**Intersections between Ray and Shapes**

练孙鸿

1. **Logical Shape Interface**

Noise3D在引入GI(Global Illumination)之前都是用三角形网格(Triangular Mesh)来表示物体的。现在要在原来的场景管理设施上新增Path Tracing Render System，为了能更快地实现运行效率较高的demo，我们还要实现一些逻辑的、解析的(analytic)基本形状，如AABB(Axis-Aligned Bounding Box)、球体、圆柱等。他们需要一些通用的接口如：

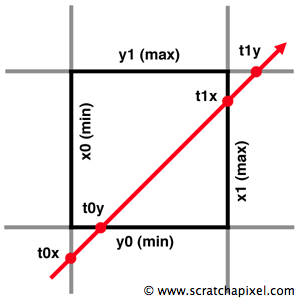
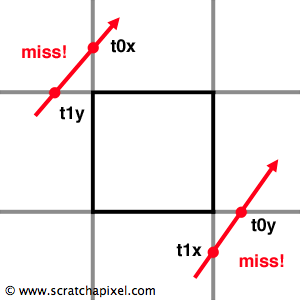
|  |
| --- |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*                      Logical Geometry         geometry representation that is not based on polygon.         instead, it's based on analytic representation.         e.g., you'll need a center and radius to represent a sphere  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma once  namespace Noise3D  {      class /\*\_declspec(dllexport)\*/ ILogicalShape:          public ISceneObject      {      public:          //ISceneObject::          virtual NOISE\_SCENE\_OBJECT\_TYPE GetObjectType() override = 0 ;          //ISceneObject::          virtual N\_AABB GetLocalAABB() override= 0;          //ISceneObject::          virtual N\_AABB ComputeWorldAABB\_Accurate() override = 0;          //Compute Area for the shape(might be useful for area lighting)          virtual float ComputeArea() =0 ;          //determine if point is inside this shape          virtual bool IsPointInside(NVECTOR3 p) = 0;          //..          void SetMaterial(Material\* pMat)          {              m\_pMaterial = pMat;          };      protected:          ILogicalShape() {};          ~ILogicalShape() {};          Material\* m\_pMaterial;      };  } |

当然我也不知道这些接口够不够用….先这么写着先。

在<http://www.realtimerendering.com/intersections.html> 上，有各种基本几何形状之间的相交算法的参考文献、资料，可以留意一下。

1. **Ray-AABB Intersection**

AABB全称Axis Aligned Bounding Box。由于它每个面都平行于某个坐标平面的优良特性，它和其他基本的几何对象的碰撞是比较快速的。而且作为碰撞检测与加速结构BVH中执行最多的操作，Ray-AABB的求交也必然要求高效。有一种高效叫做[Kay86] “slab method”，其中slab有“挡板”的意思。也就是说，在n维空间里面，AABB就是个平行于坐标超平面(hyper-plane)的超平面围成的区域，具体到三维欧氏空间里面，AABB就是三对axis-aligned planes围成的部分，每对平面分别平行于xy、xz、yz坐标平面。每一个平面都是一个slab。

图：2D情况下ray-AABB intersection

在三维欧氏空间里，为了判断射线 是否跟一个AABB相交，我们要计算射线跟三对平面的交点对应射线参数的区间，三对平面分别是，那么相交得到的三个射线位置参数的区间。然后我们要判断这个三个区间是否有重合的部分。取三个区间的交集，如果交集不为空，我们就得到了射线与AABB相交的部分，以射线参数的区间表示。

射线与平面相交的射线参数*t*求解也很简单，以平行于YZ平面的一对slabs为例：

那么同理。

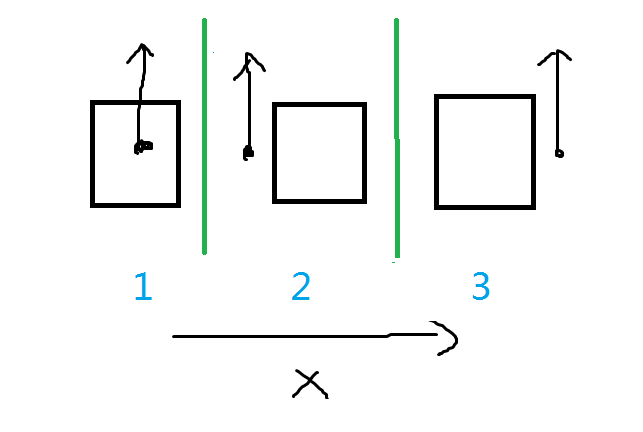
Ray-AABB求交的C++代码如下：

|  |
| --- |
| bool Noise3D::CollisionTestor::IntersectRayAabb(const N\_Ray & ray,  const N\_AABB & aabb)  {      //if a ray hit AABB, then the ray param t of x/y/z will overlap      //the intersection of those 3 t-value interval is the final intersection ray interval.      //if the final interval is invalid (say left is greater than right),      //then ray-AABB intersection failed/missed.      //for more info, refer to Peter Shirley's 《Ray Tracing : The Next Week》      //or pbrt-v3: https://github.com/mmp/pbrt-v3 /src/core/geometry.h Bounds3<T>::IntersectP      float t\_resultMin = 0;      float t\_resultMax = ray.t\_max;      //put aabb's component into an array to use a 'for'      float rayOrigin[3] = { ray.origin.x, ray.origin.y, ray.origin.z };      float rayDir[3] = { ray.dir.x, ray.dir.y, ray.dir.z };      float slab\_min[3] = { aabb.min.x, aabb.min.y, aabb.min.z };      float slab\_max[3] = { aabb.max.x, aabb.max.y, aabb.max.z };      //mark to test near and far's hit state      bool isNearHit = false;      bool isFarHit = false;      //calculate a pair of slabs for an interval in each loop      for (int i = 0; i < 3; ++i)      {          //note the corner case ray[i]==0, division result will be infinity, which is still well-defined by IEEE          //and fortunately, the [t\_near, t\_far] result is still correct          float rayDirReciprocal = 1.0f / rayDir[i];          float t\_near = (slab\_min[i] - rayOrigin[i]) \* rayDirReciprocal;          float t\_far = (slab\_max[i] - rayOrigin[i]) \* rayDirReciprocal;          // Update ray's parameter t's interval from slab intersection          if (t\_near > t\_far) std::swap(t\_near, t\_far);          // Update \_tFar\_ to ensure robust ray--bounds intersection(pbrt-v3, 'rounding errors')          t\_far \*= (1 + 2 \* mFunc\_Gamma(3));          //try to narrow the t value interval          if (t\_near > t\_resultMin) { t\_resultMin = t\_near; isNearHit = true; }          if (t\_far < t\_resultMax) { t\_resultMax = t\_far; isFarHit = true; }          //validate t interval.          if (t\_resultMin >= t\_resultMax)          {              //if not valid, then the ray miss the AABB              return false;          }      }      //t\_max rejection      if (t\_resultMax > ray.t\_max)isFarHit = false;      //2 intersection points at most, but if there is none:      if (!isNearHit && !isFarHit)return false;      return true;  } |

可以看到，Slab method简洁而高效。但有一个corner case需要在这提一下，就是当射线方向的某个分量为0时，会出现除0 的现象，这在数学中是无意义的。

|  |
| --- |
| float rayDirReciprocal = 1.0f / rayDir[i]; |

但在c++编程实现中，即使rayDir[i]==0，rayDirReciprocal也是有定义的。IEEE对[3]非零浮点数(floating point number)除零有明确定义，结果会是正无穷或者负无穷，正负号与分子一致，且其二进制表示在IEEE 754-2008r里面有明确定义。正无穷在c++标准里面的表示可以用std::numeric\_limits<float>::infinity()取得。如此一来，在rayDir[i]==0时：



图：麻烦写paper的时候重新p一下图

下面是当ray.origin.x==0时，几种除零时的状况：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 情况序号 | 实际是否可能相交 |  | 是否更新区间 | 相交判定 |
| 1 | 是 |  | 否 | 有待确定，还需运行 |
| 2 | 否 |  | 是，result\_min=+infinity | 不相交 |
| 3 | 否 |  | 是，result\_max=-infinity | 不相交 |

综上，corner case也被妥当地处理了。

1. **Ray-Sphere Intersection**

射线和球体求交第一件事就是将他们都转换到object space。在object space里，球的圆心在原点，那么球的方程是：

与射线联立得：

变式与展开成一元二次方程的形式后，有：

那么如果二次方程的解：

是两个实根 ，则射线与球有一个或两个交点，交点位置就是。C++实现代码：

|  |
| --- |
| bool Noise3D::CollisionTestor::IntersectRaySphere(const N\_Ray & ray,  LogicalSphere\* pSphere, N\_RayHitResult & outHitRes)  {      if (pSphere == nullptr)      {          ERROR\_MSG("CollisionTestor: object is nullptr.");          return false;      }      //convert ray to model space (but first scene object must attach to scene node)      N\_Ray localRay;      RayIntersectionTransformHelper helper;      if (!helper.Ray\_WorldToModel(ray, pSphere, localRay))return false;      float r = pSphere->GetRadius();      NVECTOR3& D = localRay.dir;      NVECTOR3& O = localRay.origin;      float A = D.Dot(D);      float B = 2.0f \* D.Dot(O);      float C = O.Dot(O) - r\*r;      //determinant of quadratic equation      float det = B \* B - 4.0f \* A\*C;      if (det < 0.0001f || D == NVECTOR3::Zero)      {          //0(miss) or 1(tangent) solution, or A==0(then the equation even not exist)          return false;      }      else      {          float sqrtDet = std::sqrtf(det);          //local space hit info 1          N\_RayHitInfo hitInfo1;          float t1 = (-B + sqrtDet)/ (2.0f \* A);          hitInfo1.t = t1;          hitInfo1.pos = ray.Eval(t1);          hitInfo1.normal = hitInfo1.pos;          hitInfo1.normal.Normalize();          outHitRes.hitList.push\_back(hitInfo1);          //local space hit info 2          N\_RayHitInfo hitInfo2;          float t2 = (-B - sqrtDet) / (2.0f \* A);          hitInfo2.t = t2;          hitInfo2.pos = ray.Eval(t2);          hitInfo2.normal = hitInfo2.pos;          hitInfo2.normal.Normalize();          outHitRes.hitList.push\_back(hitInfo2);      }      //transform the result hit back to world space      helper.HitResult\_ModelToWorld(outHitRes);      return true;  } |

交点的其他信息参考pbrt 3rd edition 3.2。

1. **Ray-Box Intersection**

转到Object Space，然后用Ray-AABB

1. **Ray-Mesh Intersection**

这个问题等价于Ray-Triangle Intersection，下面简单介绍几种解决方案，并详细介绍准备在Noise3D中实现的方案。

Pbrt 3rd edtion的解决方案是先用一个列主向量(column-major)的仿射变换(affine transform) 把三角形三个顶点和射线都变换到一个“求交空间”里面。其中把射线起点对齐到原点；**是置**换(permutation)，用于置换射线的分量，使得向量分量置换后射线的z分量绝对值最大；是剪切(shear) 。在求交空间里，射线与z轴对齐，也就是。然后三维空间的求交问题就退化成了：投影在2D平面上的三角形是否包含坐标系原点。解决这个问题只需要求出坐标系原点在2D三角形的重心坐标(barycentric coordinate)，并判断其重心坐标是否满足特定条件。（如果要求得交点的世界坐标，那还得把求交空间的交点坐标变换回去）

另一种方法，则是在世界空间或者模型空间中用纯粹的解析几何求出射线所在直线与三角形所在平面的交点，并计算交点的三角形重心坐标。

[4]提出一种用非常简单快速的代数方法求解射线与三角形的交点，Noise3D打算用这种方法来实现。一个在三角形里面的点可表示为：

其中。值得一提的是，这里的参数坐标可以在光栅化渲染进行顶点属性插值的时候使用。

我们现在有射线，求射线与三角形的交点，即求线性方程组的解：

我们采用列主向量的矩阵，则变形并整理得三阶线性方程组：

设以使下面的推导更加简洁。使用克拉默法则(Cramer’s Rule)求方程组的解：

又因为行列式可以写成三重积(Triple Product)的形式：

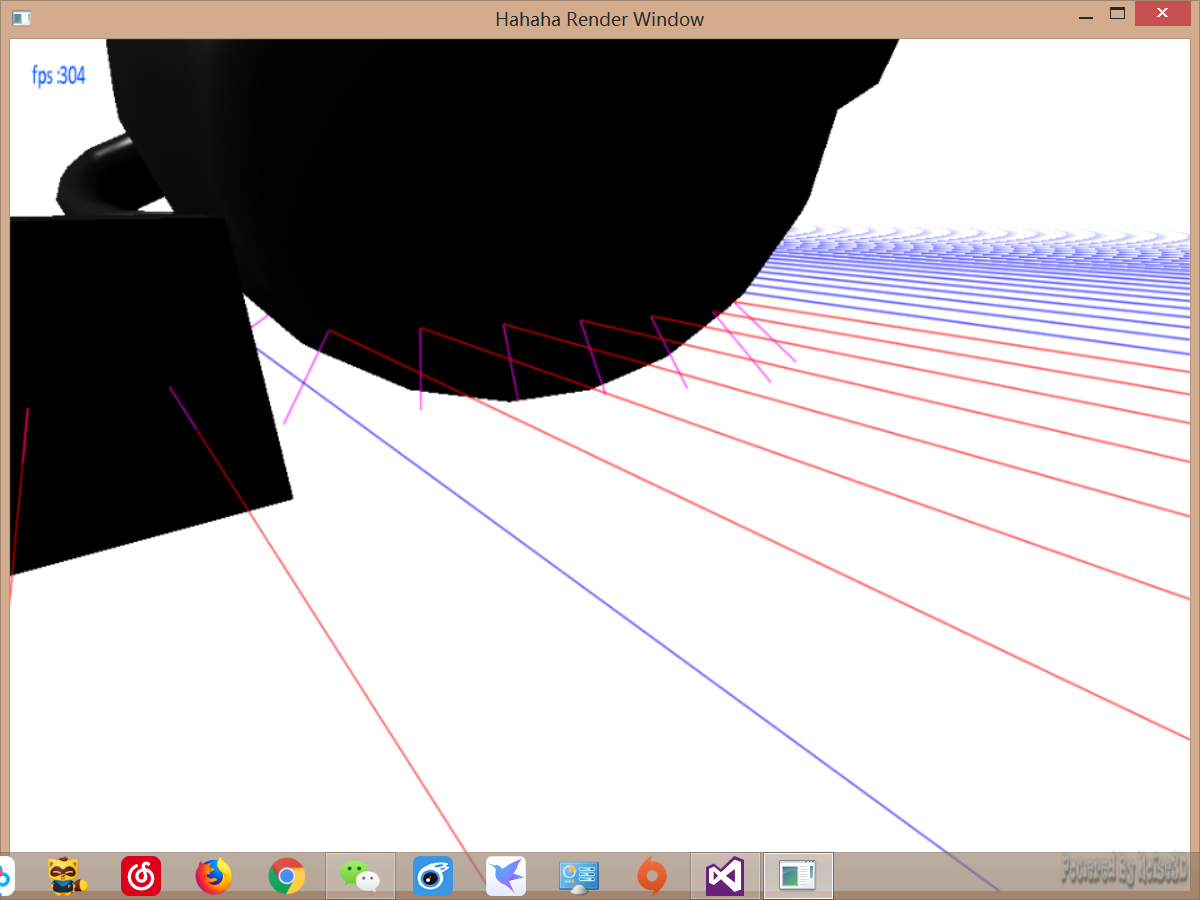
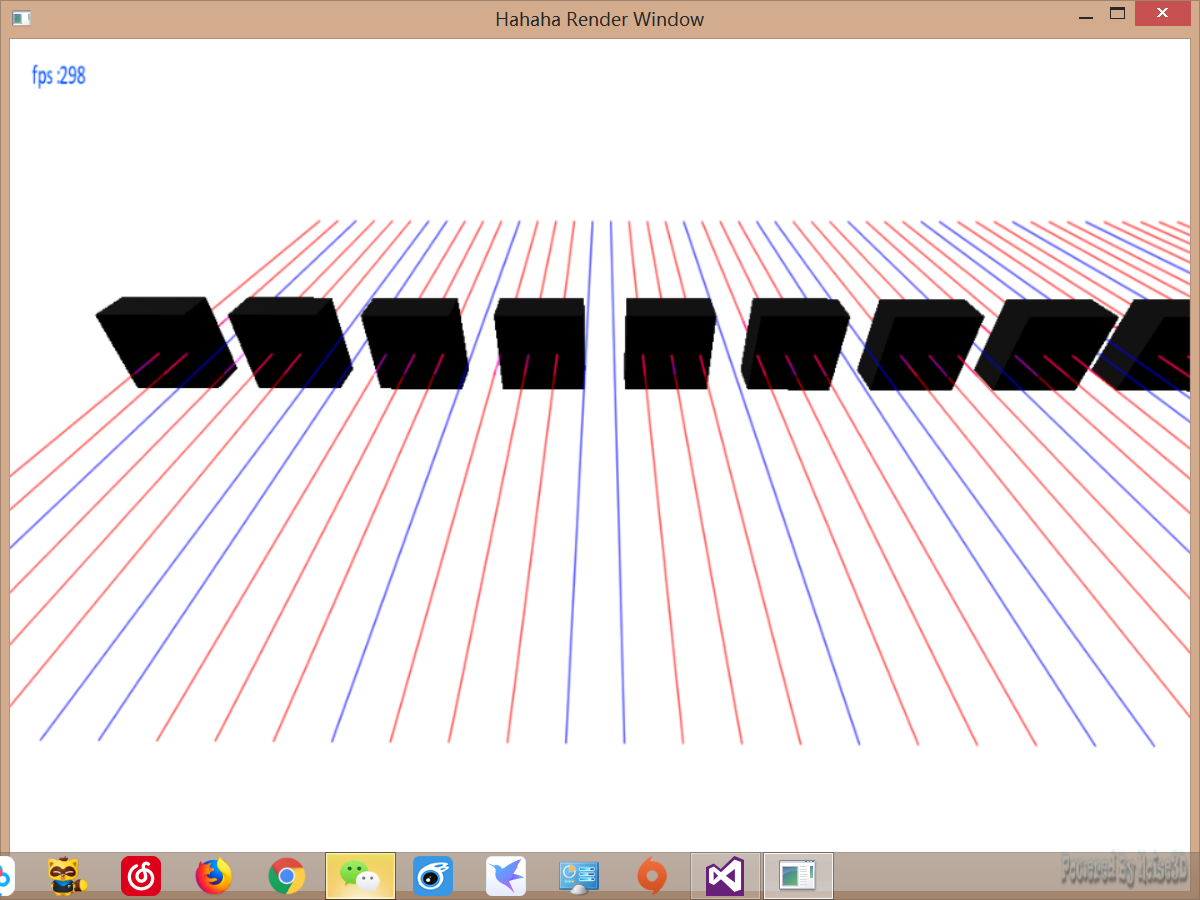
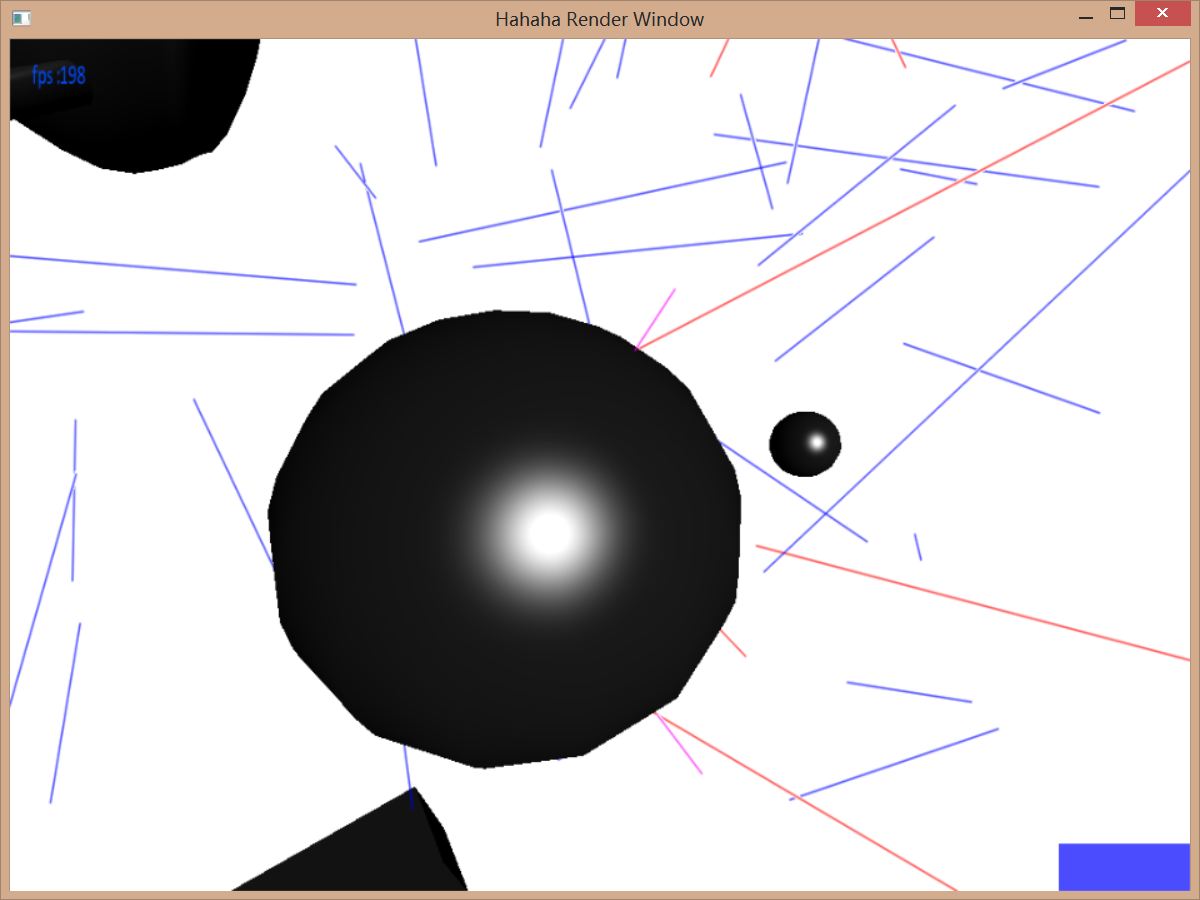
行列式也可以这样展开成三重积：

然后的解可以写成点积与叉积的运算。出于优化的目的，减少多余的计算，[4]把行列式写成三重积的时候，显然进行了精心构造：

设，则：

C++实现非常简洁（虽然并不直观）：

|  |
| --- |
| bool Noise3D::CollisionTestor::IntersectRayTriangle(const N\_Ray & ray,  NVECTOR3 v0, NVECTOR3 v1, NVECTOR3 v2, N\_RayHitInfo & outHitInfo)  {      //[Reference for the implementation]      //Tomas Möller, Trumbore B . Fast, Minimum Storage Ray-Triangle Intersection[J]. 2005.      //the method can be describe as follow:      //\*\*Triangle(V0, V1, V2)      //\*\*Ray p=O+tD (O for origin, D for direction)      //\*\*Point in Triangle R(u,v) = (1-u-v)V0 + uV1 + vV2      //then use CRAMER's RULE to solve the linear equation system for the unknown (t,u,v):      //    O+tD = (1-u-v)V0 + uV1 + vV2    \*\*\*\*\*\*(u >0, v>0, u+v<=1)      //the code here won't be intuitive, because it's mainly algebraic inductions.      NVECTOR3 E1 = v1 - v0;      NVECTOR3 E2 = v2 - v0;      NVECTOR3 P = ray.dir.Cross(E2);//P = D x E2      float det = P.Dot(E1);//3x3 matrix's determinant= (d x e2) dot e1 = P dot e1      if (std::abs(det) <= std::numeric\_limits<float>::epsilon())return false;      float invDet = 1.0f / det;//calculate once      NVECTOR3 M = ray.origin - v0;// M = O - V0      float u = invDet \* M.Dot(P);// invDet \* (M dot P)      if (u < 0.0f || u> 1.0f)return false;//early return to avoid further computation      NVECTOR3 Q = M.Cross(E1);//Q = M x E1      float v = invDet \* ray.dir.Dot(Q);//v = invDet \* (D dot Q)      if (v < 0.0f || u + v>1.0f)return false;//early return to avoid further computation      float result\_t = invDet \* E2.Dot(Q);//t = invDet \* (E2 dot Q)      //t\_max rejection      if (result\_t > ray.t\_max)return false;      outHitInfo.t = result\_t;      outHitInfo.pos = ray.Eval(result\_t);      outHitInfo.normal = E1.Cross(E2);      return true;  } |



Reference

[1] Shirley P., Ray Tracing: The Next Week[DB/OL] <http://www.realtimerendering.com/raytracing/>, 2019-3

[2] Kay T. L., Kajiya J. T., Ray tracing complex scenes.[C] Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques, ACM, New York, NY, USA, SIGGRAPH ’86, 1986, 269-278.69,75,76

[3] IEEE 754-2008, IEEE Standard for Binary Floating Point Arithmetic.[S] IEEE, 29 Aug. 2008

[4] Tomas Möller, Trumbore B . Fast, Minimum Storage Ray-Triangle Intersection[J]. 2005.