南京大学

计算机科学与技术系

图形绘制技术

Lab1 实验报告

实验名称： 物体求交以及加速结构

学　　号： 201830204

姓　　名： 顾秋涵

实验时间： 2023.3

目　　录

一、 物体求交

1. 圆环与光线的求交

1.1 求交逻辑

- 光线变换到局部空间
- 判断局部光线的方向在 z 轴是否为 0
- 通过几何信息计算光线与平面的交点：光线到达该平面的时间

$$t = \frac{(0 - ray.origin.z)}{ray.direction.z}$$

- 检验交点是否在圆环内：包括 t 与光线的 tFar 和 tNear 的大小关系、交点是否在圆环内、夹角和 phiMax 的大小关系
- 更新 ray 的 tFar，减少光线与其他物体的相交计算次数
- 填充参数

1.2 填充求交信息

- 计算出局部空间的法线，变换到世界空间
- 根据 uv 计算出位置信息，变换到世界空间

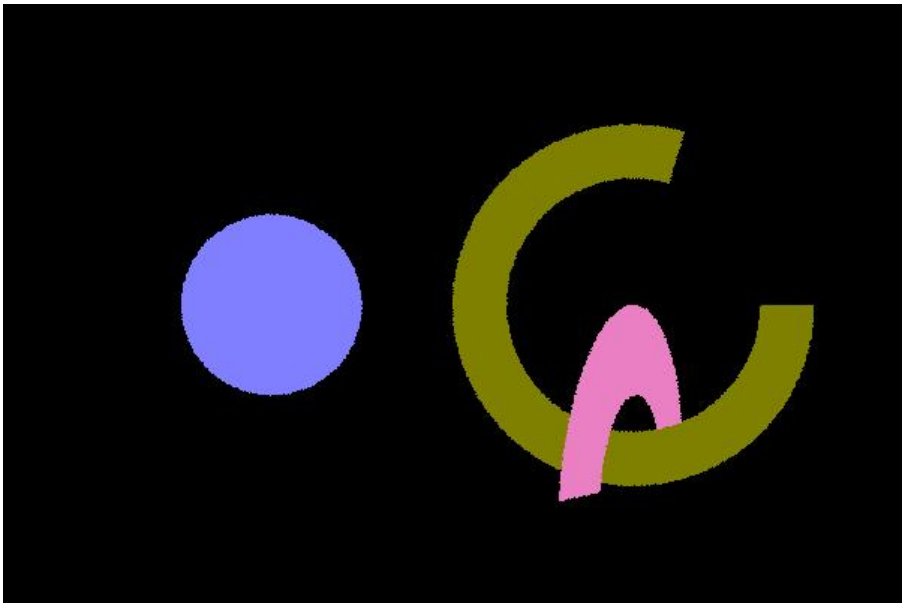
1.3 代码实现

```
7   bool Disk::rayIntersectShape(Ray &ray, int *primID, float *u, float *v) const {
8       //* todo 完成光线与圆环的相交 填充primId,u,v.如果相交，更新光线的tFar
9       //* 1.光线变换到局部空间
10      //每个shape有自己的tranform
11      Ray newRay=transform.inverseRay(ray);
12      //* 2.判断局部光线的方向在z轴分量是否为0
13      if(newRay.direction[2]==0) return false;
14      //* 3.计算光线和平面交点
15      float t=(0-newRay.origin[2])/(newRay.direction[2]);
16      Point3f p=newRay.at(t);
17
18      //* 4.检验交点是否在圆环内
19      if(t<=newRay.tNear||t>=newRay.tFar) return false;
20      //这里不确定如何访问三维坐标，下标012表示xyz?--yes
21      float r=sqrt(p[0]*p[0]+p[1]*p[1]);
22      if(r>radius||r<innerRadius) return false;
23      //检验夹角
24      float phi=atan2(p[1],p[0]);//atan2 (y, x) 求的是y/x的反正切，其返回值为[-pi,+pi]之间的一个数
25      if(phi<0) phi+=2*PI;
26      if(phi>phiMax) return false;
27
28      //* 5.更新ray的tFar,减少光线和其他物体的相交计算次数
29      ray.tFar=t;//一定要更新原来的ray！！
30      //* 6.填充
31      * primID=0;
32      * u =phi/phiMax;
33      * v =(r-innerRadius)/(radius-innerRadius);
34      return true;
35      //* Write your code here.
36      //return false;
37  }
```

```cpp
void Disk::fillIntersection(float distance, int primID, float u, float v, Intersection *intersection) con
    /// ------------------------------------------------------
    //* todo 填充圆环相交信息中的法线以及相交位置信息
    //* 1.法线可以先计算出局部空间的法线，然后变换到世界空间
    Vector3f normal(0.f,0.f,1.f);
    intersection->normal=transform.toWorld(normal);//法线已经归一化了
    //* 2.位置信息可以根据uv计算出，同样需要变换
    float phi=phiMax*u;
    float r=v*(radius-innerRadius)+innerRadius;
    float y=r/(sqrt(1/(tan(phi)*tan(phi))+1));
    if(phi>PI)//三四象限
    {
        y=-y;
    }
    float x=y/tan(phi);

    //printf("\nbefore: %lf,%lf\n",x,y);
    intersection->position=transform.toWorld(Point3f(x,y,0.f));
    //printf("after: %lf,%lf\n\n",intersection->position[0],intersection->position[1]);
    //* Write your code here.
    /// ------------------------------------------------------
```

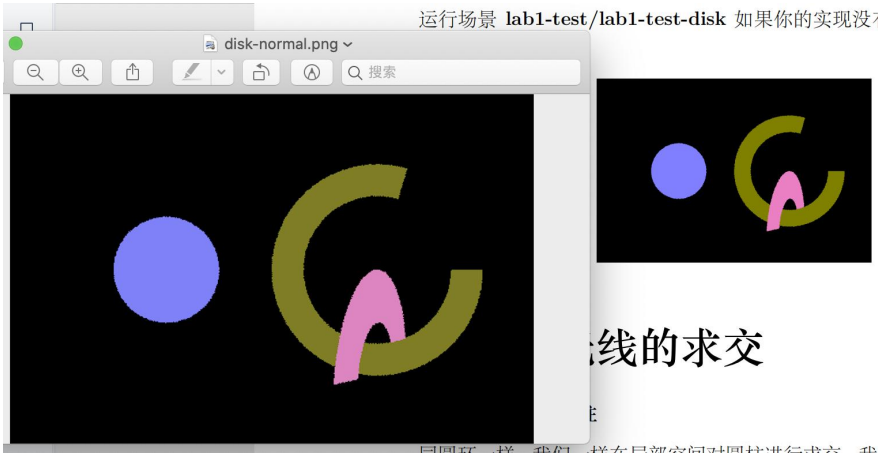
1.4 结果验证

得到渲染结果：



与 lab 文档中进行对比，一致。

运行场景 **lab1-test/lab1-test-disk** 如果你的实现没有问题，那么你将得到以



线的求交

2. 圆柱与光线的求交

2.1 求交逻辑

- 光线变换到局部空间
- 通过联立方程组计算光线与平面的交点：方程组为

$$(ray.origin.x + ray.direction.x * t)^2 + (ray.origin.y + ray.direction.y * t)^2 = r^2$$

- 检验两个交点是否在圆柱内：包括 t 与光线的 tFar 和 tNear 的大小关系、交点是否在圆柱内、夹角和 phiMax 的大小关系。若两个交点都满足，取最近的 t
- 更新 ray 的 tFar，减少光线与其他物体的相交计算次数
- 填充参数

2.2 填充求交信息

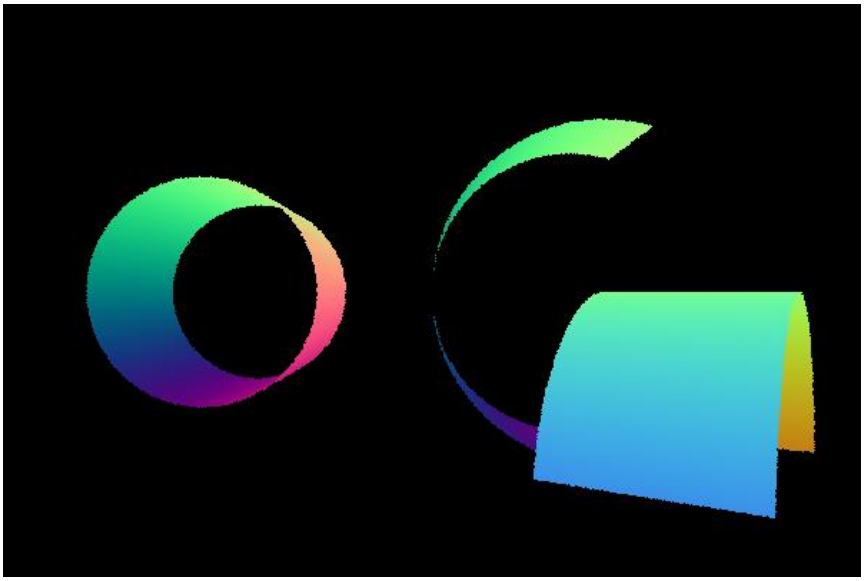
- 计算出局部空间的法线，变换到世界空间
- 根据 uv 计算出位置信息，变换到世界空间

2.3 代码实现

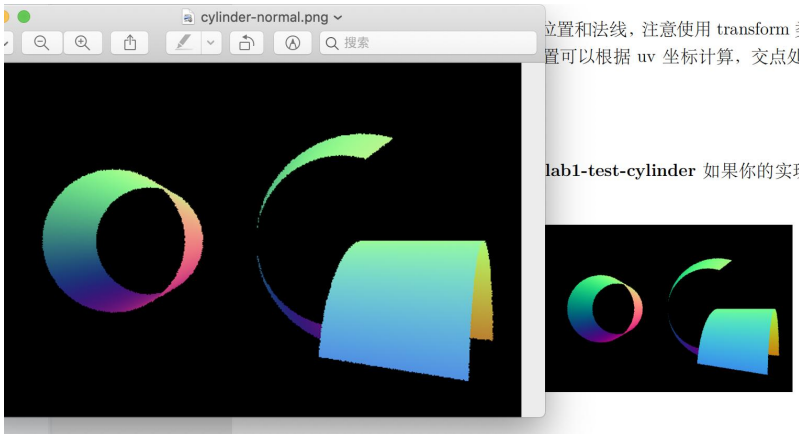
```cpp
bool Cylinder::rayIntersectShape(Ray &ray, int *primID, float *u, float *v) const {
    //* todo 完成光线与圆柱的相交 填充primId,u,v.如果相交，更新光线的tFar
    //* 1.光线变换到局部空间
    Ray newRay=transform.inverseRay(ray);

    //* 2.联立方程求解
    float t0,t1;
    float A=newRay.direction[0]*newRay.direction[0]+newRay.direction[1]*newRay.direction[1];
    float B=2*newRay.origin[0]*newRay.direction[0]+2*newRay.origin[1]*newRay.direction[1];
    float C=newRay.origin[0]*newRay.origin[0]+newRay.origin[1]*newRay.origin[1]-radius*radius;
    if(!Quadratic(A,B,C,&t0,&t1)) return false;//无解
    //* 3.检验交点是否在圆柱范围内
    Point3f p1=newRay.at(t1);
    bool flag0=true;
    bool flag1=true;
    //检验t0
    if(t0<newRay.tNear||t0>newRay.tFar) flag0=false;
    if(p0[2]<0||p0[2]>height) flag0=false;
    float phi0=atan2(p0[1],p0[0]);//atan2（y，x）求的是y/x的反正切，其返回值为[-pi,+pi]之间的一个数
    if(phi0<0) phi0+=2*PI;
    if(phi0>phiMax) flag0=false;
    //检验t1
    if(t1<newRay.tNear||t1>newRay.tFar) flag1=false;
    if(p1[2]<0||p1[2]>height) flag1=false;
    float phi1=atan2(p1[1],p1[0]);//atan2（y，x）求的是y/x的反正切，其返回值为[-pi,+pi]之间的一个数
    if(phi1<0) phi1+=2*PI;
    if(phi1>phiMax) flag1=false;
    //* 4.更新ray的tFar,减少光线和其他物体的相交计算次数
    if(flag0)//取近的一个
    {


    //* Write your code here.
    //return false;
}

void Cylinder::fillIntersection(float distance, int primID, float u, float v, Intersection *intersectio
    /// ------------------------------------------------------
    //* todo 填充圆柱相交信息中的法线以及相交位置信息
    //* 1.法线可以先计算出局部空间的法线，然后变换到世界空间
    float phi=u*phiMax;
    Vector3f normal(cos(phi),sin(phi),0.f);
    intersection->normal=transform.toWorld(normal);//法线已经归一化了！！
    //* 2.位置信息可以根据uv计算出，同样需要变换
    float y=radius/(sqrt(1/(tan(phi)*tan(phi))+1));
    if(phi>PI)//三四象限
    {
        y=-y;
    }
    float x=y/tan(phi);
    intersection->position=transform.toWorld(Point3f(x,y,v*height));
    //* Write your code here.
    /// ------------------------------------------------------


    intersection->shape = this;
    intersection->distance = distance;
    intersection->texCoord = Vector2f{u, v};
    Vector3f tangent{1.f, 0.f, .0f};
    Vector3f bitangent;
    if (std::abs(dot(tangent, intersection->normal)) > .9f) {
        tangent = Vector3f(.0f, 1.f, .0f);
    }
    bitangent = normalize(cross(tangent, intersection->normal));
    tangent = normalize(cross(intersection->normal, bitangent));
    intersection->tangent = tangent;
    intersection->bitangent = bitangent;
}
```

2.4 结果验证

得到渲染结果：



与 lab 文档中进行对比，一致。



位置和法线，注意使用 transform 类的 toWorld 将信息转换
置可以根据 uv 坐标计算，交点处的法线沿该点半径方向。

lab1-test-cylinder 如果你的实现没有问题，那么你将得

3. 圆锥与光线的求交

3.1 求交逻辑

- 光线变换到局部空间
- 通过联立方程组计算光线与平面的交点：方程组为

$$t^2((\vec{D}\cdot\vec{V})^2-\cos^2\theta)+2t((\vec{D}\cdot\vec{V})(\vec{CO}\cdot\vec{V})-\vec{D}\cdot\vec{CO}\cos^2\theta)+(\vec{CO}\cdot\vec{V})^2-\vec{CO}\cdot\vec{CO}\cos^2\theta=0$$

- 检验两个交点是否在圆锥内：包括 t 与光线的 tFar 和 tNear 的大小关系、交点是否在圆锥内、夹角和 phiMax 的大小关系。若两个交点都满足，取最近的 t
- 更新 ray 的 tFar，减少光线与其他物体的相交计算次数
- 填充参数

3.2 填充求交信息

- 计算出局部空间的法线，变换到世界空间
- 根据 uv 计算出位置信息，变换到世界空间

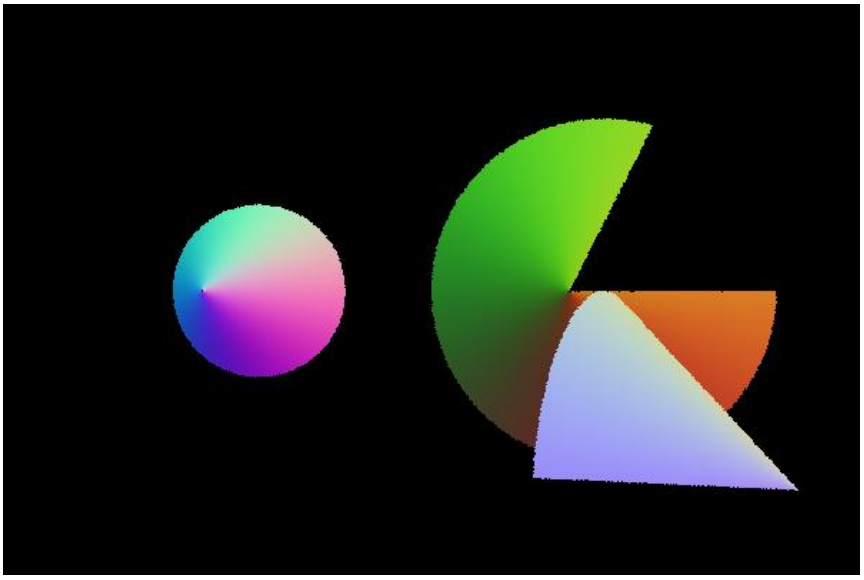
3.3 代码实现

```cpp
bool Cone::rayIntersectShape(Ray &ray, int *primID, float *u, float *v) const {
    //* todo 完成光线与圆锥的相交 填充primId,u,v.如果相交，更新光线的tFar
    //* 1.光线变换到局部空间
    Ray newRay=transform.inverseRay(ray);
    //* 2.联立方程求解
    Vector3f D(newRay.direction[0],newRay.direction[1],newRay.direction[2]);
    Vector3f V(0.f,0.f,-1.f);
    Vector3f CO(newRay.origin[0],newRay.origin[1],newRay.origin[2]-height);
    float A=pow(dot(D,V),2)-pow(cosTheta,2);
    float B=2*(dot(D,V)*dot(CO,V)-dot(D,CO)*pow(cosTheta,2));
    float C=pow(dot(CO,V),2)-dot(CO,CO)*pow(cosTheta,2);
    float t0,t1;

    //* 3.检验交点是否在圆锥范围内
    Point3f p0=newRay.at(t0);
    Point3f p1=newRay.at(t1);
    bool flag0=true;
    bool flag1=true;
    //检验t0
    if(t0<newRay.tNear||t0>newRay.tFar) flag0=false;
    if(p0[2]<0||p0[2]>height) flag0=false;
    float phi0=atan2(p0[1],p0[0]);//atan2（y，x）求的是y/x的反正切，其返回值为[-pi,+pi]之间的一个数
    if(phi0<0) phi0+=2*PI;
    if(phi0>phiMax) flag0=false;

    //检验t1
    if(t1<newRay.tNear||t1>newRay.tFar) flag1=false;
    if(p1[2]<0||p1[2]>height) flag1=false;
    float phi1=atan2(p1[1],p1[0]);//atan2（y，x）求的是y/x的反正切，其返回值为[-pi,+pi]之间的一个数
    if(phi1<0) phi1+=2*PI;
    if(phi1>phiMax) flag1=false;
    //* 4.更新ray的tFar,减少光线和其他物体的相交计算次数
    if(flag0)//取近的一个
    {
        ray.tFar=t0;
        * primID=0;
        * u =phi0/phiMax;
        * v =p0[2]/height;
    }

    else {
        if(!flag1) return false;
        ray.tFar=t1;
        * primID=0;
        * u =phi1/phiMax;
        * v =p1[2]/height;
    }

    return true;
    //* Write your code here.
    //return false;
}
```

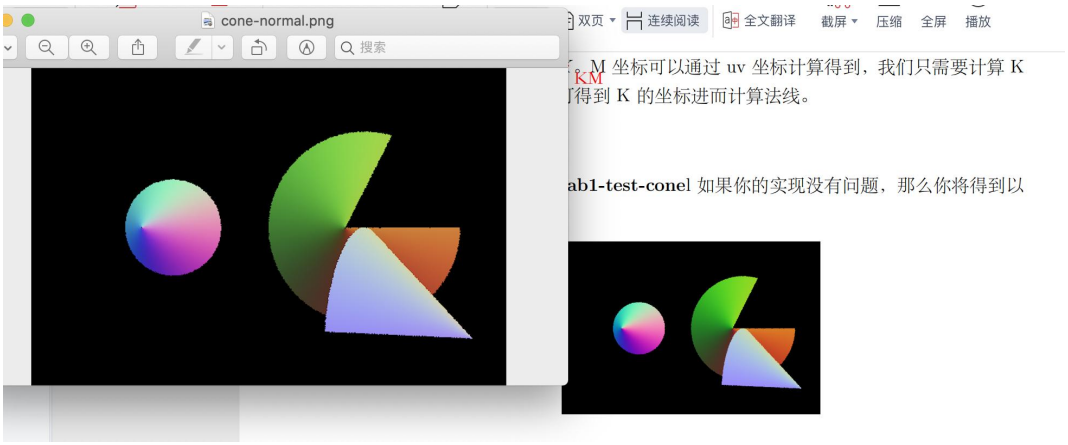
```
58
59  void Cone::fillIntersection(float distance, int primID, float u, float v, Intersection *intersection)
60      /// ------------------------------------------------------------
61      //* todo 填充圆锥相交信息中的法线以及相交位置信息
62
63      //* 2.位置信息可以根据uv计算出，同样需要变换
64      float phi=u*phiMax;
65      float y=radius/(sqrt(1/(tan(phi)*tan(phi))+1));
66      if(phi>PI)//三四象限
67      {
68          y=-y;
69      }
70      float x=y/tan(phi);
71      Point3f M(x,y,v*height);
72      intersection->position=transform.toWorld(M);
73
74      //* 1.法线可以先计算出局部空间的法线，然后变换到世界空间
75      Vector3f CM(M[0],M[1],M[2]-height);
76      float cm=CM.length();
77      Point3f K(0.f,0.f,height-cm/cosTheta);
78      Vector3f KM(M[0]-K[0],M[1]-K[1],M[2]-K[2]);
79      float len=KM.length();
80      Vector3f normal(KM[0]/len,KM[1]/len,KM[2]/len);//归一化处理！！！
81      intersection->normal=transform.toWorld(normal);
82      //* Write your code here.
83      /// ------------------------------------------------------------
84
85
86      intersection->shape = this;
87      intersection->distance = distance;
88      intersection->texCoord = Vector2f{u, v};
89      Vector3f tangent{1.f, 0.f, .0f};
90      Vector3f bitangent;
91      if (std::abs(dot(tangent, intersection->normal)) > .9f) {
92          tangent = Vector3f(.0f, 1.f, .0f);
93      }
94      bitangent = normalize(cross(tangent, intersection->normal));
95      tangent = normalize(cross(intersection->normal, bitangent));
96      intersection->tangent = tangent;
97      intersection->bitangent = bitangent;
98  }
```

3.4 结果验证

得到渲染结果：



与 lab 文档中进行对比，一致。



二、 加速结构（均已做）

1. Octree

1.1 Octree 节点结构

- 包围盒
- 存储的物体 id（叶子节点）
- 存储的 8 个子节点（非叶子节点）
- 存储的物体数量（非叶子节点设置为-1）
- 叶子节点和非叶子节点的构造函数

```
struct Octree::OctreeNode {
    AABB boundingBox;
    OctreeNode* subNodes[8];//8个子节点的指针
    int primCount = -1;
    std::vector<int> primIdxBuffer;//叶节点存储的物体id,可能超过最大数量，当某一包围盒与当前节点的所有物体都相交时

    //构造函数
    OctreeNode(const AABB& box, const std::vector<int> &idxBuffer)//叶子结点
    {
      //填写存储的图元
      primIdxBuffer=idxBuffer;
      primCount=idxBuffer.size();//图元数量

      //填写包围框
      boundingBox=box;

      //填写子节点，叶子节点无子节点
      for(int j=0; j<8; ++j) subNodes[j]=nullptr;

    }

    OctreeNode(const AABB& box)//非叶子节点
    {
        boundingBox=box;
        primCount=-1;//非叶子节点下没有图元
    }


};
```

1.2 Octree 构建

- 计算出整个场景的范围
- 从根节点开始递归构建八叉树
- 递归时
  - 如果递归深度到达最大值或者当前节点数小于一定值时，当前节点作为叶子节点返回。

• 否则将节点八等分，用八个子节点表示。子节点的包围盒通过空间划分得到。子节点索引数组通过寻找和子节点包围盒有重叠区域的物体得到，遍历子节点递归。

• 特殊的，当节点的某个子包围盒和当前节点所有物体都相交，直接当前节点作为叶子节点返回。

```
Octree::OctreeNode * Octree::recursiveBuild(const AABB &aabb,const std::vector<int>& primIdxBuffer,int de
  //* todo 完成递归构建八叉树
  //* 构建方法请看实验手册
  //* 要注意的一种特殊是当节点的某个子包围盒和当前节点所有物体都相交，我们就不用细分了，当前节点作为叶子节点即可。
  if(primIdxBuffer.size()<ocLeafMaxSize ||depth>=maxDepth)//先不管深度?
  {
      OctreeNode* node = new OctreeNode(aabb,primIdxBuffer);
      return node;
  }

  std::vector<AABB> subBoxes=aabb.getSubBoxs();//分成8个子盒子
  std::vector<std::vector<int>> subBuffers(8);//8个盒子分别对应的子节点

  for(int i = 0; i<8 ;i++)
  {
      for(auto index : primIdxBuffer)
      {
        if(shapes[index]->getAABB().Overlap(subBoxes[i]))
        {
          subBuffers[i].push_back(index);
        }
      }
      if(subBuffers[i].size()==primIdxBuffer.size())//子包围盒和所有物体都相交
      {
        OctreeNode* node = new OctreeNode(subBoxes[i],primIdxBuffer);
        return node;
      }
  }

  OctreeNode* node = new OctreeNode(aabb);//创建一个非叶子节点
  for(int i = 0;i<8;i++)
  {
    node->subNodes[i]=recursiveBuild(subBoxes[i],subBuffers[i],depth+1);
  }
  return node;
}


void Octree::build() {
  //* 首先计算整个场景的范围
  for (const auto & shape : shapes) {
    //* 自行实现的加速结构请务必对每个shape调用该方法，以保证TriangleMesh构建内部加速结构
    //* 由于使用embree时，TriangleMesh::getAABB不会被调用，因此出于性能考虑我们不在TriangleMesh
    //* 的构造阶段计算其AABB，因此当我们将TriangleMesh的AABB计算放在TriangleMesh::initInternalAcceleration中
    //* 所以请确保在调用TriangleMesh::getAABB之前先调用TriangleMesh::initInternalAcceleration
    shape->initInternalAcceleration();

    boundingBox.Expand(shape->getAABB());
  }

  //* 构建八叉树
  std::vector<int> primIdxBuffer(shapes.size());//所有的shape
  std::iota(primIdxBuffer.begin(), primIdxBuffer.end(), 0);

  root = recursiveBuild(boundingBox, primIdxBuffer,0);//从根节点开始递归构建树

}

bool Octree::rayIntersect(Ray &ray, int *geomID, int *primID,float *u, float *v) const {
  //*todo 完成八叉树求交

  return this->recursiveRayIntersect(root,ray,geomID,primID,u,v);
  //return false;
}
```

1.3 Octree 求交

• 递归过程，通过节点的包围盒对与光线是否相交进行剪枝来进行加速

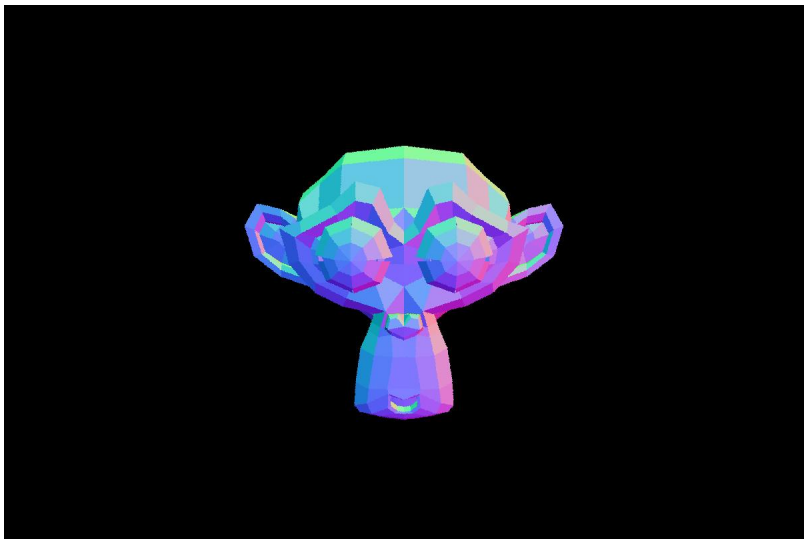
• 对于非叶子节点，首先判断当前空间是否和光线相交，如果不相交可以直接返回, 否则遍历每个子节点。

• 对于叶子节点，遍历叶子节点下的所有物体进行求交。

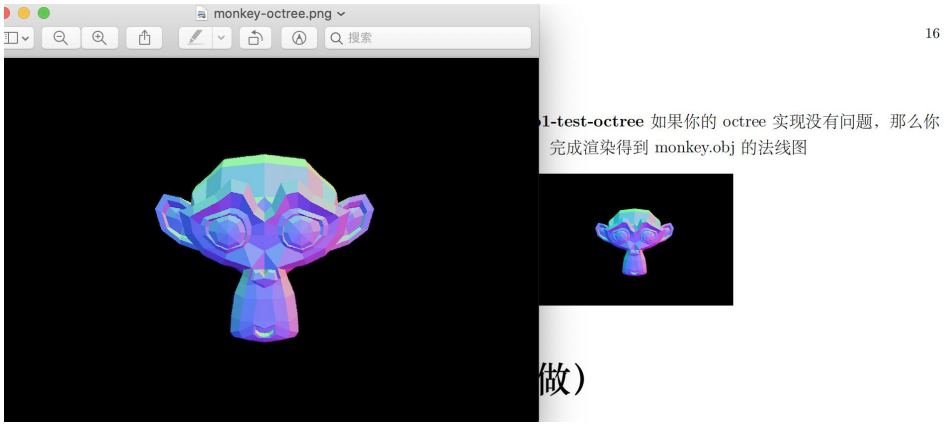
```cpp
bool Octree::rayIntersect(Ray &ray, int *geomID, int *primID,float *u, float *v) const {
  //*todo 完成八叉树求交
  return this->recursiveRayIntersect(root,ray,geomID,primID,u,v);
  //return false;
}



bool Octree::recursiveRayIntersect(OctreeNode* node, Ray &ray, int *geomID, int *primID, float *u, flo
{
  if(node==nullptr) return false;
  if(!node->boundingBox.RayIntersect(ray,nullptr,nullptr)) return false;

  //相交
  //对于非叶子节点
  if(node->primCount==-1)
  {
    bool flag=false;
    for(auto subNode:node->subNodes)
    {
      //if( subNode==nullptr) continue;
      flag|=recursiveRayIntersect(subNode,ray,geomID,primID,u,v);
      //if(flag==true) break;//不要break, 要找最终的相交
    }
    return flag;
  }

  else if(node->primCount!=0)//叶子节点
  {
    bool flag=false;
    for(auto idx:node->primIdxBuffer)
    {
      bool re=shapes[idx]->rayIntersectShape(ray,primID,u,v);
      flag|=re;
      if(re==true)
      {
        *geomID=shapes[idx]->geometryID;
        //break;
      }
    }
    return flag;
  }

  return false;

}
```

1.4 结果验证

得到渲染结果:



与 lab 文档中对比，一致。



用时在 5s 以内。

```
pro@prodeMacBook-Pro-2 bin % ./Moer /Users/pro/Desktop/Rendering_labs/Moer-lite/
examples/lab1-test/lab1-test-octree
Using acceleration type octree
100% [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||]
Rendering costs 3.06s
```

2. BVH

2.1 BVH 节点结构

- 包围盒
- 左右子节点（非叶子节点）
- 分割轴（非叶子节点）
- 第一个物体的索引（叶子节点）
- 存储的物体数量（非叶子节点设置为-1）
- 构造函数（默认为非叶子节点）

```cpp
struct BVH::BVHNode
{
    //* todo BVH节点结构设计
    BVHNode * left;
    BVHNode * right;//左右节点
    AABB boundingBox;

    //叶子节点存节点索引
    int firstShapeOffset;//第一个物体的索引
    int nShape = 0;//一共有多少物体

    int splitAxis;//非叶子节点的分割轴，求交部分用

    BVHNode(const AABB& aabb)
    {
        boundingBox=aabb;
        left=nullptr;
        right=nullptr;
        nShape=-1;//默认为非叶子节点
    }

};
```

2.2 BVH 构建

- 计算出整个场景的范围
- 从根节点开始递归构建，按中点分割
- 递归时
  - 如果递归深度到达最大值或者当前节点数小于一定值时，当前节点作为叶子节点返回。
  - 否则
    - 计算出所有物体中心构成的包围盒，选取跨度最大的坐标轴作为分割轴
    - 将所有物体的中心在分割轴上排序
    - 按照顺序将所有物体均匀分成两部分
    - 特殊的，如果所有物体的中心重合，则直接当前节点作为叶子节点返回
    - 递归构造子节点
- 给物体排序时，采用快排

```cpp
void BVH::build()
{
    //AABB sceneBox;
    for (const auto &shape : shapes)
    {
        //* 自行实现的加速结构请务必对每个shape调用该方法，以保证TriangleMesh构建内部加速结构
        //* 由于使用embree时，TriangleMesh::getAABB不会被调用，因此出于性能考虑我们不在TriangleMesh
        //* 的构造阶段计算其AABB，因此当我们将TriangleMesh的AABB计算放在TriangleMesh::initInternalAcceleration中
        //* 所以请确保在调用TriangleMesh::getAABB之前先调用TriangleMesh::initInternalAcceleration
        shape->initInternalAcceleration();
        boundingBox.Expand(shape->getAABB());
    }
    //* todo 完成BVH构建
  std::vector<int> primIdxBuffer(shapes.size());//所有的shape
  std::iota(primIdxBuffer.begin(), primIdxBuffer.end(), 0);

  std::vector<std::shared_ptr< Shape>>  orderedShapes;//存储构建好的shapes顺序
  orderedShapes.clear();

  root = recursiveBuild(boundingBox, primIdxBuffer,0,primIdxBuffer.size()-1,orderedShapes,0);//从根节点开始

  //root = recursiveBuild(boundingBox, primIdxBuffer,orderedShapes,0);//从根节点开始递归构建树
  shapes=orderedShapes;//重制shapes

}

//正确递归方式
//[l,r]范围内
BVH::BVHNode * BVH::recursiveBuild(const AABB& aabb,std::vector<int>& primIdxBuffer , int l, int r,std::v
{
    if ( r-l+1 <= bvhLeafMaxSize ||depth>maxDepth)
    {
        BVHNode* node=new BVHNode(aabb);
        node->firstShapeOffset = orderedShapes.size();
        node->nShape =  r-l+1;
        for ( int i = l ; i <= r ; ++ i )
        {
            orderedShapes.push_back(shapes[primIdxBuffer[i]]);
        }

        return node;
    }

    //计算图元**质心**的边界!!选择分割的坐标轴
    AABB b;
    for (int i = l; i <= r; ++i)
    b.Expand( shapes[primIdxBuffer[i]]->getAABB().Center());
    Point3f A(b.pMin);
    Point3f B(b.pMax);
    float dx=(B[0]-A[0]);
    float dy=(B[1]-A[1]);
    float dz=(B[2]-A[2]);
    AABB Box1;//负半边盒子
    AABB Box2;//正半边盒子

    BVHNode* node = new BVHNode(aabb);//创建一个非叶子节点

    if(dx>=dy&&dx>=dz) node->splitAxis=0;//按照x轴划分
    else if(dy>=dx&&dy>=dz) node->splitAxis=1;//按照y轴划分
    else node->splitAxis=2;//按照z轴划分

    if(A[node->splitAxis]==B[node->splitAxis]) //质心相等
    {
        printf("here/n");
        BVHNode* node=new BVHNode(aabb);
        node->firstShapeOffset = orderedShapes.size();
        node->nShape =  r-l+1;
        for ( int i = l ; i <= r ; ++ i )
        {
            orderedShapes.push_back(shapes[primIdxBuffer[i]]);
        }

        return node;
    }

    QuickSort(primIdxBuffer,l,r,node->splitAxis);//排序

    int mid=(r+l)/2;
    for(int i=l; i<=mid; ++i)
    {
        Box1.Expand(shapes[primIdxBuffer[i]]->getAABB());
    }
    for(int i=mid+1; i<=r; ++i)
    {
        Box2.Expand(shapes[primIdxBuffer[i]]->getAABB());
    }

    node->left=recursiveBuild(Box1,primIdxBuffer,l,mid,orderedShapes,depth+1);
    node->right=recursiveBuild(Box2,primIdxBuffer,mid+1,r,orderedShapes,depth+1);


    return node;
```

```cpp
//实现排序
void BVH::QuickSort(std::vector<int>& primIdxBuffer, int low, int high,int split) //快排母函数
{
    if (low < high) {
        int pivot = Paritition1(primIdxBuffer, low, high,split);
        QuickSort(primIdxBuffer, low, pivot - 1,split);
        QuickSort(primIdxBuffer, pivot + 1, high,split);
    }
}

int BVH::Paritition1(std::vector<int>& primIdxBuffer, int low, int high,int split)
{
    int pivot = primIdxBuffer[low];
    while (low < high)
    {
        while (low < high && shapes[primIdxBuffer[high]]->getAABB().Center()[split] >= shapes[pivot]->get
        {
            --high;
        }
        primIdxBuffer[low] = primIdxBuffer[high];
        while (low < high && shapes[primIdxBuffer[low]]->getAABB().Center()[split] <= shapes[pivot]->getA
            ++low;
        }
        primIdxBuffer[high] = primIdxBuffer[low];
    }
    primIdxBuffer[low] = pivot;
    return low;
}
```

2.3 BVH 求交

• 递归，通过判断节点的包围盒与光线是否相交来剪枝进行加速

• 对于非叶子节点，首先判断当前空间是否和光线相交，如果不相交可以直接返回, 否则递归调用左右子节点的求交函数

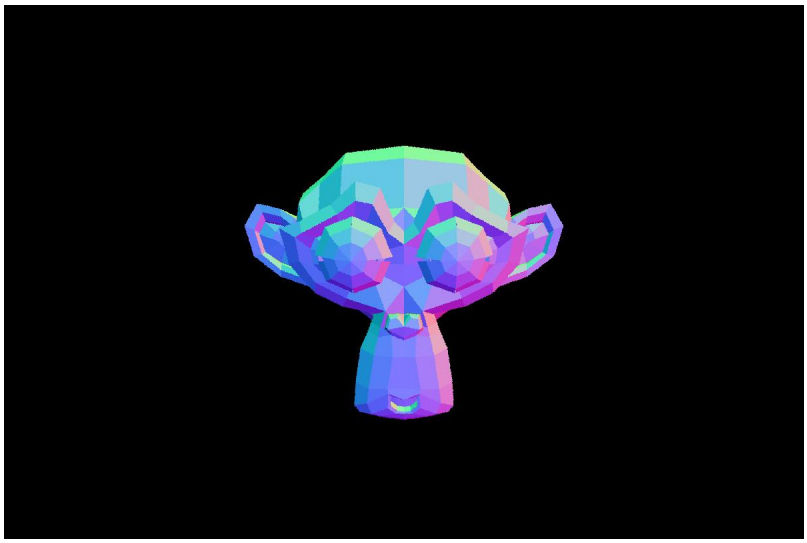
• 根据分割轴与光线在这个轴上的方向选择调用顺序，进行加速

• 对于叶子节点，遍历叶子节点下的所有物体进行求交。

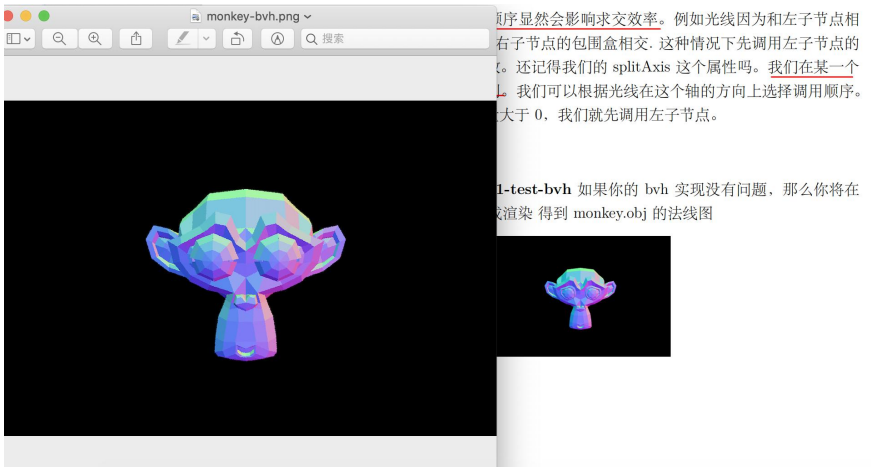
```cpp
bool BVH::rayIntersect(Ray &ray, int *geomID, int *primID, float *u, float *v) const {
    //* todo 完成BVH求交

    return this->recursiveRayIntersect(root,ray,geomID,primID,u,v);

}

bool BVH::recursiveRayIntersect(BVHNode *node, Ray &ray, int *geomID, int *primID, float *u, float *v) co
{
    if(node==nullptr) return false;
    if(!node->boundingBox.RayIntersect(ray,nullptr,nullptr)) return false;

    //相交
    //对于非叶子节点
    if(node->nShape==-1)
    {
        if(ray.direction[node->splitAxis]>0)//先调用左子节点
        {
            if(recursiveRayIntersect(node->left,ray,geomID,primID,u,v)) return true;
            else return recursiveRayIntersect(node->right,ray,geomID,primID,u,v);
        }

        else//先调用右子节点
        {
            if(recursiveRayIntersect(node->right,ray,geomID,primID,u,v)) return true;
            else return recursiveRayIntersect(node->left,ray,geomID,primID,u,v);
        }

    }

    else if(node->nShape!=0)//叶子节点
    {
        bool flag=false;
        for(int i=0; i<node->nShape; ++i)
        {
            bool re=shapes[node->firstShapeOffset+i]->rayIntersectShape(ray,primID,u,v);
            flag|=re;
            if(re==true)
            {
                *geomID=shapes[node->firstShapeOffset+i]->geometryID;
                //break;
            }
        }
        return flag;
    }



    return false;

}
```

1.4 结果验证

得到渲染结果:



与 lab 文档中对比，一致。



用时在 5s 以内。

```
pro@prodeMacBook-Pro-2 bin % ./Moer /Users/pro/Desktop/Rendering_labs/Moer-lite/
examples/lab1-test/lab1-test-bvh
Using acceleration type bvh
100% [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||]
Rendering costs 3.24s
```

三、 总结

1. 遇到的问题

（1）坐标的转换:

在 transform 类中，使用 inverseRay 方法可以将光线变换到局部空间。toWorld 方法将坐标转换到世界空间。需要注意的是每个 shape 都有自己的 transform 成员变量，不需要新建，新构造的 transform 对象所有变换矩阵都是恒等变换。

（2）法线归一化问题

圆环和圆柱的交点处归一化法线都可以直接写出来，但是圆锥的交点处法线需要计算，且最终需要归一化!

（3）确保是 release 版本

debug 版本性能较差，使用 ccmake 修改生成版本，确保使用 cmake 生成的是 release 版

本。

四、 参考博客

[1] 光线追踪渲染实战（二）：BVH 加速遍历结构
[2] BVH 树的构建