB_y - A_yB_x)$$

即边界函数对于 Px 和 Py 是完全线性的，也就是两个像素之间的边界函数差值是固定的：

$$F_{AB}(P_{x+1}, P_y) - F_{AB}(P_x, P_y) = A_y - B_y$$

$$F_{AB}(P_x, P_{y+1}) - F_{AB}(P_x, P_y) = B_x - A_x$$

于是只需要计算出边界函数初值，之后遍历每个像素只需要加上固定的增量即可快速得到该点的边界函数，无需重新计算叉乘。

如果当前点在三角形内部，则使用重心坐标插值得到该点的属性，重心坐标就是面积比值，也可以通过叉乘得到，因此还可以通过边界函数快速得到该点的重心坐标。

综上，边界函数光栅化算法基本流程为：

- 求三角形 Bounding Box
- 求三条边的边界函数初值
- 遍历 Bounding Box 中的每个像素，使用边界函数判断是否在三角形内部
- 对在内部的点，通过边界函数得到重心坐标，插值得到片段属性并执行 Fragment Shader，更新 Frame Buffer
- 使用增量更新边界函数继续遍历

具体实现在 `Graphics.cpp` 的 `Graphics::RasterizeTriangle` 方法中。

3.3.3 透视插值校正

由于光栅化过程中的插值在屏幕空间进行，而屏幕空间的顶点位置和世界空间的顶点位置不具备线性对应关系，因此直接使用线性插值会导致顶点属性错误，比如纹理坐标错误会使纹理贴图显示异常，因此要在光栅化之前，透视除法的时候对顶点的所有属性进行插值校正（除以 Z 值），在光栅化算法插值后进行透视恢复（乘以原本的 Z 值）。

这部分具体实现在 `Graphics.cpp` 的 `Graphics::PerspectiveDivision` 和 `Graphics::PerspectiveRestore` 中。

4 参考及资源

- UI 框架参考：Clawko - zhihu.com
- Box 纹理：https://learnopengl-cn.github.io/img/01/06/container.jpg
- Floor 纹理：DSC_3521 Textures from TextureKing
- Wolf 模型及纹理：Wolf Rigged and Game Ready - Free3D
- Sky Box HDRI：HDRIs • Poly Haven