区间扫描线z-buffer算法

一、编程环境

操作系统：MacOs Sierra 10.12.6

开发平台：Xcode

二、数据结构与算法

2.1 模型结构：Model.h

本项目中，模型的结构是参考obj文件模式构建的，具体如下。

```
class Model {
public:
    vector<Vertex> vertexList;  //存储模型顶点信息
    vector<Float3> normalList; //存储模型法向量信息
    vector<Face>   faceList;    //存储模型的面片信息
    //……
};
```

如上所示，模型结构还包括顶点结构面结构，它们的结构如下所示：

```
class Vertex {
public:
    Float3 pos;   //坐标
    Vertex();
};

class Face {
public:
    vector<int> vertexIDX; //面片顶点索引
    vector<int> normalIDX;//面片法向量索引
    Float3 color;              //面片颜色
    Float3 normal;             //面片法向量
    Face();                      //对color初始化
};
```

另外还有个结构是Float3，它是自定义的三维float类，操作与float类没有差别，这里不做赘述。

2.2 光栅化结构：Rasterizer.h

该类的目的是将模型成比例缩小并位移到屏幕中心点，其结构如下：

```
class Rasterizer{
private:
    int width, height;                    //显示屏幕的长宽
    float scaleRation;                    //模型相对于屏幕的放缩比例
   //….
};
```

2.3 光照渲染：Render.h

该类的目的是为了赋予模型光照和颜色，使得模型与背景区分开来，其结构如下：

```
class Render{
private:
    const float albedo = 0.8;                                  //光照参数
    const Float3 ambientColor = Float3(0.3, 0.3, 0.3);//环境光
    Float3 lightPos;                                           //点光坐标
    Float3 lightColor;                                         //点光颜色
public:
    //……
};
```

2.4 区间扫描线z-buffer：ScanlineZBuffer.h

该类是本项目的核心结构，实现模型的消隐、zbuffer、framebuffer的计算。

```
class ScanLineZBuffer{
private:
    int width, height;
    float *zBuffer;                                        //保存扫描线的z值
    vector<list<Edge>> edgeTable;                 //边表
    vector<list<Polygon>> polygonTable;                //多边形表
    vector<ActiveEdge> activeEdgeTable;                //活化边表
    vector<ActivePolygon> activePolygonTable;     //活化多边形表
    void Init(Model& model);
    void Release();
public:
    int **frameBuffer;                                     //保存多边形索引
    //……
};
```

其中，边、多边形、活化边以及活化多边形的结构如下：

```
struct Edge{
    float x;   //边上端点的x坐标
    float dxy;  //扫描线向下走过一个像素，点的x增量：-x/y
    int dy;    //边跨越的扫描线数

    float z;   //端点处多边形所在平面的深度值
    float dzx; //扫描线向右走过一个像素时，点的深度增量：z/x
    float dzy; //扫描线向下走过一个像素时，点的深度增量：-z/y

    int pid;   //所属的多边形编号
};
```

```
struct ActiveEdge{
    float x;  //交点的x坐标
    float dxy; //扫描线向下走过一个像素，点的x增量：-x/y
    float dy; //边跨越的扫描线数,随着扫描线下移减1

    float z;   //端点处多边形所在平面的深度值
    float dzx; //扫描线向右走过一个像素时，面的深度增量：z/x
    float dzy; //扫描线向下走过一个像素时，面的深度增量：-z/y

    int pid;   //所属的多边形编号
};
```

```
struct Polygon{
    int pid;           //多边形编号
    float a, b, c, d; //多边形所在平面方程
    int dy;            //多边形跨越的扫描线数
    Float3 color;      //多边形颜色
};
```

```
struct ActivePolygon{
    int pid;           //多边形编号
    float a, b, c, d; //多边形所在平面方程
    int dy;            //多边形跨越的扫描线数
    Float3 color;      //多边形颜色
    bool flag;         //是否进入、离开
};
```

可以看到，边结构和活边结构、多边形结构和活多边形结构的区别并不大。对于活边来说，其最大区别在于dy这个成员变量是可变的而边是常量；对于活多边形来说，除了dy是可变的，还多了一个flag来判断扫面线的进入和离开。

2.4 加速算法

在ScanLineBuffer.h的头文件中，加入#include <libiomp/omp.h>，并对ScanLienBuffer.cpp里的init函数中的for循环进行并行操作，操作框架为：

```
void ScanLineZBuffer::Init(Model &model){
    //…..
    omp_lock_t mylock;
    omp_init_lock(&mylock);
#pragma omp parallel for
    for (int i = 0; i < faces_size; ++i){
        //…
        omp_set_lock(&mylock);
        edgeTable[round(pt1.y)].push_back(edge);
        omp_unset_lock(&mylock);
        //…
        omp_set_lock(&mylock);
        polygonTable[round(max_y)].push_back(polygon);
        omp_unset_lock(&mylock);
        //….

    }
    omp_destroy_lock(&mylock);
}
```

三、 核心算法（在ScanLineBuffer.cpp中）

3.1 更新frameBuffer

```
//更新缓存
int i = 0;
for(int j = 0; j < activeEdgeTable.size(); ++j){
     //...
     if(activeEdgeTable[j].x == crosses[i]){
          //改变对应activePolygon的flag
     }
      else{
          //找到flag==true的所有activePolygon，存储在truePolygon里

          for(int n = 0; n < truePolygon.size(); ++n){
               //遍历flag==true的activePolygon，找到深度最大的
          }
          //赋值frameBuffer相应位置的值
          --j;
          ++i;
      }
}
```

3.2 更新活多边形表和边表

```
//更新活多边形表
for(vector<ActivePolygon>::iterator itr = activePolygonTable.begin(); itr != activePolygonTable.end(); ){
    --(*itr).dy;
    if(((*itr).dy <= 0))
        activePolygonTable.erase(itr);
    else
        ++itr;
}

//更新活边表
for(vector<ActiveEdge>::iterator itr = activeEdgeTable.begin(); itr != activeEdgeTable.end(); ){
    --(*itr).dy;
    if((*itr).dy <= 0)
        activeEdgeTable.erase(itr);
    else{
        (*itr).x += (*itr).dxy;
        (*itr).z += (*itr).dzx * (*itr).dxy + (*itr).dzy;
        ++itr;
    }
}
```

四、使用说明

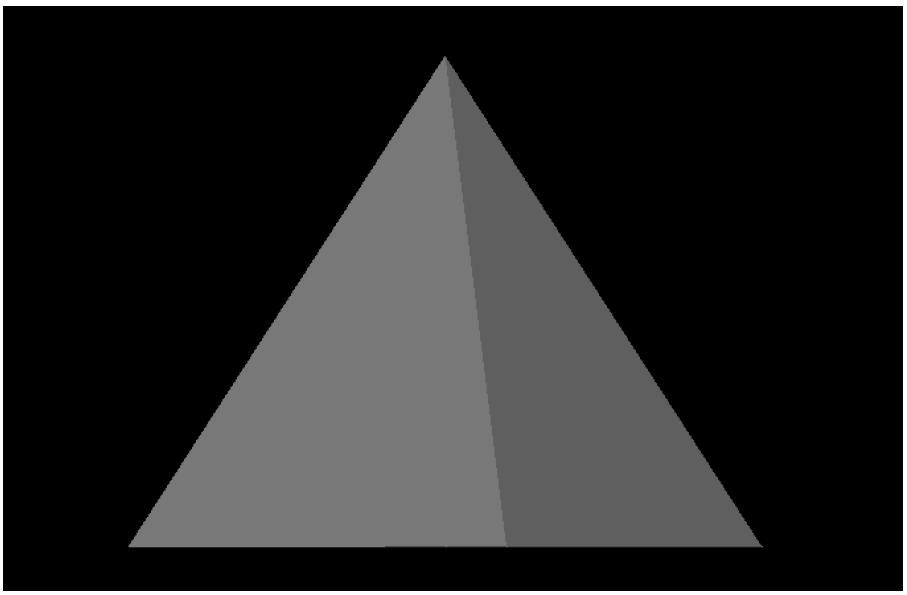
在main.cpp文件中，有两个路径，一个是根路径，一个是文件路径，同个修改这两个路径，就可以改变显示的模型。（本项目的模型文件全部统一放在 OBJ文件夹中）。

```
string ROOT = "/Users/yy/Desktop/CG/OBJ/";
string filePath = ROOT + "plane.obj";
```

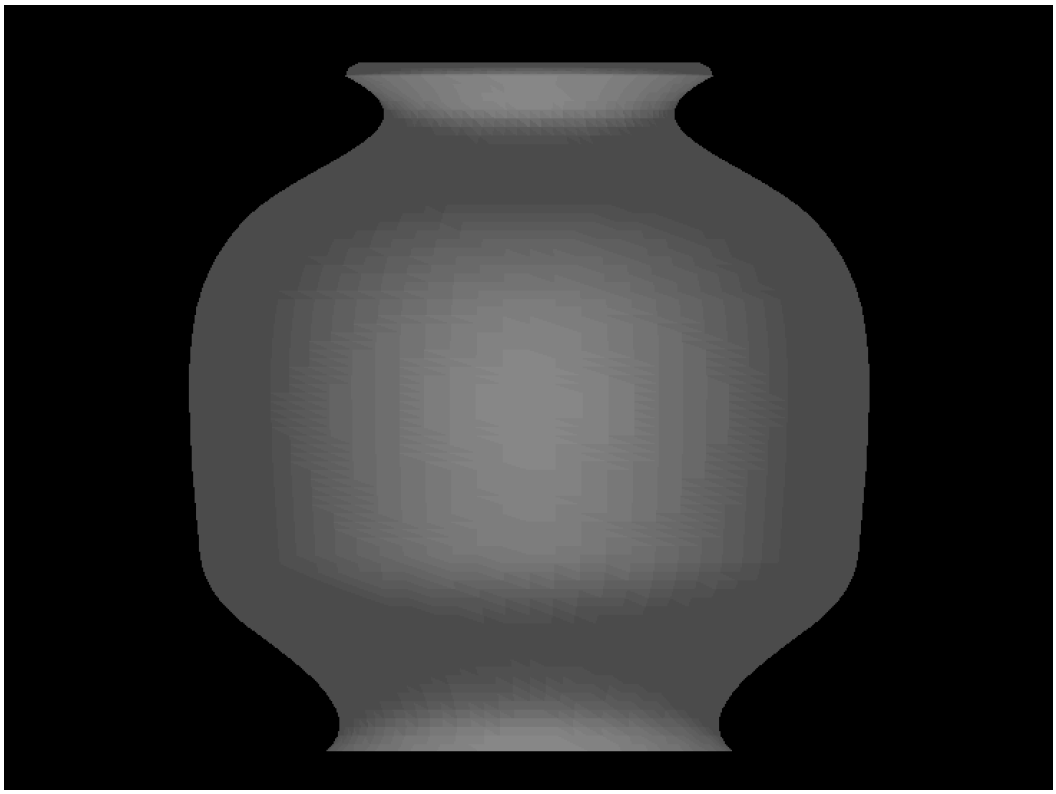
五、实现效果

5.1 简单的立方体

5.2 简单的四棱锥



5.3 面片大于1000的模型，面片数目分别为1024、27680、69451

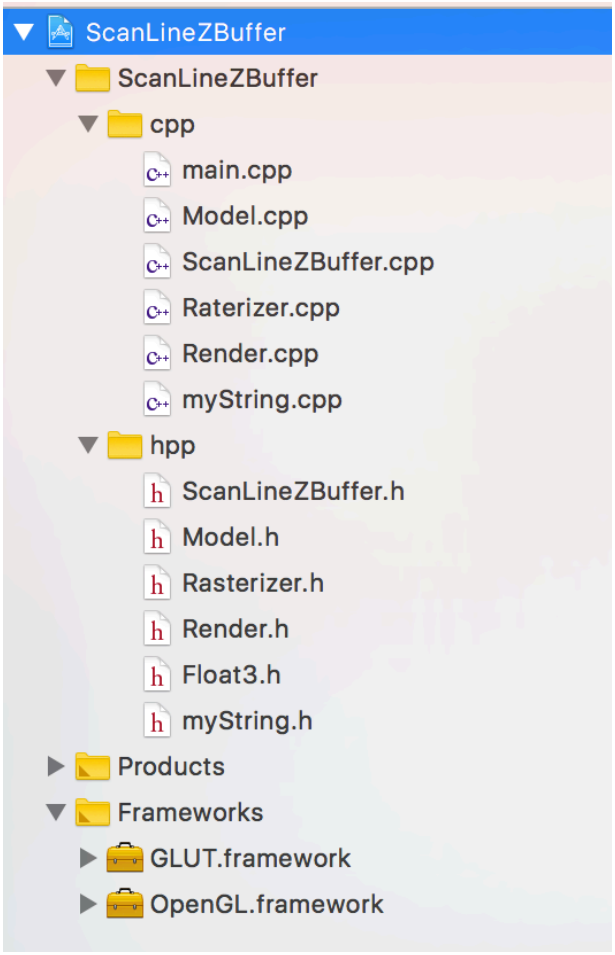


5.4 耗时计算：下图为显示兔子的耗时

```
Load model successfully.
Rasterizer successfully.
Render successfully.
Table init successfully.
耗时:4.04263s
```

六、源代码文件目录说明

本项目开发文件为ScanLineBuffer，其目录结构如下，头文件放在hpp文件夹里，定义文件放在cpp文件夹里，frameworks下面是项目开发中需要用到的框架。



项目的测试文件放在OBJ文件夹里，实现结果放在results文件夹里。