实现不同的Z-buffer算法

实现不同的Z-buffer算法
- 一.实验框架
- 二.各个头文件中的重要函数功能解释
- 三.扫描线ZBuffer算法的实现细节
  1. 明确数据结构：
  2. 算法步骤
  3. 算法细节
  4. 扫描线ZBuffer算法的公式推导
- 四. 层次Z-Buffer算法
- 五. 八叉树加速的层次Z-Buffer
- 六. 实验结果展示及分析
- 七. Appendix

一.实验框架

使用的第三方库：

- GLM库，是OpenGL的线性代数库，用于进行向量、矩阵相关运算。
- SDL2库，窗口界面库，主要用于创建窗口给并显示渲染的图片结果。
- TinyObjLoader，模型数据加载库，用于加载obj模型。

工程代码框架目录结构：

```
├─build ----------------------> 二进制文件目录，存放项目工程文件、中间文件、编译结果
| ├─Debug -------------------> vs在debug模式下的编译结果存放目录
| ├─model -------------------> 模型数据文件目录，存放待渲染的模型文件
| └─Release -----------------> vs在release模式下的编译结果存放目录
├─include --------------------> 项目依赖的第三方库头文件目录
| ├─glm ---------------------> 数学库头文件目录
| └─SDL2 --------------------> 窗口库头文件目录
├─libs -----------------------> 项目依赖的动态链接库文件目录
└─src ------------------------> 项目的源代码文件目录（存放项目的.h文件和.cpp文件）
```

本项目代码的程序入口为src/main.cpp 。

工程构建方式：见Appendix或(https://github.com/lyh1028/Different-ZBuffer)

本实验的工作：

- 构建了一个完整的渲染管线流程用于显示3维Obj模型，包括Vertex Porcessing， Triangle Processing， Rasterization， Fragment Processing， FramBuffer Operations五个阶段。
- 实现了一个简易窗口界面，可以用鼠标滚轮缩放、左右拖拽鼠标旋转画面。
- 采用了背面剔除和顶点剔除的加速。
- 使用双缓冲避免掉帧。

- 分别实现了ZBuffer，扫描线ZBuffer， 层次ZBuffer，八叉树加速的层次ZBuffer算法用于深度测试。
- 实现了朗伯特反射模型，并加入了环境光。

二.各个头文件中的重要函数功能解释

我们配合数据流来说明每个文件的功能：

首先需要读取Obj文件，这里用到了DrawableMesh类

DrawableMesh.h

- 主要用于从Obj文件中读取顶点和面片信息。由于我们没有实现纹理功能，而大部分obj文件里没有指定颜色，所以只需要读取v, vn, f即可(tinyobjloader会自动把颜色读取为默认值(1.0f,1.0f,1.0f))
- VertexData类：
  - 包含一个顶点的世界坐标、NDC空间坐标、屏幕坐标，以及这个顶点的颜色和法线。
- MeshFace类：
  - 包含一个顶点的模型坐标和法线坐标对应的下标索引
- 给定一个obj文件路径，读取对应数据的函数为`void DrawableMesh::loadMeshFromObjFile(const std::string& filename)`

读取文件后，就开始绘制工作，绘制的主要方法声明在`RenderPipeline.h`内。

如果你选择最普通的ZBuffer，每个面片的顶点会通过模型变换-->视图变换-->透视变换-->视口变换得到顶点对应的屏幕坐标，这一过程是通过vertexShader函数实现的，而各个变换的矩阵是通过`calcViewMatrix`等函数计算得到的。此外，我还设置了**透视投影前后的透视矫正**函数，分别为`void prePerspCorrection(VertexData& v);`和`void aftPrespCorrection(VertexData& v);`， 在计算出顶点的屏幕坐标后，利用bounding box确认面片的顶点坐标范围，之后依次遍历bounding box内的顶点，判断其是否在三角形内部，如果在三角型内部，就用重心插值计算其法线、颜色，完成这一光栅化步骤的函数如下：

```
static void Pipeline::rasterize_fill_edge_function(
        const VertexData& v0,
        const VertexData& v1,
        const VertexData& v2,
        const unsigned int& screen_width,
        const unsigned int& screen_height,
        std::vector<VertexData>& rasterized_points);
```

最后，深度缓冲和帧缓冲相关函数分别保存在`VanillaZbuffer.h`，`FrameBuffer.h`文件中。

FrameBuffer.h

- frame_buffer, 维护每个像素点的颜色
- 屏幕窗口的长宽

这里我们使用了**双缓冲**技术（其实也可以不用，因为我们靠划鼠标控制摄像机移动，速度比较慢），避免显示时的闪烁现象。

所有的深度缓冲类都继承于Zbuffer.h， 它们一定包含下列变量：

- pipeline类的指针，方便调用相关函数。
- p_frontBuffer，指向当前显示的帧缓存。

- p_backBuffer，指向当前正在写入的帧缓存。
- Z-buffer，维护每个像素的深度。

如果使用ScanlineZBuffer， 就在顶点进行坐标变换之后构建分类边表和分类多边形表，随后使用扫描线逐行扫描，更新屏幕上每一点的像素值。

Polygen.h:

- Edge类，表示多边形投影到屏幕上的边
- Polygen类，表示NDC空间中的多边形面片
- ActiveEdgePair类，表示活化边表(每一个活化边表的元素都是一个边对)
- `std::pair<int,int> makeEdges(const std::vector<VertexData>& vertices, std::unordered_map<int, std::list<Edge> >& edge_table, int id)` 给定一个多边形的顶点，构建这个多边形包含的所有边，加入到分类边表中。

如果使用Hirerach ZBuffer（对应文件`HierarchZbuffer.h`）， 需要利用`QuadTree.h`里的四叉树数据结构构建Z金字塔，

如果使用八叉树加速层次ZBuffer，则需要引入`HzbPolygen.h`和`Octree.h`，前者构造了一个包含AABB包围盒的多边形（与Polygen.h不同，只有顶点和包围盒），后者实现了八叉树数据结构。

三.扫描线ZBuffer算法的实现细节

1. 明确数据结构：

- 分类边表：要保存边的屏幕上的y_{max}坐标（用于索引），边上端点的屏幕x坐标，相邻两条扫描线经过的x距离dx（设边方程可以写为y=kx+b)，$dx=-\frac{1}{k}$， 边跨越的扫描线数目dy, 边所属多边形的编号id
- 分类多边形表：要保存多边形在NDC空间是一个三维平面，其方程为ax+by+cz+d=0，要保存a,b,c,d；还要保存多边形的编号，跨越扫描线的数目dy, 以及多边形在屏幕上的y_{max}坐标用于索引。
- 然后通过查看算法步骤，发现分类边表和分类多边形表在建表之后需要通过y值频繁查询，并且内部需要增删边和多边形，所以使用**哈希表嵌套双向链表的数据结构**， 例如`std::unordered_map<int, std::list<Edge> >`
- 活化边表：通过分析算法步骤，发现活化多边形表不需要频繁增删元素，直接使用vector
- 活化多边形表：通过分析算法步骤，发现在三角形面片条件下，这个算法压根就**不需要活化多边形表!**

2. 算法步骤

1. 在扫描每个面片，对其进行Vertex Processing之后，构建对应的多边形和边，加入分类多边形表和分类边表中。
2. 扫描线从下到上扫描(Y值从大到小)，对每一条扫描线，有add->draw->update->delete四步骤
    1. add过程：检查分类的多边形表，如果有新的多边形涉及该扫描线，则把该多边形在 xOy 平面上的投影和扫描线相交的边加入到活化边表中
    2. draw过程：从左到右扫描x，更新这一行的每一个像素值和深度值。

3. update过程：更新活化边表中边的dyl, dyr, xl, xr, zl, 如果dyl或者dyr小于0，则从分类边表中找到id相同的边，用这条边替换。（如果都小于0，无需处理，等后面删除）
4. delete过程：在一条扫描线切换到下一条扫描线之前，把活化边表中所有dy小于0的边删除。

3. 算法细节

这个算法很难写，**写出来效果也不好**，原因是每个**面片的颜色是固定的，而不能按顶点着色。**

1. 多边形要增加一个保存多边形在NDC空间中的顶点的变量，边要增加一个保存该边上端点在NDC空间中对应顶点的变量。这样是为了后面计算dzx，dzy， 并且在平面平行于z轴时也能计算其深度
2. 关于极值点的像素截断：
    1. ppt中说根据y_max（上端点 y 坐标 ）把边放入相应的类中(非极值点：截断 1 Pixel)， 这里截断的时候一定要小心！（具体看代码，需要分情况讨论）
    2. 但是**可以不截断**， 只需要更改一下update，先判断dyl或者dyr是否等于0，等于0则替换边，然后再更新活化边的dyl, dyr, xl, xr, zl.
3. 关于活化边的dzx和dzy，在软光栅、OpenGL坐标系的框架下，**不能使用ppt的公式。**（详情见下一部分的推导过程）
4. 特殊情况处理：**如果存在平面Ax+By+Cz+D=0(C=0)的情况**，那dzx, dzy可以直接设为0，zl是对应上端点的zl.
5. 关于坐标的细节：我们使用的OpenGL的坐标系，世界空间是右手系，NDC空间是左手系，z越小，离屏幕越近。屏幕空间左上角为坐标原点，向右和向下为x轴、y轴的正方向。

4. 扫描线ZBuffer算法的公式推导

我求的多边形方程是**NDC空间的平面方程**，而NDC到屏幕还有一个视口变换矩阵M，**ppt里认为dzx=-A/C，是直接利用了多边形方程Ax + By + Cz+D=0，简单地把x换成x+1.** 我们这里屏幕坐标x_l变成x_l+1时， 虽然NDC坐标的y肯定不会变，但x_{NDC} 并不变成$x_{NDC}+1$

具体推导过程：假设窗口高h，宽w，则视口变换矩阵M如下：（由于OpenGL和屏幕坐标系y轴方向不一样，所以第二行第二列的元素是-h/2）

$$M=\begin{bmatrix}\frac{w}{2} & 0 & 0 & \frac{w}{2}\\ 0 & \frac{-h}{2} & 0 & \frac{h}{2}\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

所以，$x_l=\frac{w(x_{NDC}+1)}{2}, \frac{w(x_{NDC}+dx+1)}{2}=x_l+1$， 解得$dx_N=\frac{2}{w}$

所以，多边形在屏幕空间的投影(x_l,y_l)对应NDC空间的坐标是$(x_{NDC},y_{NDC},z_l,1)$，则(x_l+1,y_l) 对应$(x_{NDC}+dx_N,y_{NDC},z_l+dz_x,1)$，满足$A(x_{NDC})+Adx+By_{NDC}+Cz_l+Cdz_x+D=0$

所以在同一条扫描线上，x向右移动一个像素，z的变化是$z_l=z_l+dz_x$, 其中$dz_x=\frac{-2A}{wC}$

同理，当y向上移动一个像素时，$dz_y=\frac{-2B}{hC}$

同理，当屏幕坐标(x_l,y_l)变成(x_l',y_l-1)时(扫描线向上移动)，（这里x_l'对应的是ppt里的x_l+dx），可以求出来

$$\frac{h(1-y_{NDC})}{2}=y_l,\ y_l-1=\frac{h(1-(y_{NDC}+dy_N))}{2},dy_N=\frac{2}{h}$$

$$x_l+dx=\frac{w(x_{NDC}+1)+wdx'_N}{2},dx'_N=\frac{2}{w}dx$$

$$A(x_{NDC}+\frac{2}{w}dx)+B(y_{NDC}+\frac{2}{h})+C(z+dz)+D=0,$$

所以$dz=\frac{-2A}{WC}dx-\frac{2B}{hC}=dz_xdx+dz_y$

四. 层次Z-Buffer算法

层次Z-Buffer算法的原理是构建一个屏幕空间的Z值金字塔。

根据论文，Z金字塔的原理如下：

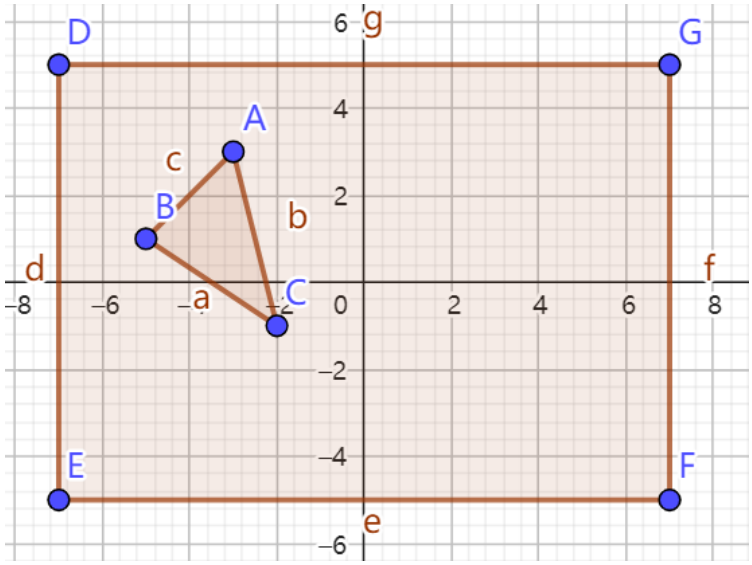
> Z 金字塔的基本思想是使用原始 Z 缓冲区作为金字塔中最精细的级别，然后通过选择距观察者最远的 Z，将每个级别的四个 Z 值合并为下一个较粗级别的一个 Z 值。因此，金字塔中的每个条目都代表 Z 缓冲区的方形区域的最远 Z。在金字塔的最粗层，有一个 Z 值，它是整个图像中距离观察者最远的 Z。

简单总结，就是对原始的Z-Buffer不断使用2*2的max pooling，直到池化为只剩下一个Z值。每一次池化的结果都要保存起来，构成了一个Z金字塔。鉴于每次都是4个"像素"合成1个"像素"，我们使用四叉树维护Z金字塔。

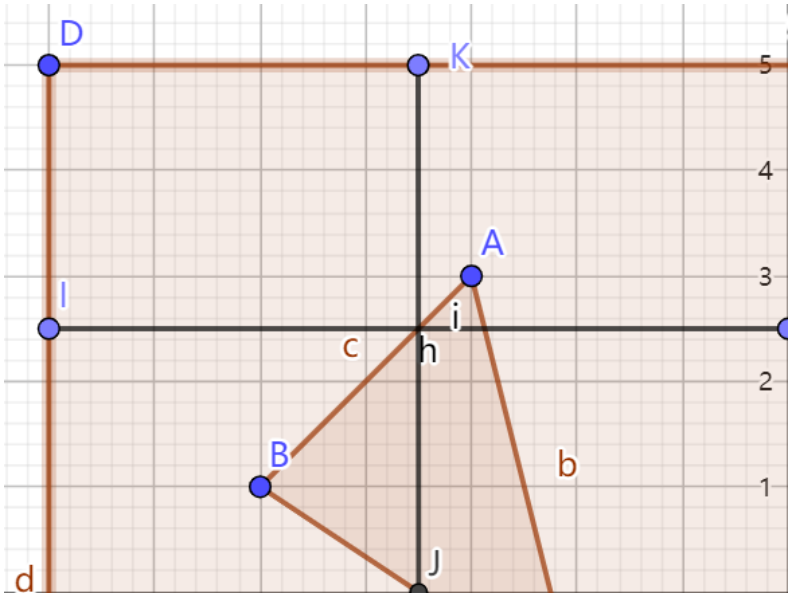
Z金字塔可以加速可见性测试：

- 对于一个面片F，找到能够覆盖F的最细粒度的金字塔块Q
- 如果面片F里离屏幕最近的点的z值还要大于Q维护的最大z值，则面片F不可见。
- 如果无法判断不可见，则进入子节点，分别判断是否可见（递归）

Z金字塔检索可见性的例子：



如图，三角形ABC的最细粒度的包围盒是DEGF，即四叉树的根节点。如果无法判断，下一步进入四个子节点，注意到只有左上和左下的节点包含三角形ABC，所以只看这两个节点。



这两个节点如果无法判断，则进一步划分空间，如此递归。

递归结束的条件：

这里有两种做法，第一种做法是递归时计算多边形在子区域内的深度最小值进行比较，这种递归可以递归到单个像素，进而判断这个像素点是否可见。但这种方式每次都要计算多边形的区域深度最小值，有些困难。

另外一种是论文采用的方法，每次只与整个多边形的最近深度值进行比较，这样只能确定被排除的多边形一定不可见。当无法证明多边形是隐藏的时，将恢复到普通的扫描转换，以确定可见性。

由此可见，我们要维护每个四叉树节点对应的深度最大值，对应为`QuadTreeNode.h`头文件中的`updateZpyramid`函数，当某个像素的深度值更新时，会同时逐级向上更新Z-pyramid

五. 八叉树加速的层次Z-Buffer

如果多边形的任何像素都是隐藏的，我们会说该多边形相对于 Z 缓冲区是隐藏的。类似地，如果立方体的所有面都是隐藏多边形，我们会说立方体相对于 Z 缓冲区是隐藏的。

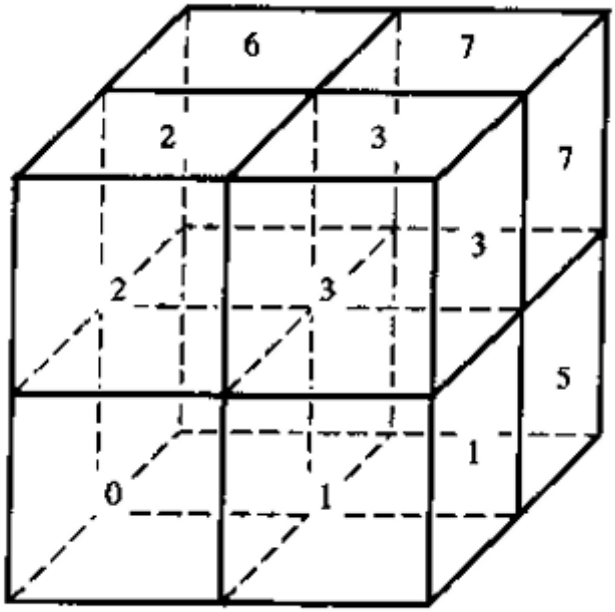
因此，如果一个八叉树节点是隐藏的，那么被这个八叉树节点包含的所有多边形都是隐藏的。这一点可以用来加速深度判断。

先把所有的顶点全部转换到NDC空间，然后在这个空间内创建一个八叉树，把每个面片都和一个cube（八叉树的某一节点）绑定

创建八叉树和创建四叉树的过程相同。只不过多了一个z轴。主要的不同在于递归终止的判定。在这里，**当一个cube只包含n个多边形时，我们不再拆分这个cube。**n的最佳选择（使运行时间最短）和模型包含面片数量有关。如果n太小，可能会因为递归深度过大导致Stack Overflow. 实验中对于包含20k面片的模型，n取10，对于包含5k面片的模型，n取5.

从八叉树的根节点开始使用以下递归步骤：首先，检查当前节点是否在NDC空间内。如果在，我们扫描转换其子节点的面以确定是否有子节点被隐藏。如果子节点未被隐藏，扫描这个子节点立方体内的多边形，然后按从前到后的顺序递归绘制。

对于一些非常小、但处于立方体中心的多边形面片，它们会被认为属于很大的Octree节点，只要整个大节点不隐藏，它们就要花时间绘制。论文中的解决方案是，如果发现一个图元(primitive)与cube的分割平面相交，但与cube相比很小，那么只将其与当前cube相交的cube的所有子立方体相关联。但这样，绘制过程中我们可能会多次遇到它们。第一次遇到它们后，我们将它们标记为已绘制即可。



a)

我们从前向后遍历八叉树的顺序如上图所示，从0-7依次遍历。

六. 实验结果展示及分析

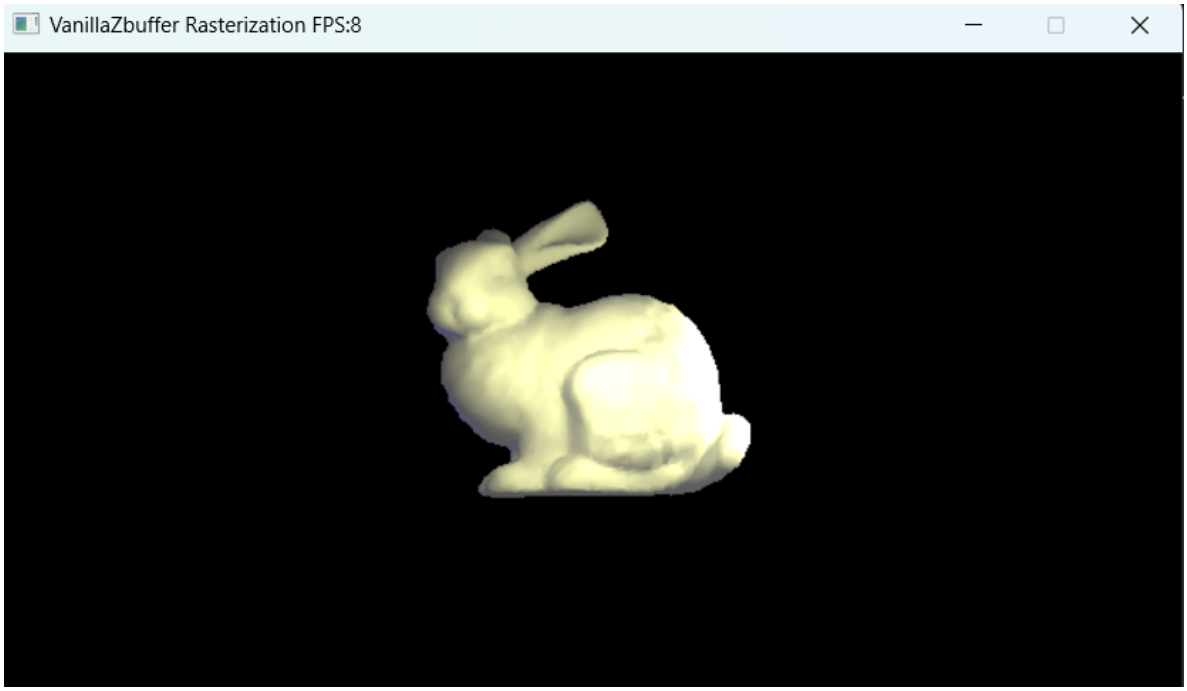
不同面片数模型在不同ZBuffer情况下绘制所用时间（单位：毫秒）：【注意：以下时间为visual studio debug模式下的时间！使用release模式能够直接加速几倍！】

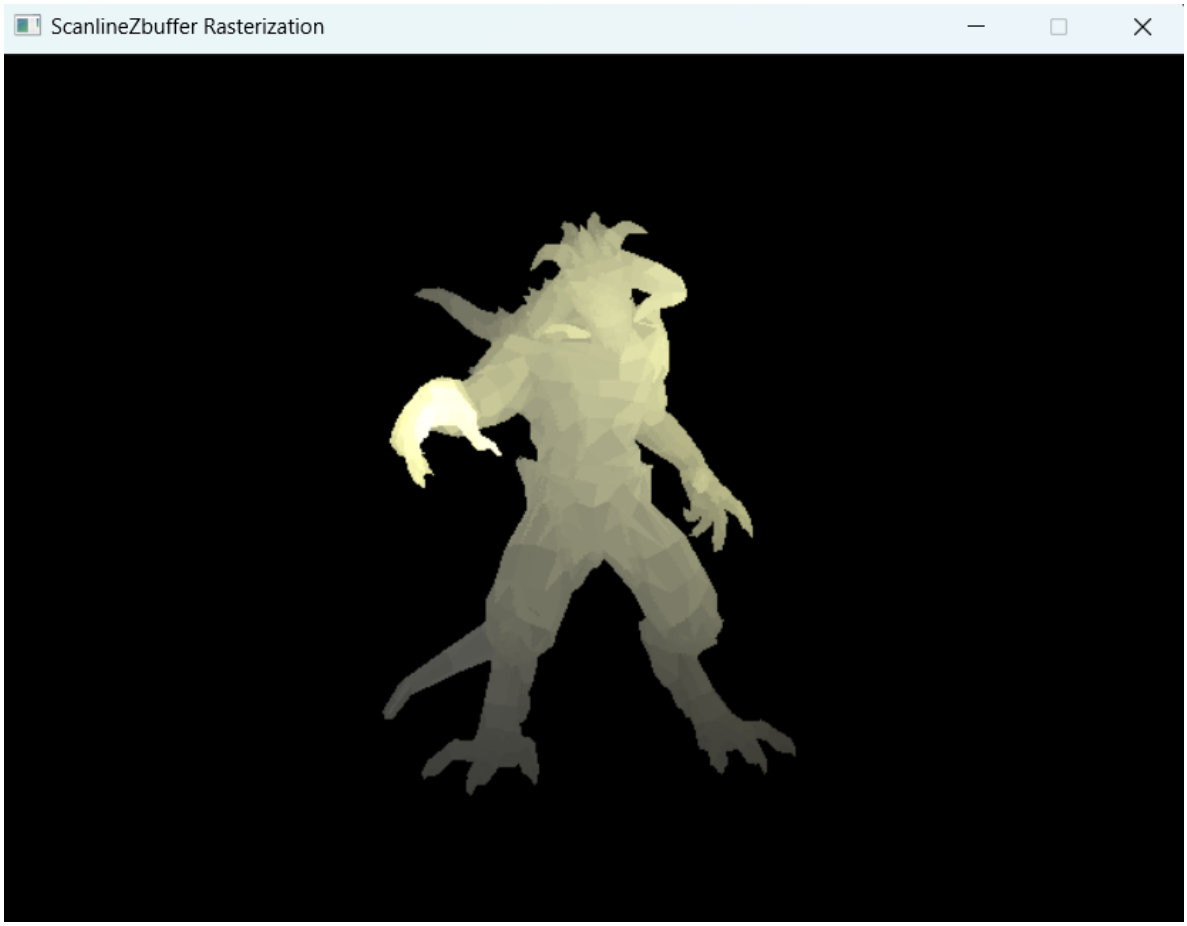
模型名称	面片数	普通ZBuffer	扫描线ZBuffer	层次ZBuffer	八叉树加速层次ZBuffer
cube.obj	12	521	40	3894	3087
monster_5k.obj	5022	117	153	819	373
bunny_10k	10000	173	292	744	343
armadillo_20k	212574	2029	13351	3332	4199

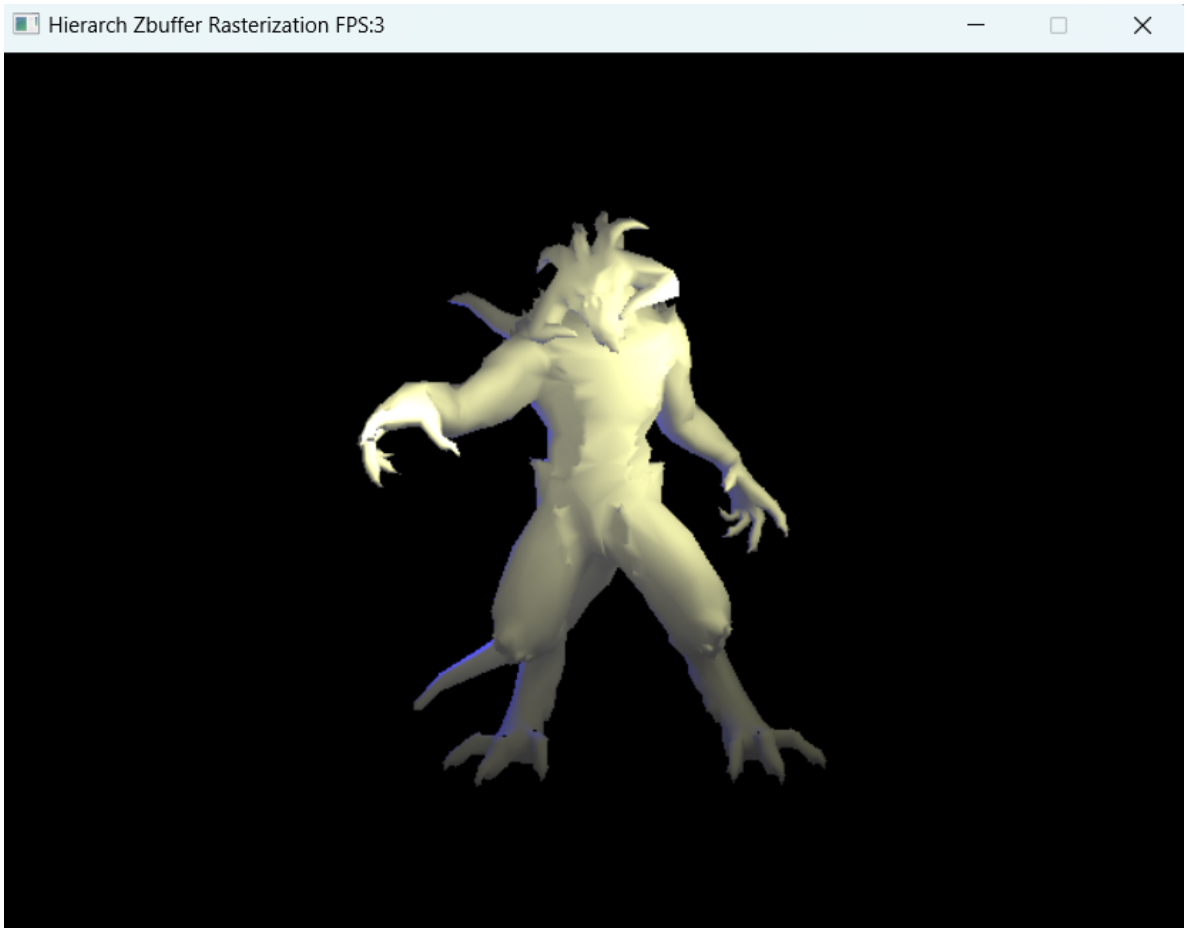
分析：在面片数较少的时候，层次ZBuffer和八叉树加速的层次ZBuffer用时最多，因为它们需要构建Z金字塔和场景八叉树，而这些数据结构在面片少的时候无法起到应有的加速效果。可以看到，随着面片数量增多，层次ZBuffer和八叉树加速的层次ZBuffer绘制模型的时间增长较少，在20k面片时，层次ZBuffer和八叉树加速层次ZBuffer，相对扫描线ZBuffer的**加速比分别为4.01和3.18**

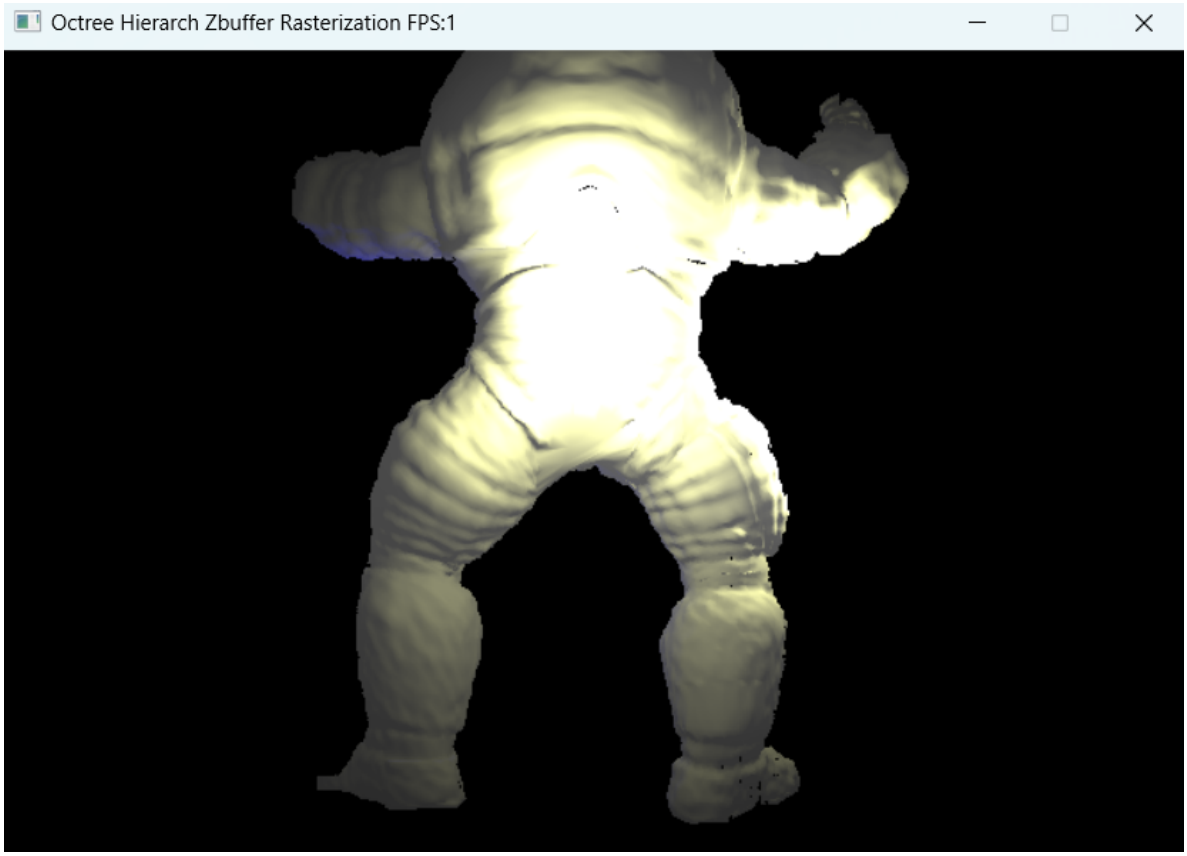
此外，还可以发现，其他的Zbuffer都是“按点着色”的，而扫描线Zbuffer把一个多边形整个设置为一个颜色，在效果表现上不如其他方案。cube.obj中，扫描线ZBuffer用时最少的原因也是因为cube的面片非常大，省去了大量的点光源着色计算。

图片展示：


VanillaZbuffer Rasterization FPS:8


ScanlineZbuffer Rasterization





七. Appendix

所有类的介绍：

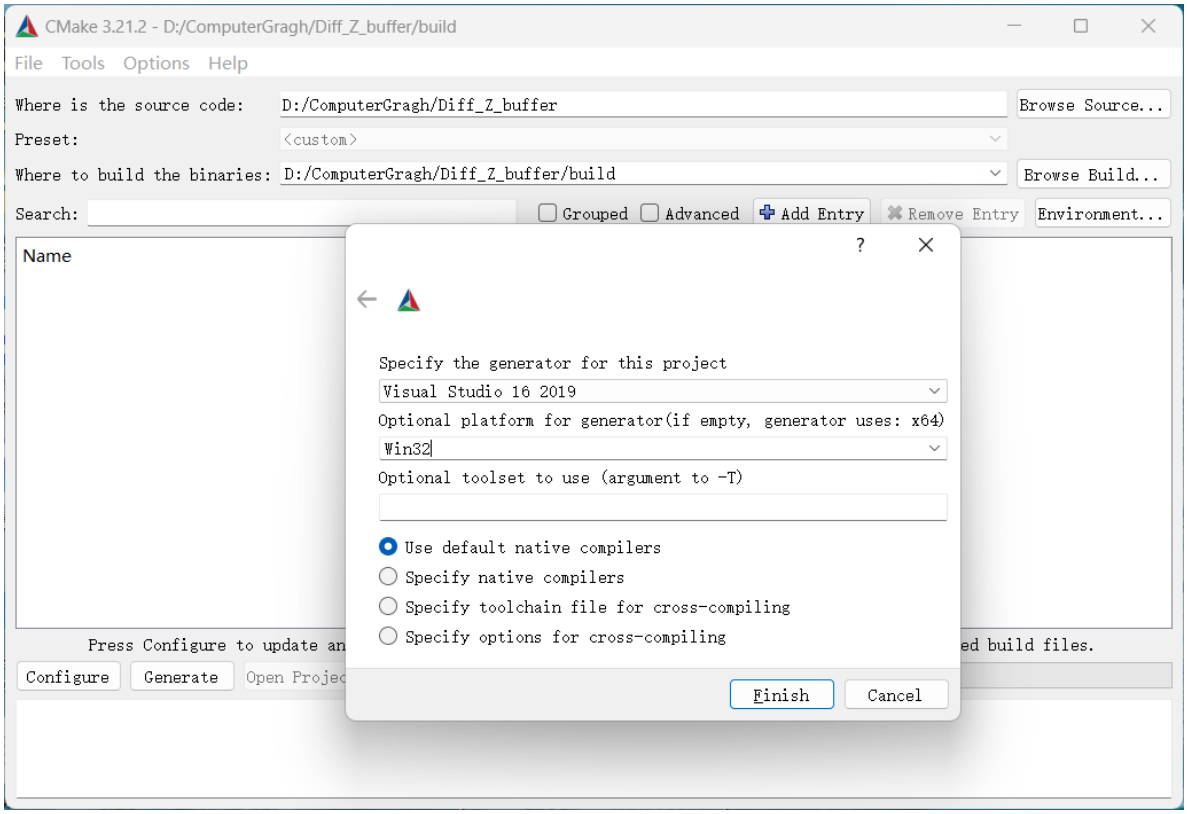
class Zbuffer	Zbuffer的基类
class VanillaZbuffer	普通Z-buffer
class ScanlineZbuffer	扫描线Z-buffer
class HierarchZbuffer	层次Z-buffer
class QuadTreeNode	四叉树
class OctreeNode	八叉树
class OctreeHZBuffer	八叉树加速的层次Z-buffer
class HzbPolygen	OctreeHZBuffer需要用到的多边形类
class Edge	扫描线Z-buffer需要用的边类(构造边表)
class Polygen	扫描线Z-buffer需要用的多边形类(构造多边形表)
class ActiveEdgePair	扫描线Z-buffer需要用的活化边类
class PointLight	点光源
class Pipeline	包含渲染管线的所有函数，包括设置MVP矩阵，顶点和片段着色器
class TRWindowsApp	窗口显示

实验环境配置：

CPU	Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz
RAM	7.88GB
操作系统	Windows 11

构建工程的方式：

使用Cmake-gui，令source code的文件目录为整个工程文件的根目录，build目录为/build，设置中选择Win32，最后把CGAssignment3设置为启动项目。


CMake 3.21.2 - D:/ComputerGragh/Diff_Z_buffer/build
File Tools Options Help
Where is the source code: D:/ComputerGragh/Diff_Z_buffer
Browse Source...
Preset: <custom>
Where to build the binaries: D:/ComputerGragh/Diff_Z_buffer/build
Browse Build...
Search:
Grouped
Advanced
Add Entry
Remove Entry
Environment...
Name
Specify the generator for this project
Visual Studio 16 2019
Optional platform for generator(if empty, generator uses: x64)
Win32
Optional toolset to use (argument to -T)
Use default native compilers
Specify native compilers
Specify toolchain file for cross-compiling
Specify options for cross-compiling
Finish
Cancel
Press Configure to update an
ed build files.
Configure
Generate
Open Projec