南京大学

NANJING UNIVERSITY

图形绘制技术 Lab1 实验报告

课程： 图形绘制技术

姓名： 王思远

学号： 201220114

专业： 计算机科学与技术

时间： 2023.4

目录

1 实验内容

1.1 圆环与光线的求交

求交逻辑：

圆环与光线的求交逻辑非常简单，我们只需要把光线变换到局部坐标系下，计算它与 xoy 平面的交点就可以了，由于圆环的圆心位于原点，并且圆环与 xoy 平面平行，所以只要判断交点与圆心的距离是否在圆环内径和外径之间，就可以判断光线是否有与圆环相交的可能。

```
1  //* 1.光线变换到局部空间
2  auto local_ray = transform.inverseRay(ray);
3  //* 2.判断局部光线的方向在z轴分量是否为0
4  if(local_ray.direction[2]==0)
5      return false;
6  //* 3.计算光线和平面交点
7  auto t=(0-local_ray.origin[2])/(local_ray.direction[2]);
8  //* 4.检验交点是否在圆环内
9  //* 4.1 检验交点是否在光线范围内
10 if(t<local_ray.tNear||t>local_ray.tFar)
11     return false;
12 //* 4.2 检验交点是否在圆环半径范围内
13 auto intersected_point=local_ray.at(t);
14 //* 距离就是开平方根(x^2+y^2)
15 auto distance_to_origin=(float)sqrt(pow(intersected_point[0],2)
       +pow(intersected_point[1],2));
16 if(distance_to_origin<innerRadius||distance_to_origin>radius)
17     return false;
```

当然，上述代码仅仅是对于一个完整的圆环的求交过程，我们渲染的圆环可能是残缺的，因此我们要对交点对应的角度进行判断，看看交点是否落在圆环部分存在的角度，而不是残缺的位置。这一点可以通过 x 坐标与

y 坐标计算象限角度的弧度来得到。当我们确认相交后，更新光线的 tFar，来减少多余的计算，并根据公式计算得出 u，v 坐标即可。

```
//* 4.3 检验交点是否在圆环角度范围内
float phi;
if(intersected_point[1]>=0)
    phi=atan2(intersected_point[1],intersected_point[0]);
else
    phi= 2*PI + atan2(intersected_point[1], intersected_point
        [0]);
if(phi>phiMax)
    return false;
//* 5.更新ray的tFar,减少光线和其他物体的相交计算次数
ray.tFar = t;
//* 6.填充u,v坐标
*u=phi/phiMax;
*v=(distance_to_origin-innerRadius)/(radius-innerRadius);
//* Write your code here.
```

运行结果：

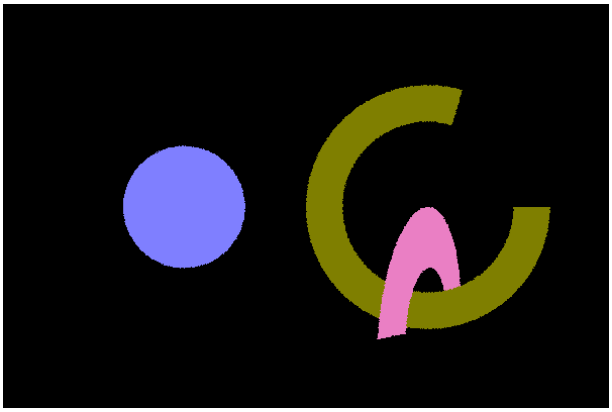

图 1: 补全圆环求交代码后所得图片

1.2 圆柱与光线的求交

求交逻辑：

圆柱与光线求交的逻辑也并不复杂，第一步首先要将光线变换到圆柱局部空间，便于计算，接下来，手册已经给出了交点方程，这是一个二次方程，我们只需要化简得到对应的方程 ABC 系数，代入二次方程求解函数内求解即可得到交点。

```
1  //* 1.光线变换到局部空间
2  auto local_ray=transform.inverseRay(ray);
3  //* 2.联立方程求解
4  auto A=(float)pow(local_ray.direction[0],2)+(float)pow(
       local_ray.direction[1],2);
5  auto B=2*(local_ray.origin[0]*local_ray.direction[0]+local_ray.
       origin[1]*local_ray.direction[1]);
6  auto C=(float)pow(local_ray.origin[0],2)+(float)pow(local_ray.
       origin[1],2)-(float)pow(radius,2);
7  float t0,t1;
8  //* 若无解，说明没有交点
9  if(!Quadratic(A,B,C,&t0,&t1))
10     return false;
```

但是，这里有一个小小的细节上的问题，二次方程的解可能有 1 个 2 个或者 0 个，对于 1 个或 0 个的解我们很好处理，但是对于两个解我们如何判断哪一个交点是我们需要的呢？仔细观察求解函数我们会发现，函数返回的解是有序的，t0 一定是最小的光线到达时间，而对于渲染求交，我们也只需要光线第一次到达物体表面的交点。

那么是不是只取 t0 就可以了呢？不是的，即使 t0 是交点方程的一个解，但这个所谓的交点其实只是在一个无限高的完整圆柱上，在我们的渲染器中，圆柱并不是无限高的，而且有残缺，所以我们的目标应该是寻找符合条件下到达时间最小的点，也就是说，如果我们发现 t0 是符合条件的，那

么答案就是 t0，若 t0 不符合条件，我们还要看 t1。

这个流程并不难实现，我们只要定义一系列检查，寻找第一个通过检查的点即可。

```
//* 3.检验交点是否在圆柱范围内
float final_t,final_phi;
bool found= false;
for(auto t: {t0,t1}){
    //* 3.1 检验交点是否在光线范围内
    if(t<local_ray.tNear||t>local_ray.tFar)
        continue;
    //* 3.2 检验交点是否在圆柱高度范围内
    auto p=local_ray.at(t);
    if(p[2]<0||p[2]>height)
        continue;
    //* 3.3 检验交点与圆心夹角是否符合要求
    float phi;
    if(p[1]>=0)
        phi=atan2(p[1],p[0]);
    else
        phi= 2*PI + atan2(p[1], p[0]);
    if(phi>phiMax)
        continue;
    //* 全部符合
    final_t=t,final_phi=phi,found= true;
    break;
}
if(!found)
    return false;
```

最后，更新光线的 tFar，计算填充 uv 坐标，返回结果。

```
//* 4.更新ray的tFar,减少光线和其他物体的相交计算次数
ray.tFar=final_t;
//* 5.填充u,v坐标
auto p=local_ray.at(final_t);
*u=final_phi/phiMax;
*v=p[2]/height;
return true;
```

运行结果：

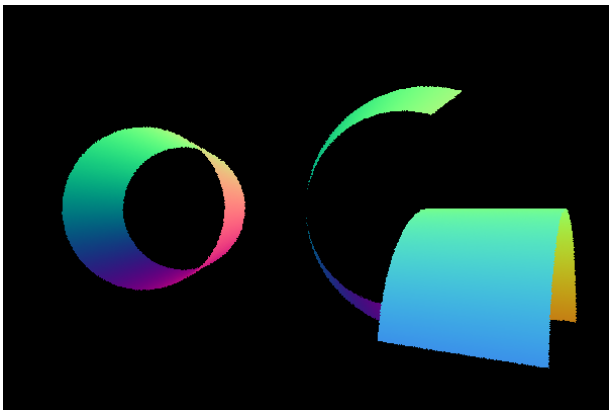

图 2: 补全圆柱求交代码后所得图片

1.3 圆锥与光线的求交

求交逻辑：

圆锥求交的思想与圆柱求交类似，同样是解方程，检查交点，计算 uv 坐标，这里就不再赘述了。我认为比较值得一提的是圆锥的求法线问题，圆锥的交点处法线我们可以直接根据交点来求得，具体分析可以参考下图：

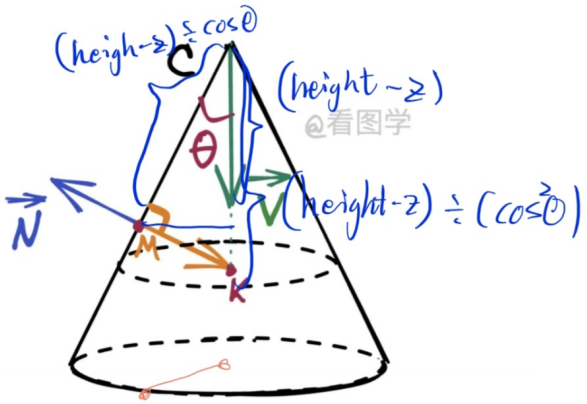

图 3: 求出 K 点坐标，KM 向量就是法线方向

```
auto phi=phiMax*u;
auto M_radius=radius*(1-v);
auto x=M_radius*cos(phi);
auto y=M_radius* sin(phi);
auto z=height*v;
auto M=Point3f{x,y,z};
intersection->position=transform.toWorld(M);
auto K=Point3f{0,0,height-(height-z)/(float)pow(cosTheta,2)};
auto MK=M-K;
intersection->normal=transform.toWorld(normalize(MK));
```

运行结果：

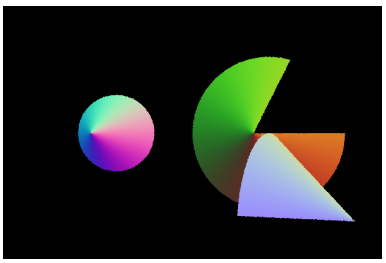

图 4: 补全圆锥求交代码后所得图片

2 Bonus 部分

2.1 加速思路

本次 Bonus 部分实现的加速结构是八叉树，八叉树的原理其实很简单，就是把空间均匀地分成八份，建立一个树形的索引，以此达到一个求交加速的效果。

八叉树的节点是有分工的，我们的求交算法对于叶子节点和枝干节点是有去奋斗，不能一概而论，对于枝干节点，我们要与枝干节点的子节点的包围盒求交，并决定是否进入其下的某个子节点。对于叶子节点，我们要遍历叶子节点内存储的所有图元求交。

如何判断叶节点和枝节点呢？我们可以利用节点结构中图元数量这个成员变量，当图元数量为-1 时，该节点为枝干节点，否则，该节点为叶子节点。

如何构建一颗八叉树？同样很简单，下面给出一个详细注释的代码：

```
1  //* 递归构建八叉树
2  //* 节点数足够少
3  if (primIdxBuffer.size() <= ocLeafMaxSize)
4      return std::make_shared<OctreeNode>(primIdxBuffer, aabb);
5  //* 划分空间
6  auto subBoxes = getSubBoxes(aabb);
7  //* 划分每个空间的primIdx数组
8  std::vector<std::vector<int>> subBuffers(8);
9  //* 记录空白空间的数量
10 int empty = 0;
11 //* 对于每一个划分出的空间
12 for (int i = 0; i < 8; ++i) {
13     for (auto primIdx: primIdxBuffer) {
14         // 如果任意图元与当前空间相交，就把它加入当前空间的primIdx
           数组中
```

```
            if (shapes[primIdx]->getAABB().Overlap(subBoxes.at(i)))
                subBuffers.at(i).push_back(primIdx);
        }
        if (subBuffers.at(i).empty())//如果当前空间的primIdx数组为
            空，说明该空间没有与任何图元相交
            ++empty;
    }
    //* 新建节点
    auto node = std::make_shared<OctreeNode>();
    node->boundingBox = aabb;
    //* 对每个子空间递归建树
    for (int i = 0; i < 8; ++i) {
        node->subNodes[i] = recursiveBuild(subBoxes[i], subBuffers[
            i]);
    }
    return node;
```

接下来只要根据八叉树求交就可以了。思路同样很简单，我们可以递归地计算，对于每一个节点：

- 若节点的包围盒不与光线相交，直接结束
- 如果这个节点是一个图元数量大于 0 的叶节点，遍历所有图元，求交点。
- 如果这个节点是一个枝节点，递归访问其所有子树。

我们可以看到，所谓的加速过程其实就是一种剪枝，下面给出一个用队列优化的求交代码：

```
    bool result = false;
    std::queue<std::shared_ptr<OctreeNode>> job{};
    job.push(root);
    while (!job.empty()) {
```

```cpp
        auto currentNode = job.front();
        job.pop();
        if (currentNode->boundingBox.RayIntersect(ray)) {
            if (currentNode->primCount == -1) {
                //primCount为-1，必为树枝，把树枝下的子节点都加入工作集中
                for (const auto &subNode: currentNode->subNodes) {
                    job.push(subNode);
                }
            } else if (currentNode->primCount > 0) {
                //primCount大于0，说明为叶子节点且存在相交可能，进行求交判断
                for (int i = 0; i < currentNode->primCount; ++i) {
                    int index = currentNode->primIdxBuffer[i];
                    bool hit = shapes[index]->rayIntersectShape(ray,
                        primID, u, v);
                    if (hit)
                        *geomID = index;
                    result = hit || result;
                }
            }
        }
    }
    return result;
```

2.2 加速效果对比

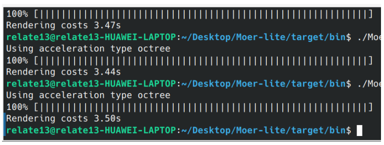

图 5: 使用八叉树加速，图片生成需要 3 秒左右

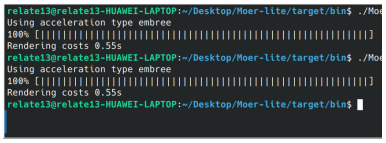

图 6: 使用 embree 加速，仅需半秒

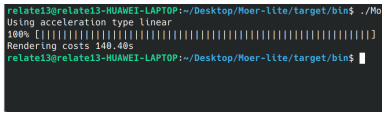

图 7: 线性遍历则需要 120 秒

3 吐槽

- 关于加速性能,不同的操作系统与硬件有很大的不同,使用 windows+AMD CPU 运行八叉树加速，生成图片需要 15s 以上，而使用 Linux+Intel CPU 运行同样的代码，只需要 2-3s。
- 手册上的伪代码描述的八叉树的构建逻辑与文件注释中描述的特殊情况貌似存在冲突，如果遵循手册的伪代码逻辑，当图元数目小于最大容量时就生成叶子节点，则文件注释中描述的特殊情况“当节点的某个子包围盒和当前节点所有物体都相交时，将当前节点作为叶子节点”永远不会出现，因为当前节点作为叶子节点的前提就是图元数目小于最大容量。