层次 z-buffer 算法

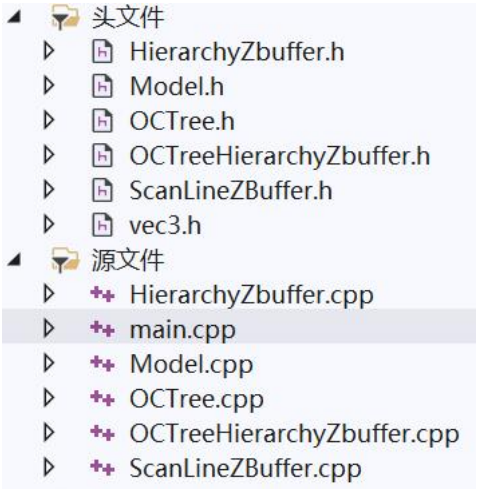
一、参考说明

（1）vec3.h 的实现参考了《RayTracing in One Weekend》系列，https://raytracing.github.io/
（2）ScanLineZBuffer 实现思路参考了该项目，https://github.com/sccbhxc/ScanLineZBuffer
（3）层次 z-buffer 实现思路参考了该篇博客，https://zhuanlan.zhihu.com/p/586628315
（4）八叉树的实现参考了该项目，https://github.com/wyzwzz/HierarchyZBuffer 中的 OCTree 部分。

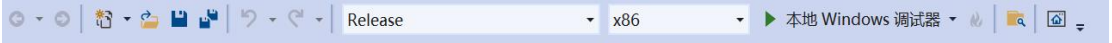
二、文件与使用说明

（1）本项目共包含以下文件，其中：

- main.cpp 中包含了使用 glut 的渲染主框架，支持窗口缩放、鼠标点击旋转、按 c 键切换不同的 buffer 模式的功能
- Model.h 和 Model.cpp 中包含了模型加载及预处理的各种功能
- ScanLineZBuffer.h 和 ScanLineZBuffer.cpp 中实现了扫描线 zbuffer 的功能
- HierarchyZbuffer.h 和 HierarchyZbuffer.cpp 中实现了层次 zbuffer 的简单模式
- OCTreeHierarchyZbuffer.h 和 OCTreeHierarchyZbuffer.cpp 中实现了层次 zbuffer 完整模式
- OCTree.h 和 OCTree.cpp 中实现了简单对于三角形面片的八叉树

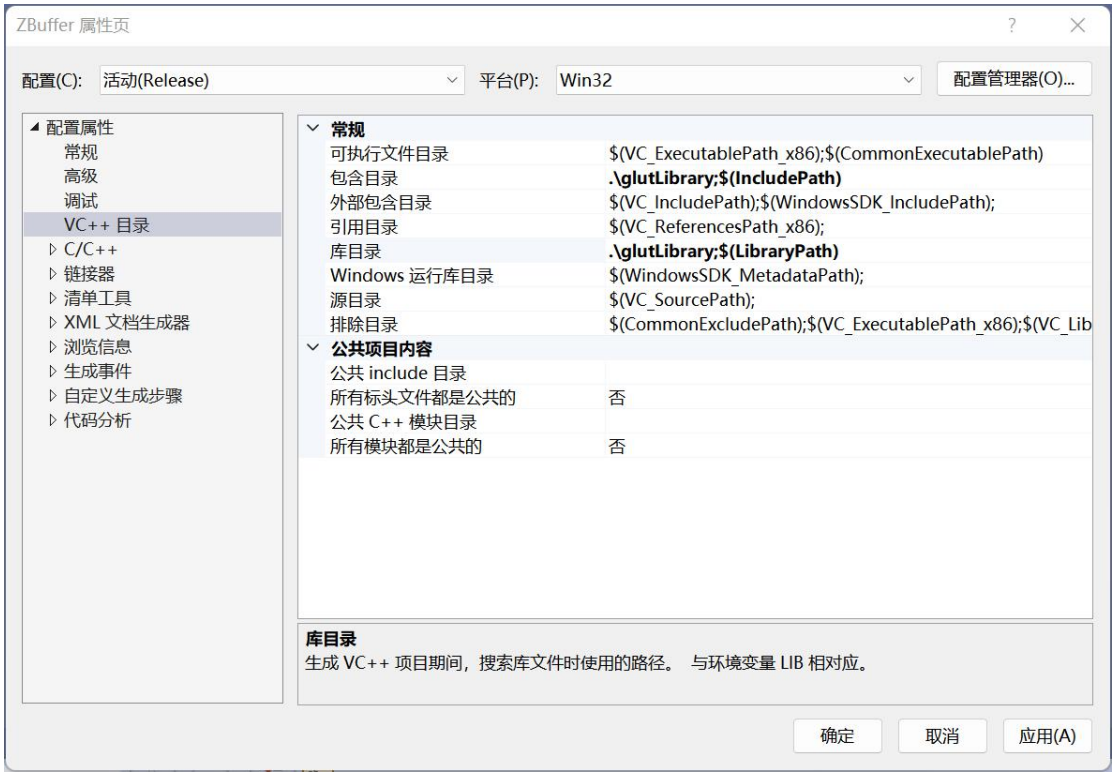


（2）当需要运行 vs 程序时：
①首先选择 x86 环境(因为使用的库是 32 位的)：



②点击项目-属性，设置平台为 32 位

③修改包含目录和库目录，包含文件底下的 glutLibrary

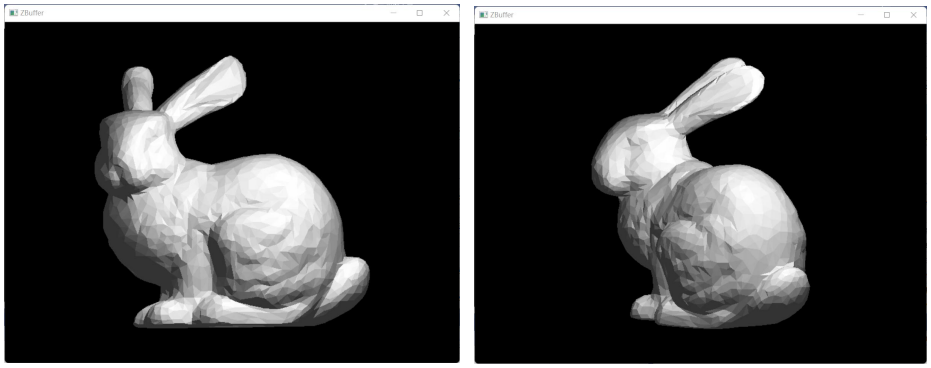


④点击运行后即可运行程序，在 main.cpp 的开头可以修改需要加载的模型。

```
std::string modelPath("./model/bunny.obj");
```

⑤运行程序后，可以通过以下方式进行控制

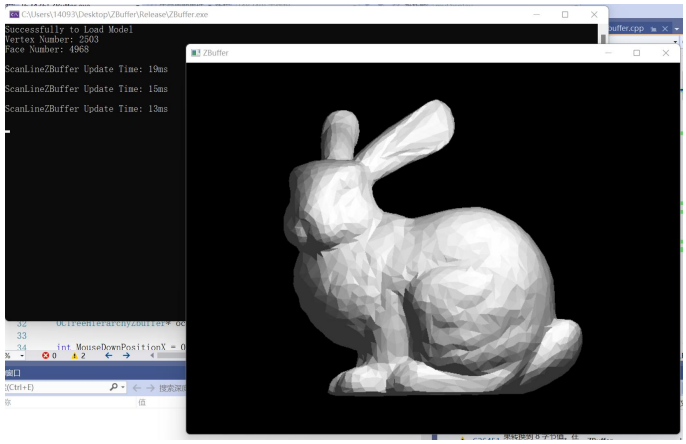
1. 点击并拖动鼠标，在松开时根据鼠标滑动的水平距离和垂直距离旋转模型

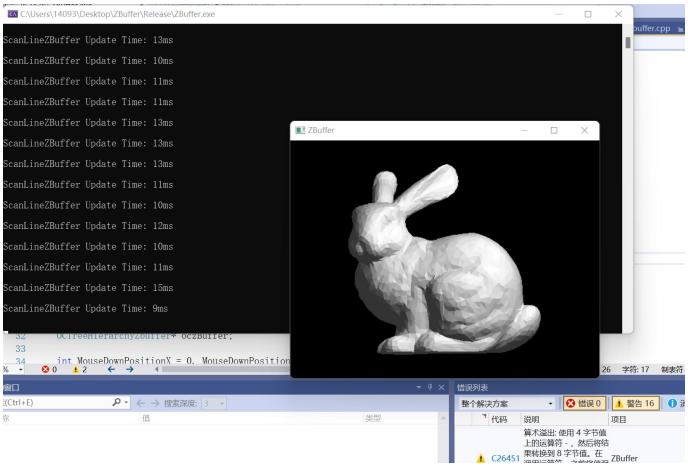


2. 按下 c 键可以切换不同的 buffer 模式，一共有三种模式可供选择

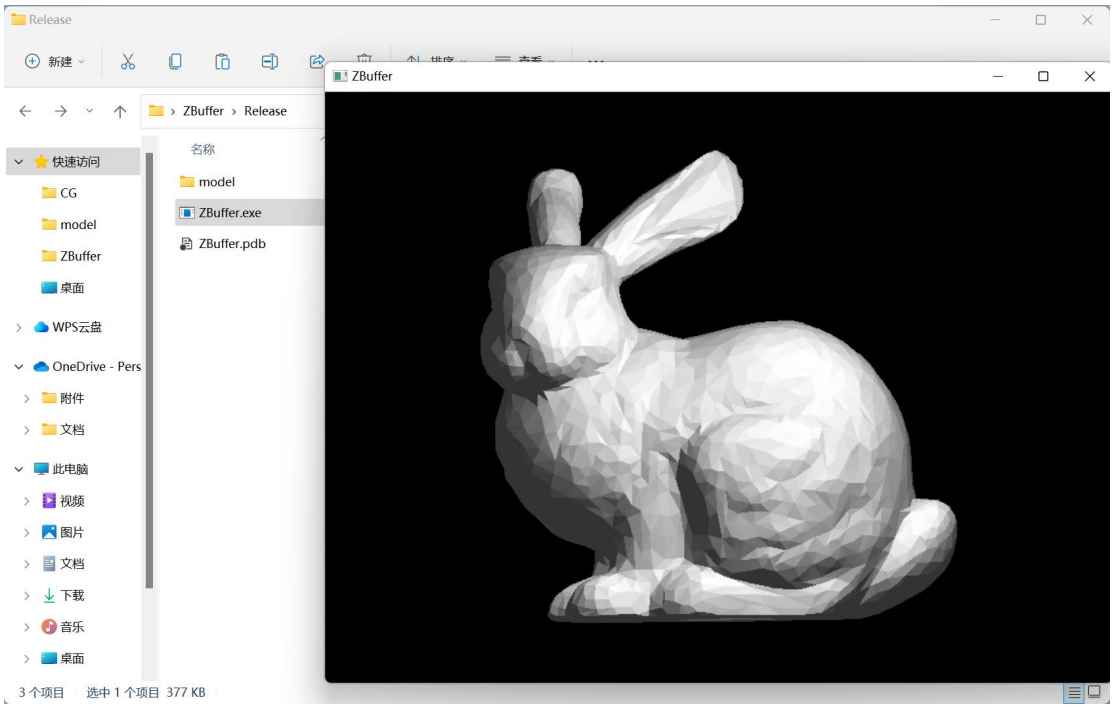
```
Change Mode To ScanLineZbuffer

ScanLineZBuffer Update Time: 16ms

Change Mode To HierarchyZbuffer

HierarchyZBuffer Update Time: 11ms

Change Mode To OCTreeZbuffer

OCTreeZBuffer Build Tree Time: 5ms
OCTreeZBuffer Update Time: 15ms
```

3. 拉动窗口可以改变分辨率的大小：





（3）如果需要直接运行运行 exe 文件，那么可以将模型放入 release 目录底下的 model 文件夹中并命名为 model.obj 即可迅速运行。



三、ScanLineZBuffer 原理及实现

（1）对于扫描线 zbuffer，最关键的是定义所需的存储数据结构，包括如下的分类多边形表、分类边表、活化多边形表、活化的边表。以下是在 ScanLineZBuffer 中定义的存储结构和存储变量：

```
//分类边表结构单元
struct Edge
{
    double x;
    double dx;
    int dy;
    int id;
};
```

```
//活化的边表结构单元
struct ActiveEdge
{
    double x;
    double dx;
    int dy;
    double z;
    double dzx;
    double dzy;
    int id;
};
```

```
//分类多边形表结构单元
struct Polygon
{
    double a, b, c, d;
    int id;
    int dy;
    std::vector<ActiveEdge> ActiveEdgeTable;
};
```

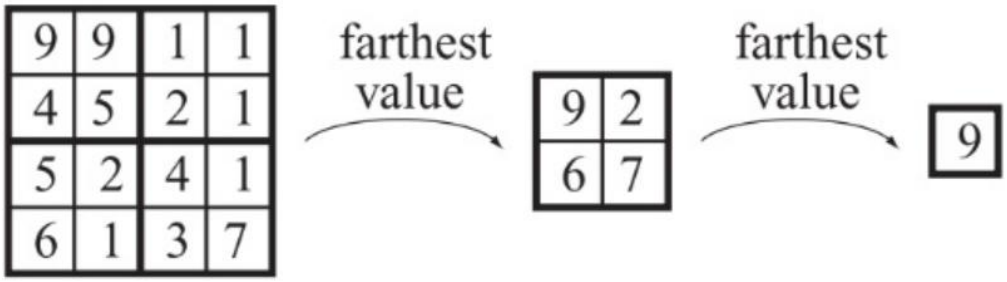
```
std::vector<std::vector<Polygon>> PolygonTable;
std::vector<std::vector<Edge>> EdgeTable;
std::vector<Polygon> ActivePolygonTable;
```

（2）具体更新时依次从上到下扫描，对应实现的伪代码如下。对于每个 y 值首先更新活化多边形表，然后对每个活化多边形判断是否有新的活化边。按照活化边对处理两者区间内的值如果大于已经存储的深度值则进行更新。每次判断完一行更新所有的活化结构，并删掉那些 dy=0 即已经结束了的活化多边形和活化边。

```
//从上到下进行扫描
for(int y=Height-1; y>=0; y--){
    //如果有新的多边形则加入活化的多边形表
    for (auto& i : PolygonTable[y]) {
        ActivePolygonTable.push_back(i);
    }

    //遍历每个活化多边形
    for (int i = 0, plsize = ActivePolygonTable.size(); i < plsize; i++){
        //把对应的新的活化边加入活化边表
        Polygon& selectPolyon = ActivePolygonTable[i];
        selectPolyon.renewActiveEdgeTable();

        //将这些边排序(按照x小的，以及在x相同时dx更小的)
        selectPolygon.sortActiveEdgeTable();

        //成对地处理这些边对，在这些边对的区间中则是可能的填充范围
        for(int j=0; j<selectPolygon.size(); j+=2){
            firstPoint = selectPolyon[j];
            secondPoint = selectPolygon[j+1];
            update(firstPoint, secondPoint);
        }

        //如果dy=0去掉已经结束的活化边
        selectPolyon.remove(0);
    }
    //如果dy=0去掉已经结束的活化多边形
    ActivePolygonTable.remove(0);
}
```

四、层次 z-buffer 简单模式原理及实现

（1）层次 z-buffer 通过将每一帧的深度信息保存为一个图像金字塔(四叉树)，用于让后续的测试能够使用之前的深度信息迅速进行遮蔽判断。如下图所示层次 zbuffer 分为多层，第 0 层即为最高分辨率级别，也就是标准的 z-buffer。然后依次向上，每四个格子合并变成一个新的格子，取其中的深度最大值(实际实现的时候由于 z 轴是朝向屏幕外的，所以我们保存的应该是深度的最小值)。反复进行此合并操作直到某一个维度达到 1 为止就完成了我们的构建过程。



（2）我们依次查看每个面片，将面片的包围盒(AABB 包围盒)投影到屏幕空间，根据如下公式，用最长的投影边估计所在的层次，这里 n-1 代表最高的层数。通过这种方法能够保证在层次 z-buffer 的对应层中最多只覆盖 2*2 个像素信息。如果面片的深度最小值和最多的 2*2 区域对比后的值都要大，则面片被遮蔽，否则说明不被遮蔽需要进行后续的渲染操作。同时完成渲染后需要按照(1)中的步骤对整个层次 z-buffer 进行更新。

$$\lambda = \min\left(\left\lceil \log_2\left(\max(l,1)\right)\right\rceil, n-1\right)$$

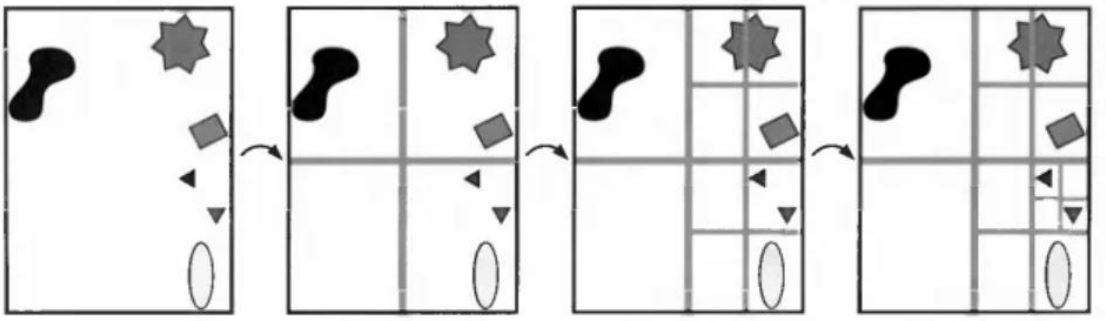
（3）根据先验可以得知，如果上一帧中被遮蔽的物体在这一帧中很有可能也会被遮蔽。所以我们可以先渲染那些上一帧中可以被看见的物体，然后渲染那些不可见的物体，这样可以很好地进行加速。

（4）在 HierarchyZbuffer.cpp 中我们使用 updateAreaDepth 函数从指定的第 0 层范围开始进行更新。并在 RenewAllBuffer 中首先对每个三角形进行层次判断如果被遮蔽则不进行渲染，如果没有遮蔽则对三角形进行扫描记录，对应的伪代码如下所示：

```
void HierarchyZbuffer::RenewAllBuffer(Model* m){
    for(auto &i : allTriangle){
        //使用公式估计出层数
        int level = EstimateLevel(i);
        //检查是否被遮挡
        bool isOclussion = checkOclussion(level, i);
        //如果没有被遮挡则进行绘制以及更新
        if(!isOclussion){
            drawTriangle();
            updateAreaDepth();
        }
    }
}
```

五、层次 z-buffer 完整模式原理及实现

（1）完整模式使用八叉树获取场景的信息，首先将场景模型保存在一棵八叉树当中，场景中遮挡区域的层次剔除由此八叉树支持。我们以 front-to-back 的方式去遍历这棵八叉树，取八叉树的最近面的 z 值去和层次 z-buffer 中的对应层进行比较，如果深度小于某个格点的深度值则说明存在未被遮挡的部分，于是继续往下遍历八叉树。重复这个过程，直至达到八叉树的叶子节点(说明需要绘制这个叶子节点中的多边形)或者是在某个层次的时候八叉树节点被遮挡(说明不需要继续探索其下属节点)。
通过这种从前向后的遍历方式，可以更早地绘制出那些不被遮挡的多边形，而那些会被遮挡的多边形则会在之后的判断中尽早地被剔除，从而达到加速的效果。



（2）对应的实现伪代码如下所示，八叉树的节点存储时应该先存储靠近的也就是深度更小的那些节点再存储深度更大的那些节点，方便遍历时按照 front-to-back 的方式进行遍历。

```
void OCTreeHierarchyZbuffer::RenewAllBuffer(Model* m, OCTNode* node) {
    if (node == NULL) return;
    //层次估计
    int levelEst = ceil(log2(max(maxL, 1)));
    //判断该体有无被遮挡
    bool judge = judgeCulling(levelEst, max_point.y);
    if(judge) return;
    //如果没有被遮挡且存在三角形则对其进行绘制
    for (int i = 0, tsize = node->allTri.size(); i < tsize; i++) {
        drawOneTriangle(node->allTri[i]->id, m);
    }
    //递归下属节点
    for (int i = 0; i < 8; i++) {
        RenewAllBuffer(m, node->child[i]);
    }
}
```

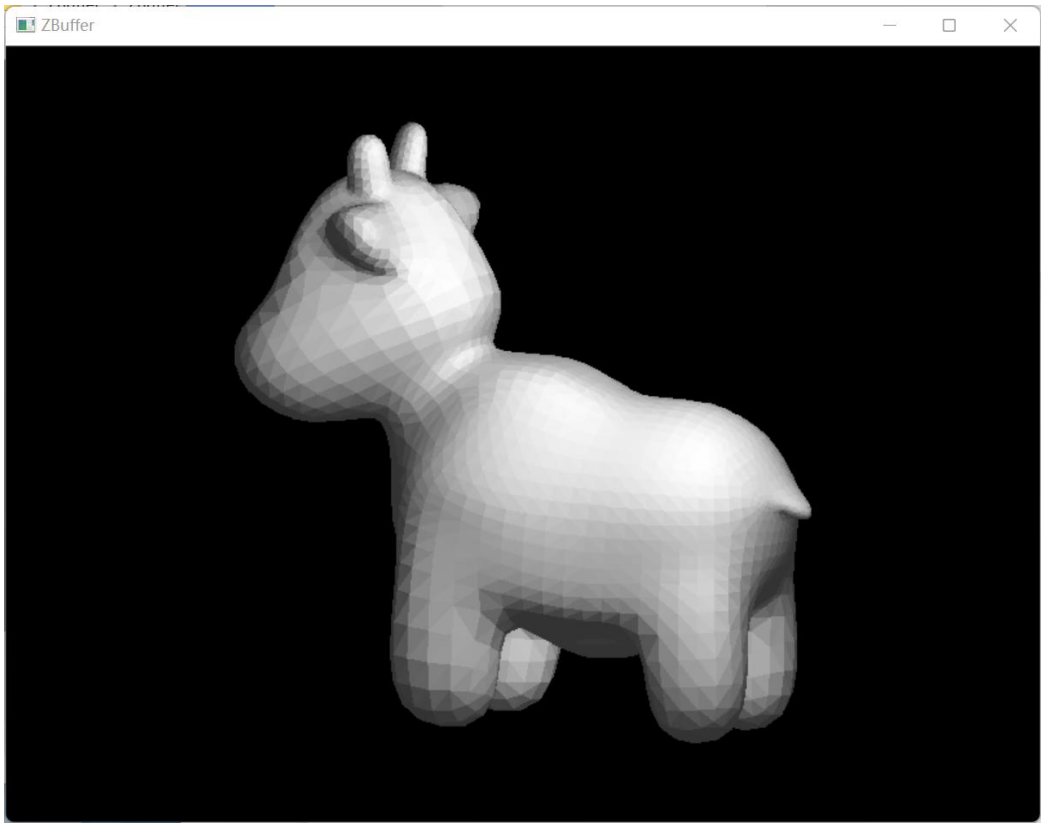
（3）而对应八叉树的主要数据结构的关键部分如下所示

```
class OCTNode{
    //表示这个节点的边界信息
    Bound b;
    //表示这个节点的所有孩子
    OCTNode* child[8];
    //存储的数据
    vector<Triangle*> allTri;
}
```

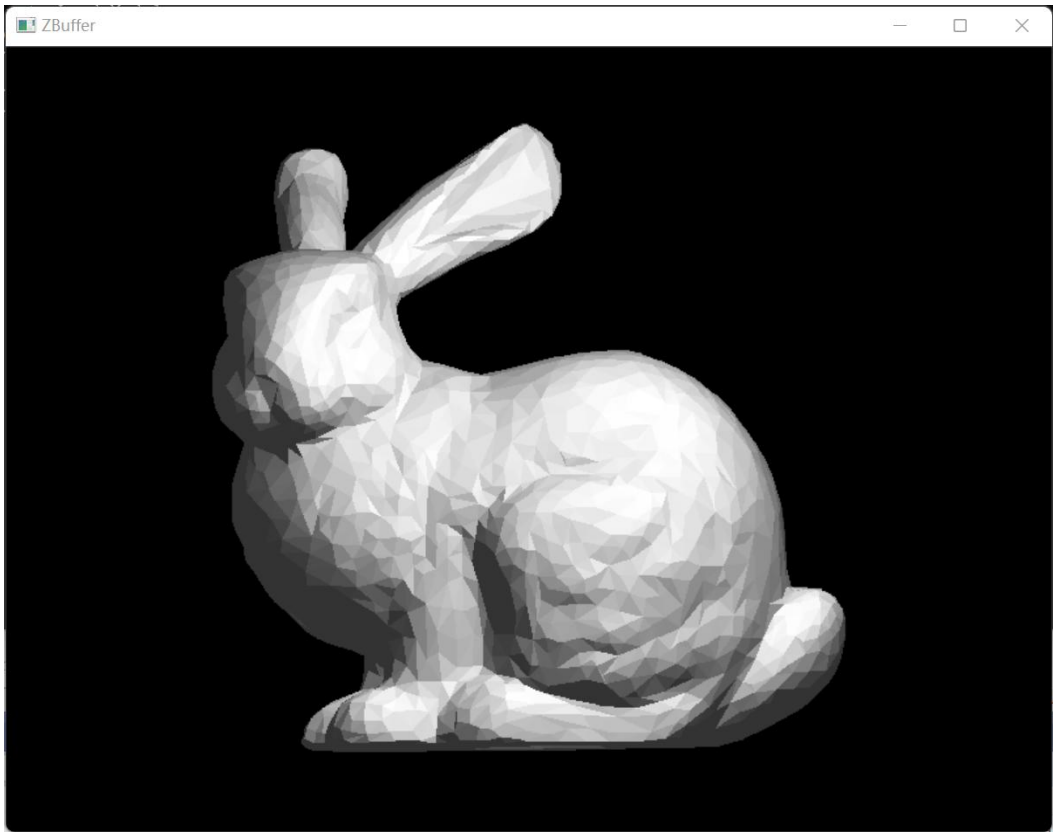
```
class OCTree{
    //构建整棵树,用于插入节点
    void buildTree();
    //最大的节点数量
    int maxTriNum;
    //根节点
    OCTNode* root;
}
```

六、层次 z-buffer 完整模式结果展示

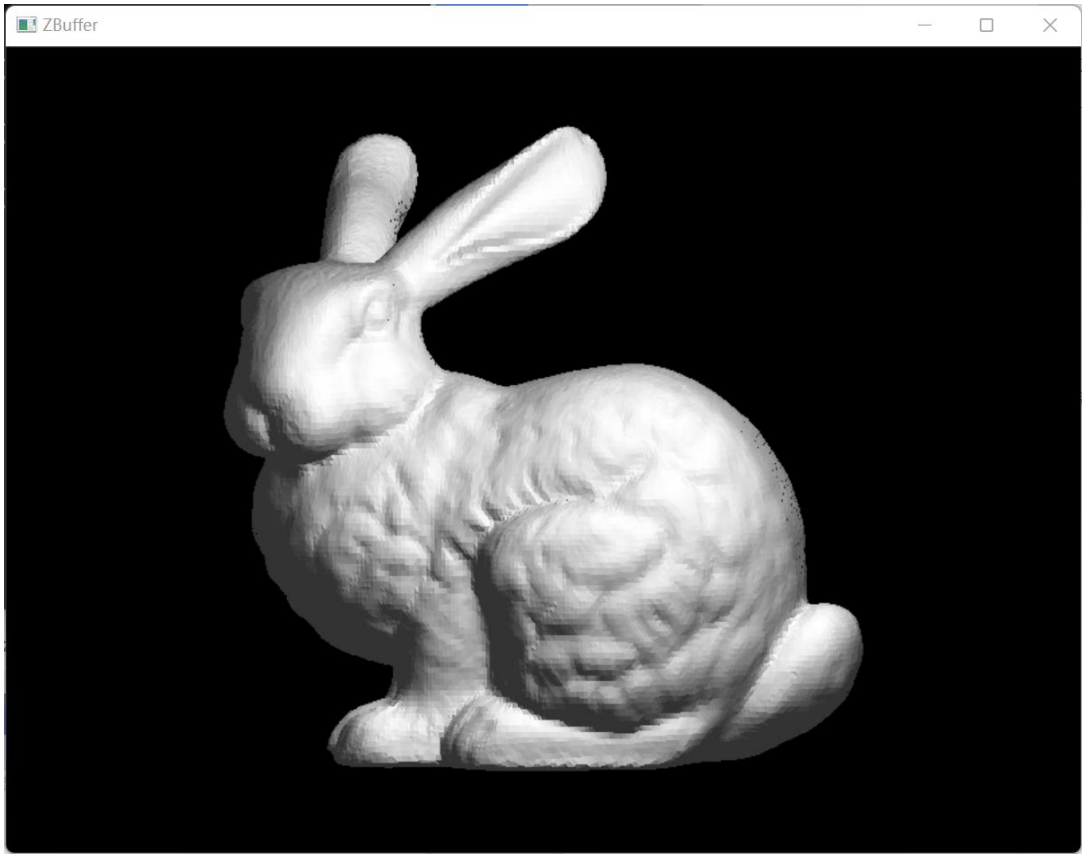
（1）小奶牛 spot.obj，顶点数 3225，面数 5856



（2）简单的兔子 bunny.obj，顶点数 2503，面数 4968



（3）复杂的兔子 bunnyall.obj，顶点数 34834，面片数 69451



（4）茶壶 teapot.obj，顶点数 530，面片数 992

七、对比分析

实验在 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 的台式电脑上进行，结果如下所示。

	茶壶	小奶牛	兔子(简单版)	兔子(复杂版)
顶点数量	530	3225	2503	34834
面片数量	992	5856	4968	69451
扫描线 zbuffer	4ms	13ms	13ms	103ms
简单层次 zbuffer	7ms	12ms	12ms	55ms
简单层次 zbuffer (记录并先渲染上一帧的信息)	6ms	11ms	10ms	50ms
完整层次 zbuffer（单个节点最多包含 20 个面片）	2ms/6ms	6ms/10ms	7ms/10ms	100ms/48ms
完整层次 zbuffer（单个节点最多包含 50 个面片）	1ms/6ms	4ms/10ms	5ms/10ms	76ms/50ms
完整层次 zbuffer（单节点最多包含 100 个面片）	0.5ms/7ms	3ms/10ms	3ms/10ms	56ms/52ms

（1）简单层次 zbuffer 使用了两种模式，一种是每次都按照读入 obj 中面片的顺序来渲染面片，另一种则是利用了上一帧的信息，先渲染上一帧显示出来的三角形。根据时间上的关联性，上一帧中显示出来的面片在这一帧中也很有可能显示出来，所以先渲染这些面片有利于直接排除潜在的被遮挡的面片。

（2）对于完整的层次 zbuffer，由于八叉树单个节点的大小决定了空间剖分的精细程度，因此这里对比了单个节点最多包含 20、50、100 个面片时的情况。每个数据包含两个数字前一个代表构建八叉树的时间，后一个代表渲染所需要的时间。可以看到当一个节点最多包含面片数越小时构建八叉树的时间越短，而由于剖分的不精细不能够有效地使用遮挡信息所以渲染所需要的时间就会更多。两者需要一个折中处理。

（3）综合比较，当面片数量很少时，由于层次 zbuffer 需要建立额外的数据结构，其所需的时间是大于扫描线 zbuffer 的，但是当面片数量越来越多的时候，扫描线 zbuffer 所需的时间开始逐渐变大。我们可以看到利用了前一帧渲染信息的简单层次 zbuffer 是最高效的，因为它更好地运用了“连贯性”原理，相对来说完整模式的 zbuffer 还需要建立八叉树，他能够更快地得出哪些面片是位于前方的。

（4）可以看到建立 octree 所需的时间非常长，这是本程序的一个不足之处，可以对八叉树的建立过程进行进一步的优化以提高效率。

（5）综合来看，如果不考虑建立八叉树的过程只考虑渲染面片的过程那么在复杂情况下：

加速比简单模式(不使用前一帧信息) = 1.87

加速比简单模式(使用前一帧信息) = 2.06

加速比完整模式 = 2.146

（6）如果进一步提高面片的数量复杂场景表示，那么使用了空间关系的层次 zbuffer 方法应该能够获得更大的加速比例。