A5: Voxel Rendering

一、概述

1.目标：

（1）设计网格加速的数据结构
（2）绘制体素
（3）实现射线和网格中的交互

2.知识点介绍：

```
create grid
insert primitives into grid
for each ray r
  find initial cell c(i,j), t_min, t_next_x & t_next_y
  compute dt_x, dt_y, sign_x and sign_y
  while c != NULL
    for each primitive p in c
      intersect r with p
      if intersection in range found
        return
    c = find next cell
```

算法如图：

（1）首先创建一个 grid，可以设置成无限大的。然后创建各个图元的 boundingbox 并插入到 grid 中（将对应 cell 设置为不透明的）。

（2）之前在进行射线和物体的求交过程中，都是单独调用每个图元的 intersect 函数来单独求交。但是现在射线需要先和 grid 进行求交，利用 MarchingInfo，可以模拟 ray 在 grid 中的行进。如果找到不为空的 cell，再和 cell 中的每个物体进行求交，找到最近的交点，返回给像素颜色。否则，继续找下一个 cell。通过这样的方式可以避免每条射线对场景中的所有物体进行运算，大大提高效率。

（3）在 OpenGL 中绘制体素，就是判断 Grid 中的每个 cell 是否为空，如果不为空，就画出一个小立方体。

二、实现细节

1. 创建 Bounding Box 的过程：

对于球体，包围盒的最小值是球心减去半径，最大值是球心加上半径。对于三角形，最小值是三个顶点中各个轴上的最小值，最大值则是三个顶点在各个轴上的最大坐标。对于平面，因为是没有边界的，所以不设置包围盒。对于 Group，包围盒则是物体数组中各个包围盒的并集。

2.将 Bounding Box 插入 Grid：

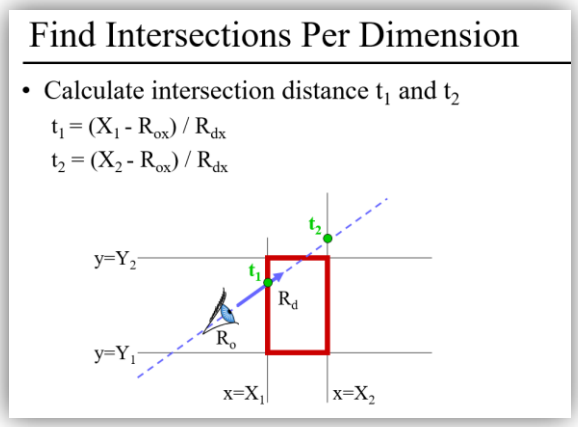
在 Grid 的中设置一个存储每个 cell 的透明度的 vector 和存储每个 cell 中所有物体的 vector。所谓将 Bounding Box 插入，其实就是将包含物体的 cell 设置为可见。对于球体，可以比较球心到每个 cell 的距离，将这个距离和半径进行比较，如果距离小于半径，则将该 cell 设置为可见（但是这种方法其实并不准确）。因为三角形的包围盒也是长方体，所以可以直接将该包围盒的最小值和最大值直接和每个 cell 的最小值和最大值进行比较。

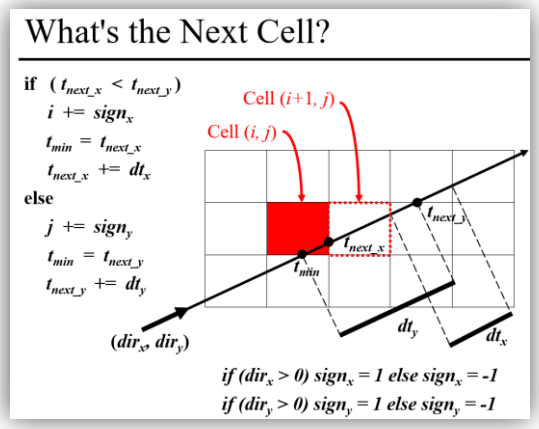
```
void Triangle::insertIntoGrid(Grid *g, Matrix *m){
    //此三角形的包围盒
    m_min = boundingBox->getMin ();
    m_max = boundingBox->getMax ();
    //grid的包围盒
    Vec3f gird = g->getGird();
```

```
    BoundingBox *bb = g->getBoundingBox();
    Vec3f g_min =bb->getMin();
    Vec3f g_max = Vec3f(bb->getMax().x() + FLT_EPSILON, bb->getMax().y() +
FLT_EPSILON, bb->getMax().z() + FLT_EPSILON);
    int nx = gird.x();    int ny = gird.y();    int nz = gird.z();
    Vec3f size = g_max - g_min;
    float grid_x = size.x() / nx;  float grid_y = size.y() / ny;  float grid_z =
size.z() / nz;
    //遍历grid的每个cell，如果三角形包围盒最小值大于它的最小值且最大值大于它的最大
值，就置为真
    for (int _i = 0; _i < nx; _i++){
        for (int _j = 0; _j < ny; _j++){
            for (int _k = 0; _k < nz; _k++){
                Vec3f cellMin (_i*grid_x, _j*grid_y, _k*grid_z); cellMin += g_min;
                Vec3f cellMax ((_i + 1)*grid_x, (_j + 1)*grid_y, (_k + 1)*grid_z);
cellMax += g_min;
                if (inBound (cellMin, cellMax)){
                    g->insertIntoThis ((_i * ny + _j) * nz + _k, true, this);
                }}}} }
```

3. ray 和 grid 进行交互：

要通过 MarchingInfo 来模拟 ray 的行进，首先要找到 ray 和 grid 的最近交点，也就是 marchiing 的起点。这个过程用到了求射线和 box 相交的方法。要在每一维上分别计算 ray 和包围盒的交点距离。找到 tmin 之后，根据计算得到的 sign 和 dt 找到 tnext。得到这个 cell 的索引之





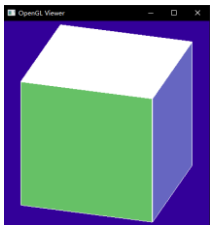
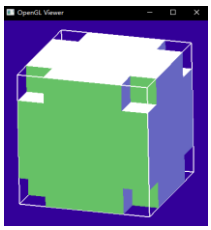
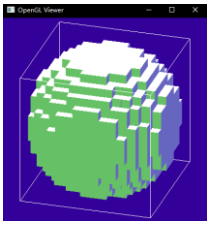
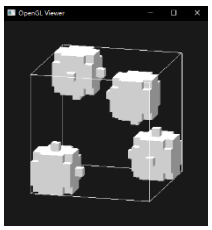
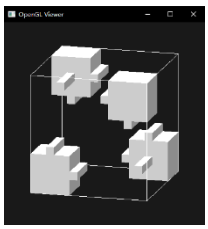
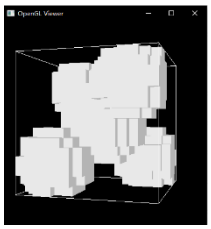
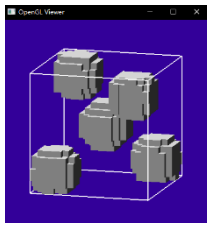
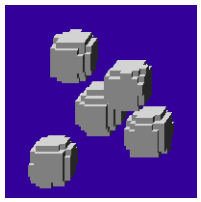
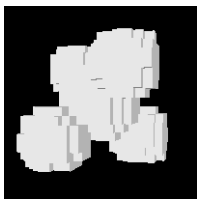
后，判断这个 cell 是否可见以及如果可见，里面的 obj 数目是多少，决定最后显示的颜色。

```
    bool Grid::intersect (const Ray &r, Hit &h, float tmin){
    bool result = false;
    MarchingInfo mi;
    initializeRayMarch (mi, r, tmin);//初始化mi中的变量
    if (mi.tmin < h.getT ()) {
        while (mi.i < nx && mi.j < ny && mi.k < nz &&
```

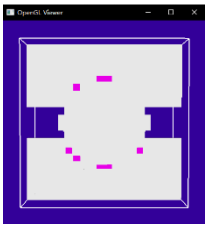
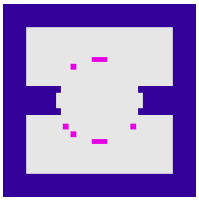
```
            mi.i >= 0 && mi.j >= 0 && mi.k >= 0) {//在grid 的range内
            int index = (mi.i * ny + mi.j) * nz + mi.k;
            if (m_is_voxel_opaque[index]) {
                    changeMaterial (index);
                    h.set(mi.tmin, material, mi.normal, r);
                    result = true; break; }
            mi.nextCell ();
        }   }    return result;}
```

三、结果展示

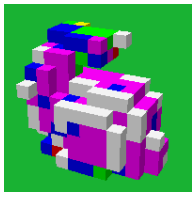
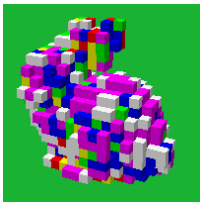
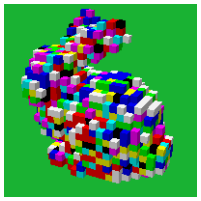
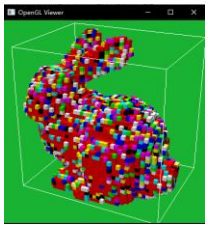
1.球体测试:

2.三角形测试:

3.overlap 测试:

四、心得体会

沿着光线行进，可以只与在光线路径上的物体进行求交操作，大大降低了计算量。从暴力地和场景中所有物体进行交互到之和 grid 进行交互，增强目的性的同时提高了效率。