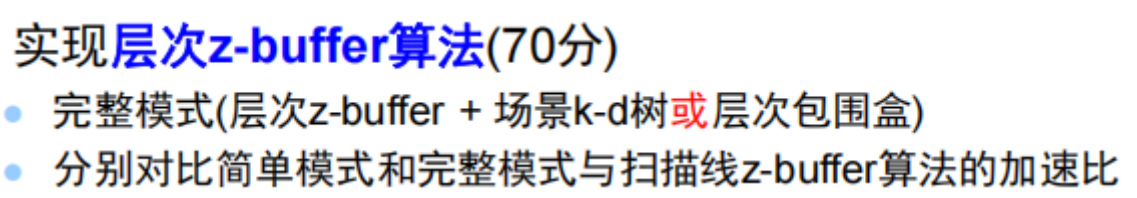
层次化Zbuffer说明文档

孙川 22421279

摘要：

- 实现了**层次z-buffer算法**的完整模式：



- 工程的大致思路：就是用CPU模拟GPU在光栅化部分的工作，将cpu的计算结果放到贴图上，然后在imgui中展示这个贴图。
- 实现了**obj文件管理器**
- 实现了 **GPU版本的光栅化框架**，用来在早期和CPU的结果进行比较校验。
- 实现了**CPU版本的简单光栅化框架**
- 实现了CPU版本的**扫描线算法**，（尝试了GPU不过貌似shader不适合写这个）
- 实现了**简单的zbuffer算法**
- 实现了**层次包围盒**，可以构建出场景bvh树
- 实现了**层次化zbuffer算法**
- 编译环境：
  - **Microsoft Visual Studio 2022**通过测试
  - 使用**WSL GCC**编译测试通过，所以理论上linux环境也可以
- 使用了层次包围盒进行加速，此外还有openmp在cpu端上针对循环的加速。
- 独立完成，用git管理版本，每次版本增删记录都可查

1 工程文件目录

```
CMakeLists.txt*
assets/
include/
shaders/
src/
submodules/
```

- `CMakeLists.txt` 我的CMake配置文件
- `assets/` obj文件的默认路径文件夹，把模型文件放在这里
- `include/` 头文件目录
- `shaders` GPU版本的shader目录
- `src` 源文件目录
- `submodules` 一些需要的库文件

2 外部依赖

我加了几个库，这样就可以不用在本地安装opengl什么库，直接编译这几个外部依赖就好了。

这样做的好处是，不需要针对windows 和linux在进行别的奇怪的配置；

我把这些东西放在submodules中，Cmake会自动编译他们：

- gl3w： opengl库，主要用来进行窗口化
- glfw：opengl库，用来开启一个窗口
- glm：opengl的数学库，用来统一坐标系，这样我的cpu版本就和gpu版本有相同的结果
- imgui：用来把我的cpu绘制的结果放到窗口上面。

3 编译方法

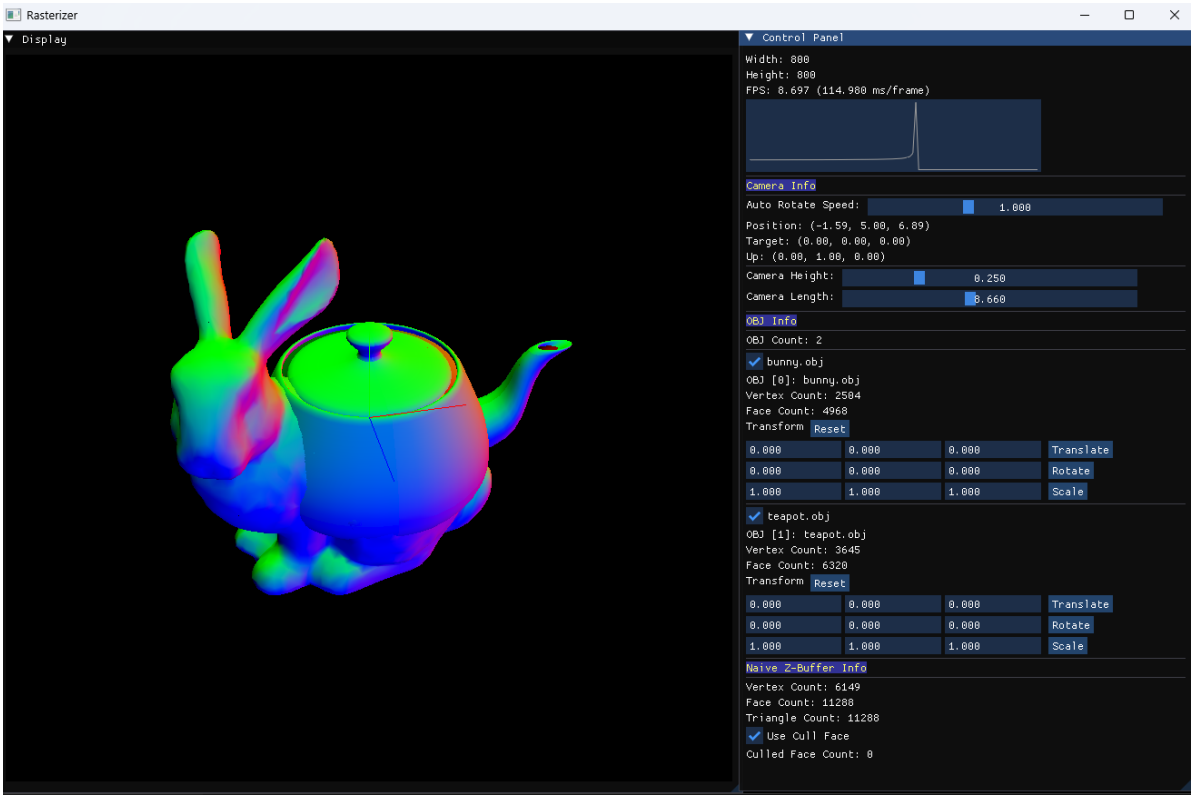
使用CMakeLists.txt可以直接构建，支持windows端的Visual Studio以及Linux端的cmake。

4 使用方法/UI界面

得益于我的C++封装和合理的命名方式，代码可读性非常好，在main函数中可以指定使用特定的渲染器（简单绘制、扫描线算法、简单zbuffer、层次化zbuffer），在这里选择编译使用哪种绘制方式：

```
int main()
{
    bool isGPU = true;
    isGPU = false;
    // auto rasterizer = std::make_unique<EzRasterizer>(SCRA::Config::WIDTH,
SCRA::Config::HEIGHT, isGPU);
    // auto rasterizer = std::make_unique<ScanlineRaster>(SCRA::Config::WIDTH,
SCRA::Config::HEIGHT, isGPU);
    // auto rasterizer = std::make_unique<NaiveZBufferRaster>
(SCRA::Config::WIDTH, SCRA::Config::HEIGHT, isGPU);
    auto rasterizer = std::make_unique<HeirarZBufferRaster>(SCRA::Config::WIDTH,
SCRA::Config::HEIGHT, isGPU);
    //  在这里选择用哪个

    rasterizer->setCamera(glm::vec3(5.0f, 5.0f, 5.0f), glm::vec3(0.0f, 0.0f,
0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
    // 这是设置相机的初始位置（实际上程序里面也能操作相机）

    rasterizer->loadOBJ("./assets/bunnyL.obj", "basic");
    rasterizer->loadOBJ("./assets/bunny14k.obj", "basic");
    rasterizer->loadOBJ("./assets/complex_sphere.obj", "basic");
    // 这里可以选择加载哪些obj模型

    rasterizer->implementTransform("bunnyL.obj",
SCRA::Utils::TranslateMatrix(0.0f, -0.1f, 0.0f));
    rasterizer->implementTransform("bunny14k.obj",
SCRA::Utils::ScaleMatrix(20.0f, 20.0f, 20.0f));
    // 这里是针对上述模型进行形变

    rasterizer->run(); // 绘制程序入口
    std::cout << "Bye" << std::endl;
}
```

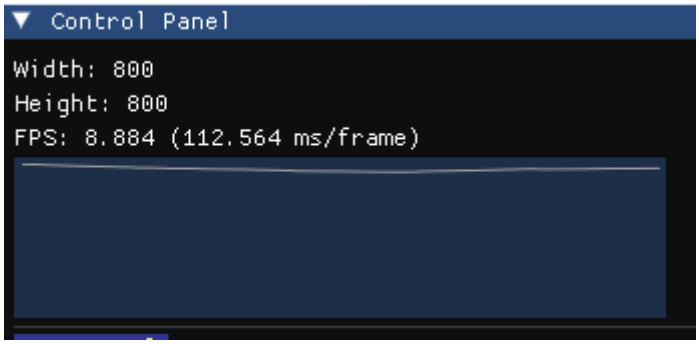
得益于imgui的封装，我的ui界面也被我设计的特别用户友好：



左边是我用的cpu的绘制结果，右边是我的控制面板。具体来说：

窗口信息：

展示了窗口的大小和当前帧率，以及之前的帧率情况。



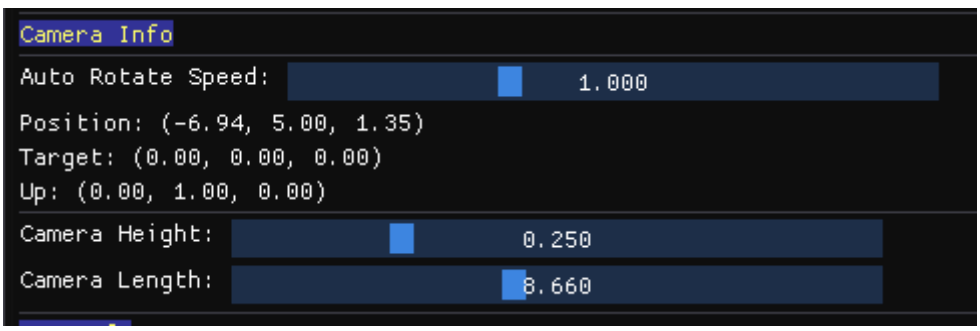
相机信息：

相机会默认围绕竖直y轴旋转。

auto rotate speed控制相机的旋转速度。

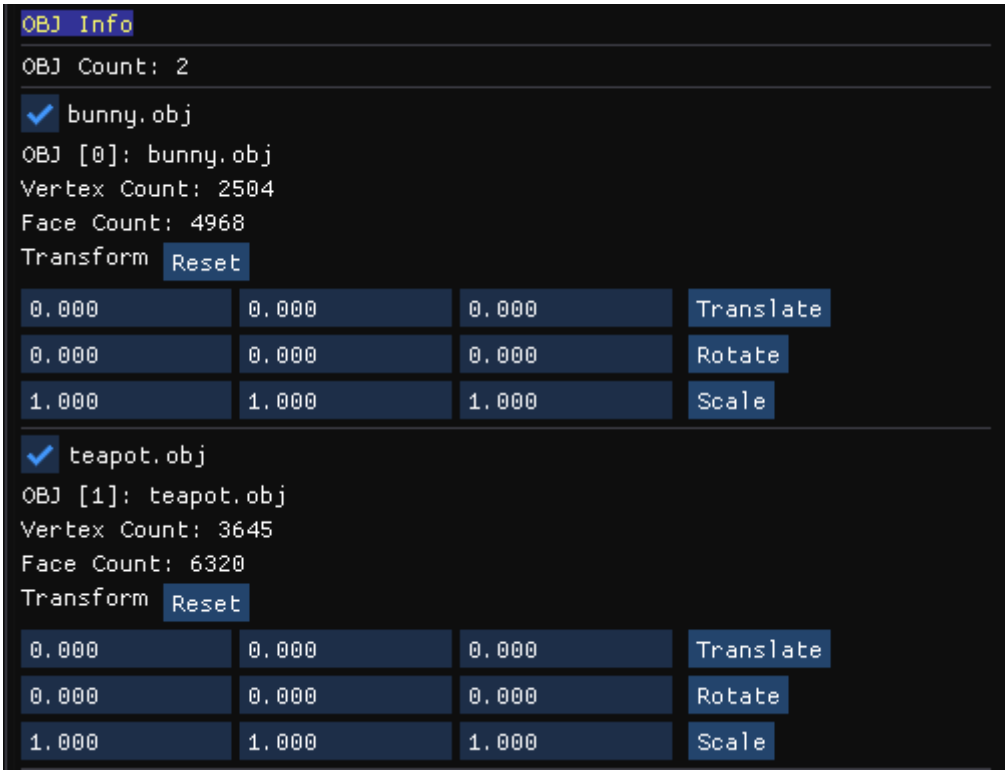
下面是相机的信息。

后面两个可以调节相机的高度，以及相机距离(0,0,0)的距离（视角相机被我固定 look at 坐标原点方便观察）

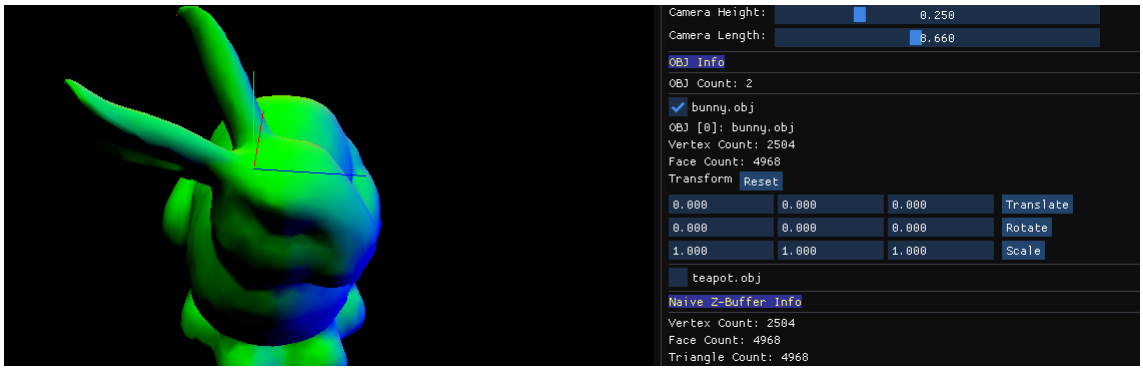


场景物体信息：

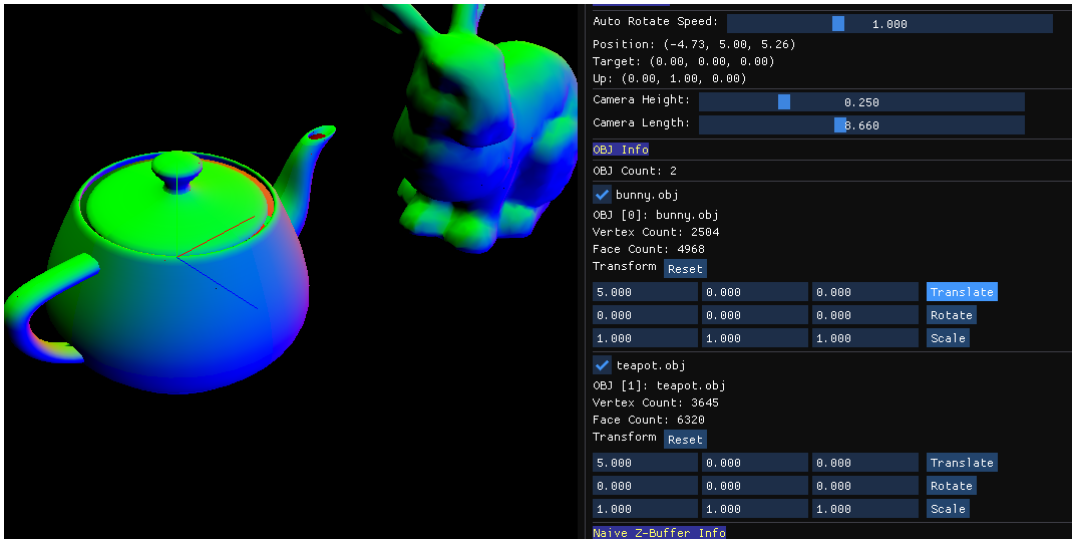
这里面展示了我们在main函数中加载的各个obj物体的信息，



取消勾选会隐藏物体：



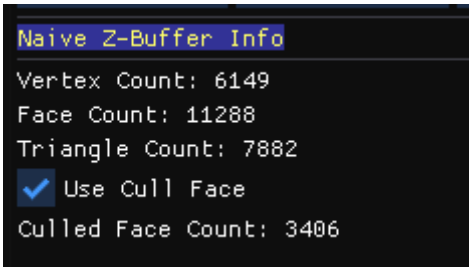
在这里也可以对每个物体进行移动、旋转和缩放，重置的话按reset按钮：



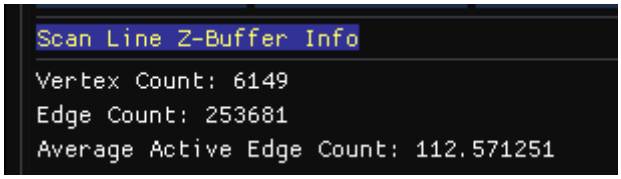
渲染器信息：

会显示当前使用的渲染器的一些debug信息，不同的渲染器有不同的结果。

这个是简单zbuffer的结果，会显示一些

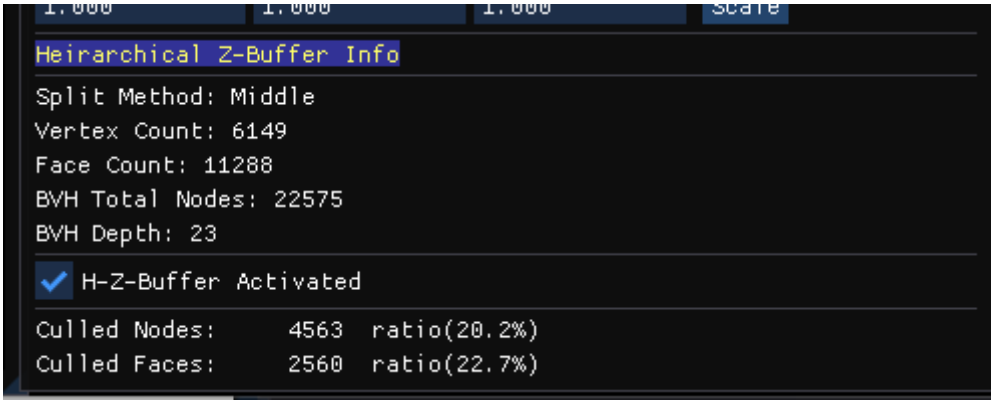


这个是扫描线算法的结果，会显示每个扫描线的平均活化边（毕竟扫描线算法中的面不是很重要，但是边edge很重要，活化边的数量是衡量计算复杂程度的一个很大的因素）

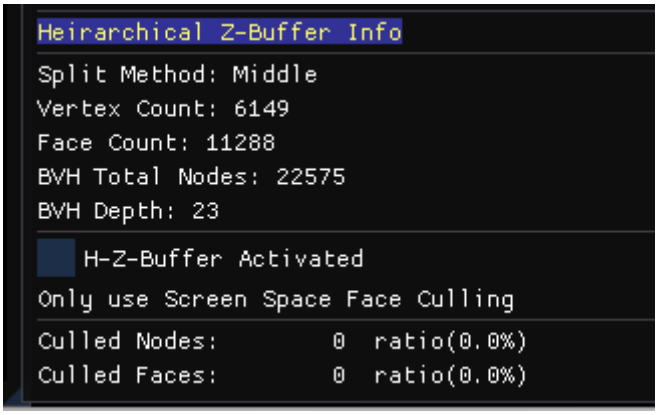


这个是层次化zbuffer的结果，在这里可以选择是否使用“BVH拒绝”，也就是说作业中所说的“完整模式”和“简单模式”。这里还会显示一些BVH的信息，比如说一共多少个节点，使用什么分割方式等等（这里我使用的是中值分割，我也实现了SAH和等分分割一共三种方式，不过目前SAH还没调试完）

完整模式：



简单模式（都在屏幕内所以全是0）：



5 算法实现细节

5.0 着色方式

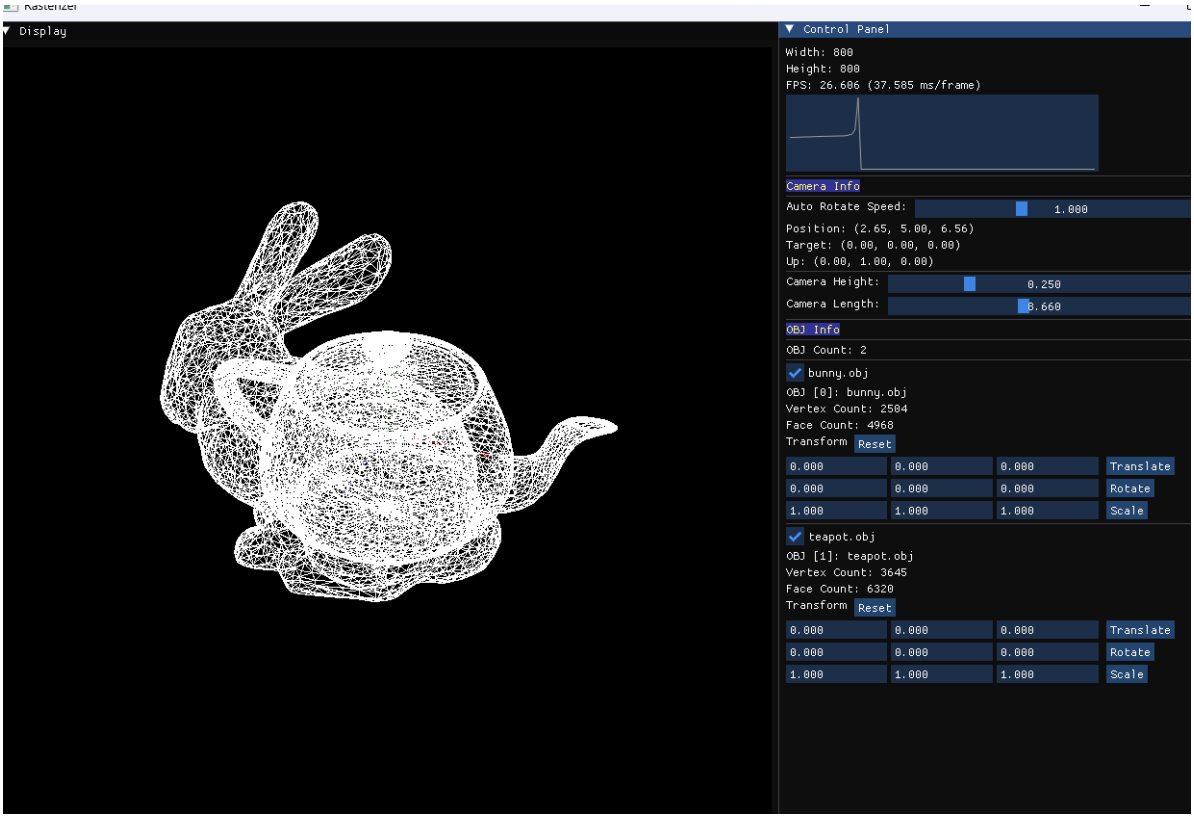
我是计算了每个obj的顶点normal，然后用顶点normal的xyz作为rgb插值出最终的颜色。

顶点normal是我自己算的，所以提供的obj文件最好也是自身没有vn也就是顶点Normal信息的才好，我测试了blender导出来了有vn信息的模型，最后的结果很怪异，我也不太清楚为什么。

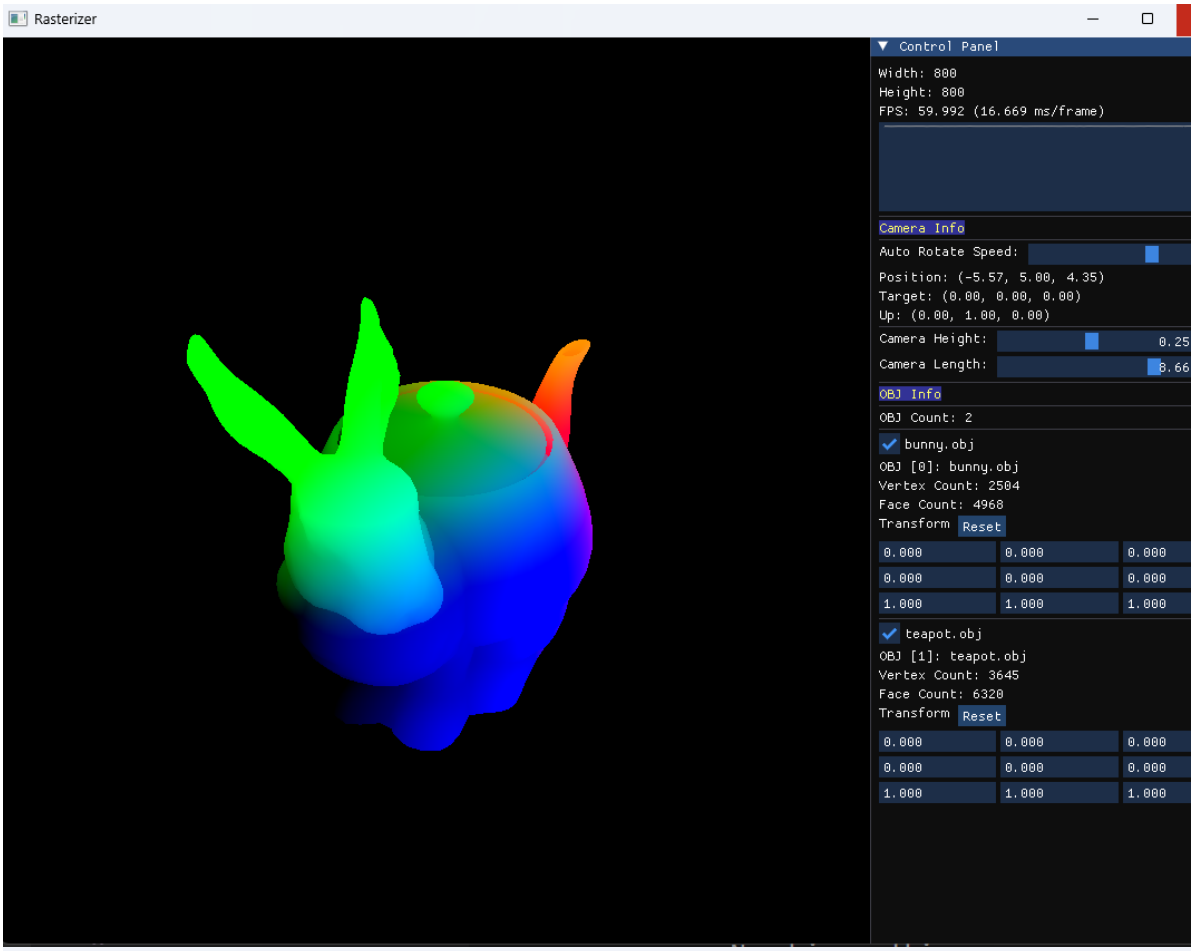
5.1 简单光栅化

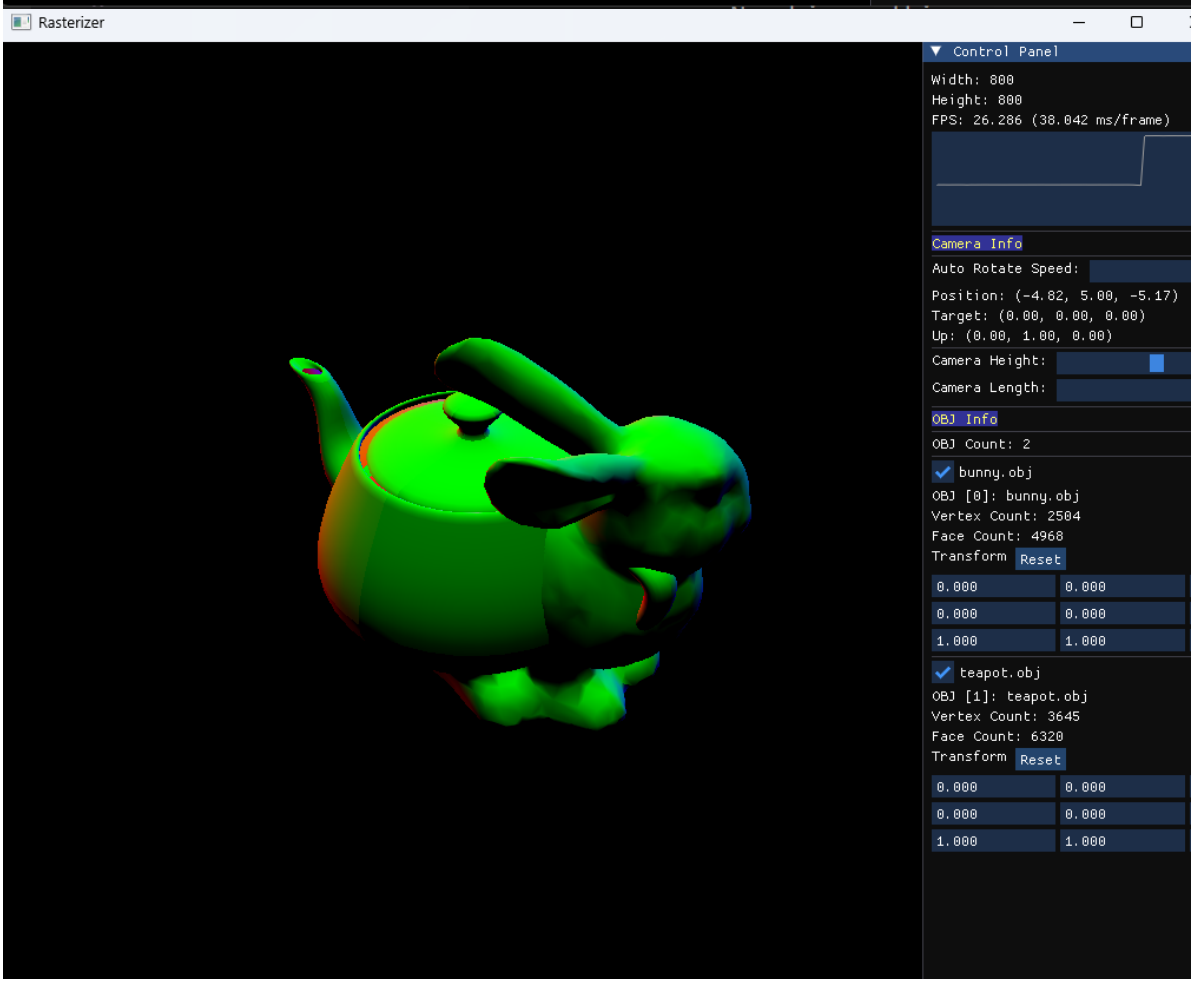
很简单，照着learnopengl来了一遍，实现了cpu和gpu版本

这里面的CPU版本我原本只想着只画frame线框不画面片，但是感觉这么做就不能体现遮挡关系了。



GPU版本，用来和其他算法的结果进行比较，GPU性能摆在这，没办法帧率CPU没得比（第一张是用position着色，第二张是用normal着色）：

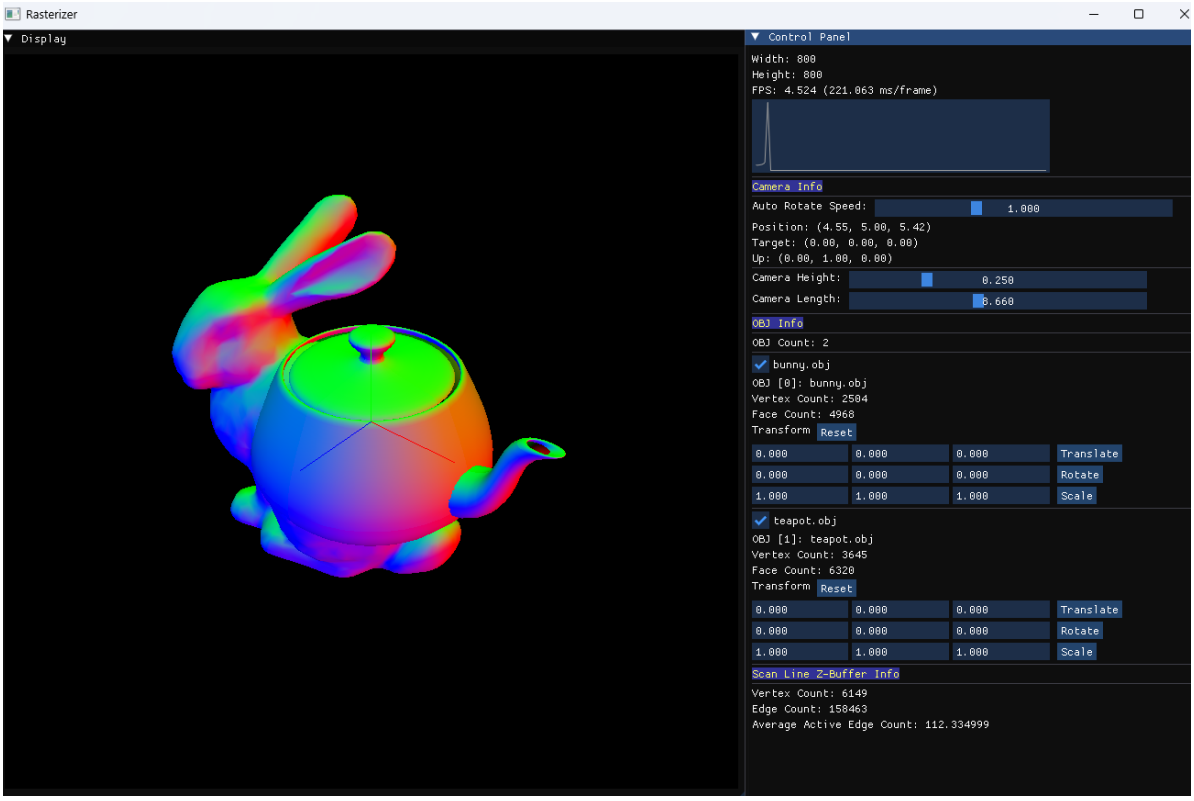

Rasterizer
Control Panel
Width: 800
Height: 800
FPS: 59.992 (16.669 ms/frame)
Camera Info
Auto Rotate Speed:
Position: (-5.57, 5.00, 4.35)
Target: (0.00, 0.00, 0.00)
Up: (0.00, 1.00, 0.00)
Camera Height:
0.25
Camera Length:
8.66
OBJ Info
OBJ Count: 2
bunny.obj
OBJ [0]: bunny.obj
Vertex Count: 2504
Face Count: 4968
Transform
Reset
0.000
0.000
0.000
0.000
0.000
0.000
1.000
1.000
1.000
teapot.obj
OBJ [1]: teapot.obj
Vertex Count: 3645
Face Count: 6320
Transform
Reset
0.000
0.000
0.000
0.000
0.000
0.000
1.000
1.000
1.000


Rasterizer
Control Panel
Width: 800
Height: 800
FPS: 26.286 (38.042 ms/frame)
Camera Info
Auto Rotate Speed:
Position: (-4.82, 5.00, -5.17)
Target: (0.00, 0.00, 0.00)
Up: (0.00, 1.00, 0.00)
Camera Height:
Camera Length:
OBJ Info
OBJ Count: 2
bunny.obj
OBJ [0]: bunny.obj
Vertex Count: 2504
Face Count: 4968
Transform
Reset
0.000
0.000
0.000
0.000
1.000
1.000
teapot.obj
OBJ [1]: teapot.obj
Vertex Count: 3645
Face Count: 6320
Transform
Reset
0.000
0.000
0.000
0.000
1.000
1.000

5.2 扫描线算法

这个算法用了我最长的时间，原本想用compute shader实现，奈何最后感觉这个算法就是为cpu光栅化而生的，因为毕竟从上到下扫描这个工作不能并行，于是老老实实cpu实现。

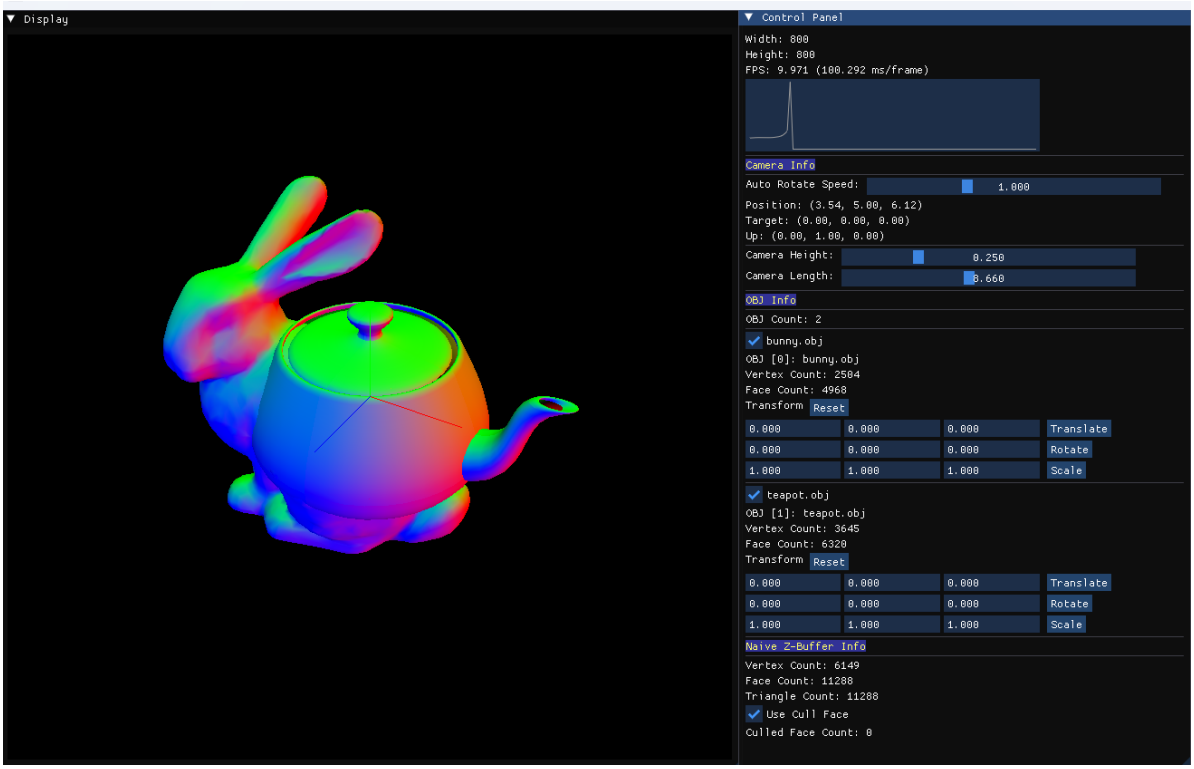
原理不难，不过debug用了不少时间。我参考了网上许多代码，不过感觉他们的不够巧妙，更有甚者甚至弄出了"活化多边形"这种多余的操作。这里面我想了一个更适合我这个框架的思路，设计了两个激活反激活表，用来替代矩阵式的活化边表，也就是一个是用于记录active边，另一个用于记录deactive边，这么做可以大大减少内存开销，代码实现起来也方便。



其实我也弄了gpu模式，不过原理是用cpu算出scanline的结果然后穿给gpu，再在computeshader中装配这些信息，事后感觉多此一举了。这部分代码保留了但没有实装。

5.3 简单zbuffer

最简单的zbuffer，不过不同于扫描线算法只绘制一遍的这个算法特性，naive zbuffer的涂色部分是最浪费时间的。这个涂色是每个三角形都要涂一遍，很麻烦，实际上也是后续H-zbuffer出现的主要原因。



5.4 BVH（层次包围盒）

本科的时候上过周昆老师的浙大的计算机图形学高级专题（好像是叫这个），在实现双向路径追踪的时候，写过一个存放光顶点的bvh。

这里实际上就是把存的东西从点云信息变成存面片信息。

大致思路是这样的：有一大堆面片，然后要做的事就是，把他们按某个轴分割成两部分，这里会进行一次排序。我是将整个面片提取其面片的空间包围盒进行计算的。

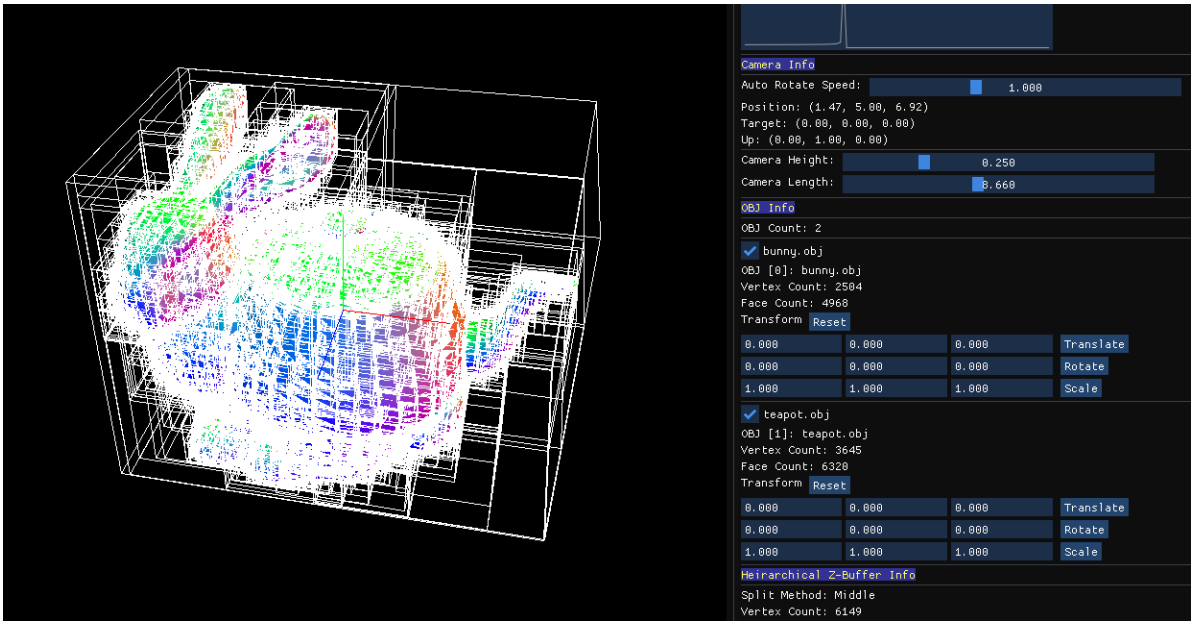
分割的话会有不同的方式，我使用的是中点分割，也就是在中间值的位置分割。也有其他分割方式比如SAH（一种启发式方法）和等量分割。

如果分割得太小了或者只剩下一个叶子节点，就停止分割。

注意我这里使用的世界坐标做的bvh，因为我测试我使用相机空间的话每次绘制都要花时间构建bvh，太变态了。用世界坐标系构建bvh的问题是分割可能不是和相机空间的z正交的，但是时间上的效率弥补了这一点，只需要每次变相机坐标的时候遍历一遍整个bvh算一下w2c变换就好了。

当然了这也导致了一个问题，就是每次场景变化就需要重新构建bvh，所以层次化zbuffer中我还没实现ui界面的场景编辑功能（就是即使是你按了那些按钮，也什么都不会发生），因为一变场景就要重新构建bvh。

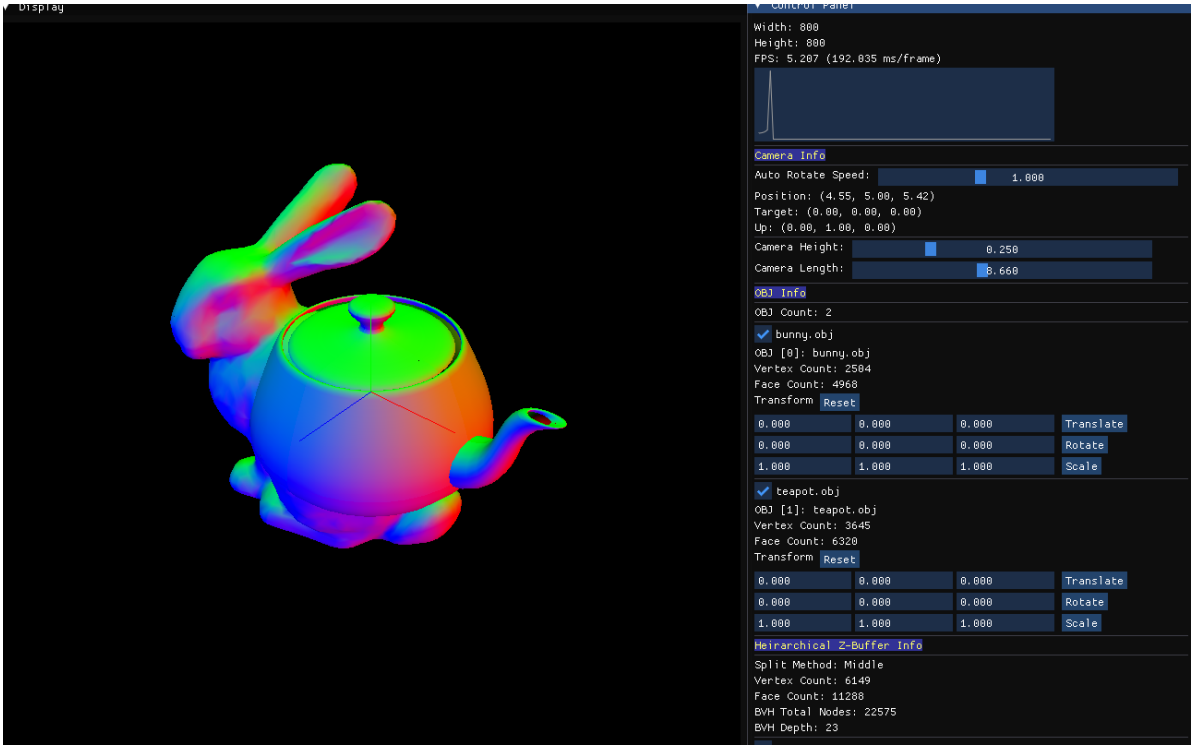
这个是bvh的可视化，uncomment bvh.cpp的278行就可以得到这个结果：



5.5 层次化zbuffer

两部分：一个是像素空间的mipmap式的zmax管理，一个是bvh拒绝。

也就是对应着题目要求的简单模式和完整模式。



6 结果分析

数值是FPS

面片数量	Scanline	Naive Zbuffer	H-Zbuffer	H-Zbuffer + BVH
5k（bunny）	10.8	18	10	12
10k（bunny+teapot）	3.9	10	6.5	7.5
14k（2bunny+teapot）	2.6	6.8	3.9	4.6
144k (bunnyLarge,144046 faces)	0.6	2.4	1.193	1.44

其实感觉最终结果基本上是符合预期的。scanline由于edge的原因，不管是否遮挡都会计算每个edge，因此edge一多就完蛋；相比之下zbuffer更倾向于和像素相关。一方面我用的世界空间建立的bvh，这导致遍历bvh的顺序实际上和相机无关，也就是不同的相机方向最后的帧率还是会有一定的上下浮动。而且每次绘制都会提前遍历bvh，用来计算bvh在相机空间中的坐标，这对于大的bvh来说也是一笔不小的开销。感觉和PTvsBDPT一样，zbuffer vs H-zbuffer也会有这种，构建加速结构导致的开销问题，可能我找到的这几个obj模型还不够大，没能充分体现出hzbuffer的优点~

7 总结

时间其实有限，这个作业还没弄的特别完美，很多部分比如说层次化zbuffer可能还有一些小的瑕疵。不过这段时间搞这个确实受益匪浅，感觉手搓了一个光栅化渲染器+blender的感觉（虽然没有那么多功能）。有重复造轮子的自我怀疑，但是更多的是自己的代码跑起来后得到自己预期中的画面的时候的快乐。