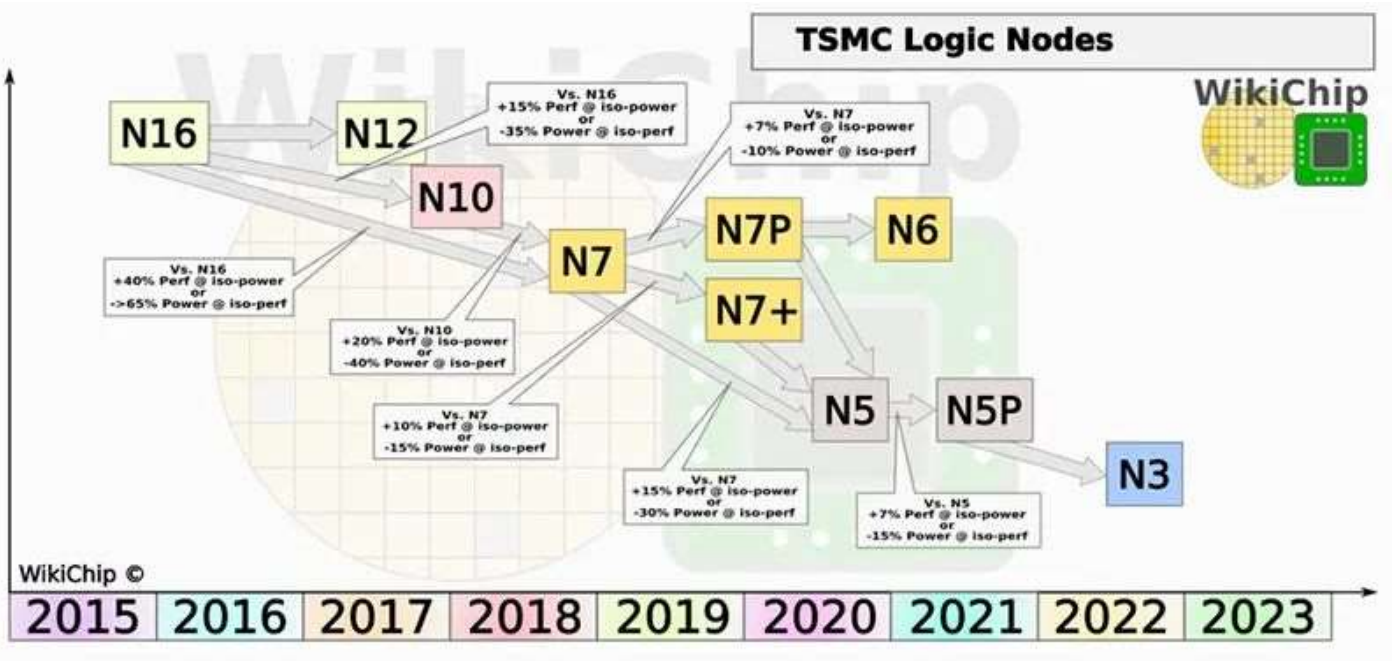
Hardware Accelerated Ray Tracing 硬件加速 光线追踪的原理、设计及应用

汇报人：余斐然

2023/4/17

背景

随着芯片工艺制程的推进，以及芯片设计的进步，
使得硬件加速的光线追踪达到实时效果成为可能。



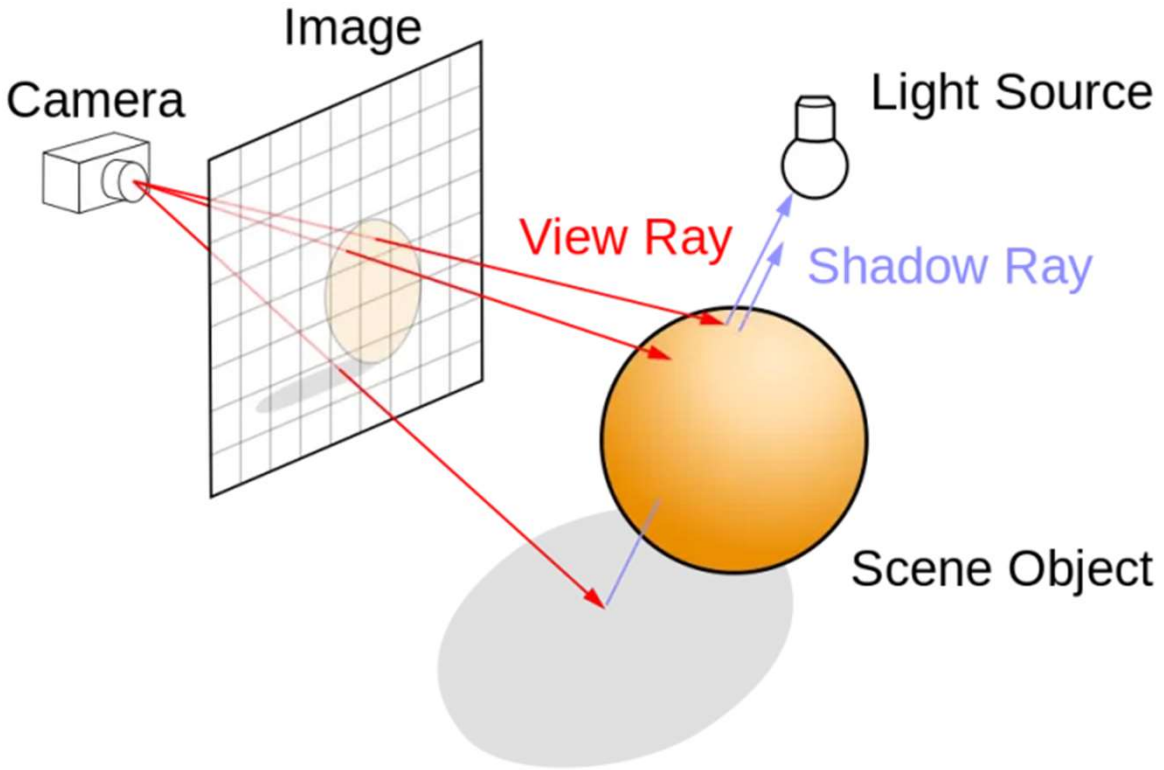
目录

硬件加速光线追踪的

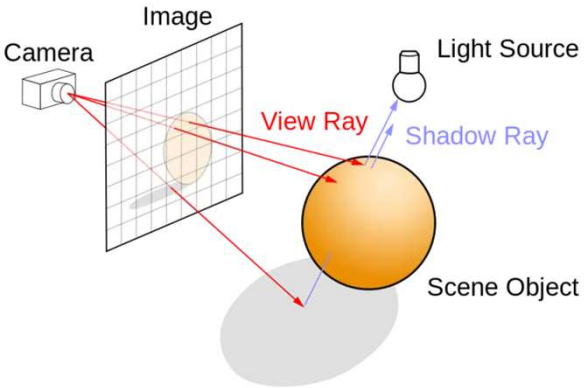
1. 原理
2. 设计
3. 应用

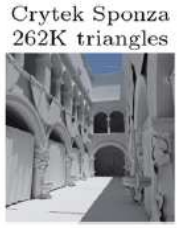
1.原理：什么是光线追踪?

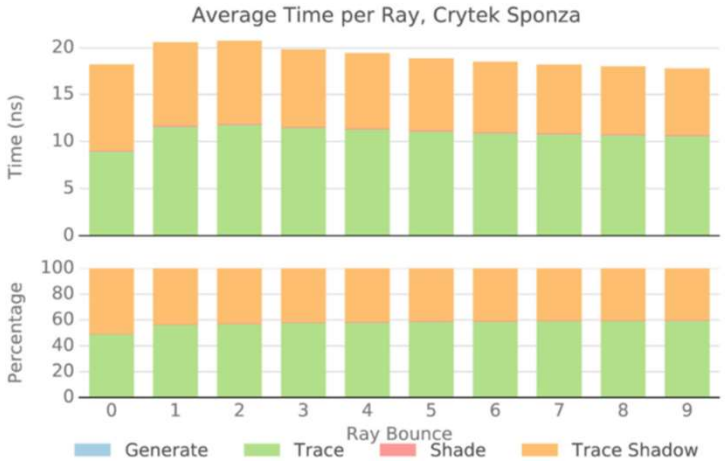
- 光路可逆
- 蒙特卡洛积分

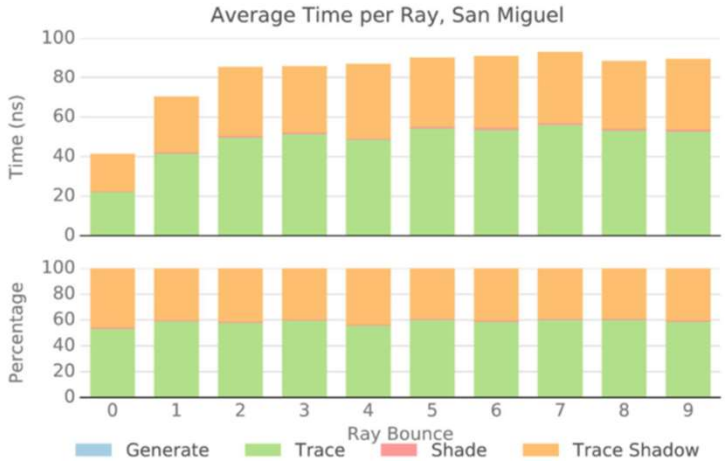


1.原理：什么是硬件加速的光线追踪?







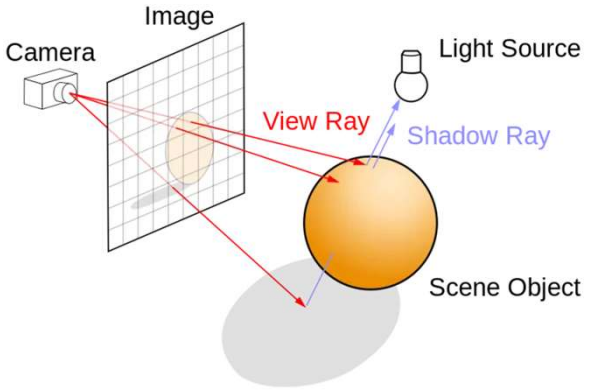


(统计图来自论文: A detailed study of ray tracing performance: render time and energy cost)

绝大部分时间都用于Trace!

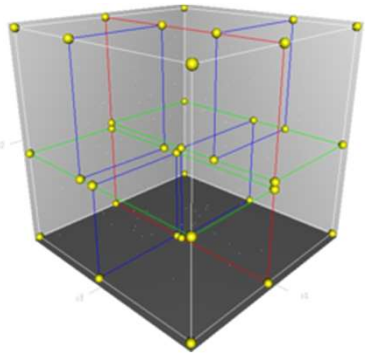
硬件加速光追的本质就是对Trace过程加速

1.原理：如何加速Trace过程?



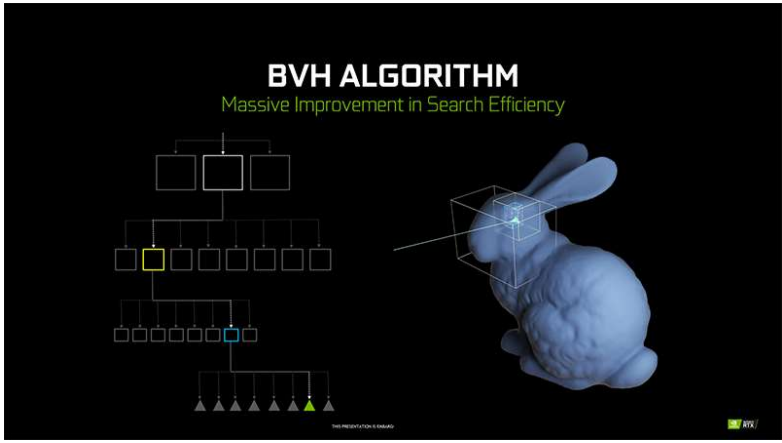
直接遍历需要时间复杂度为O(N)

需要一种空间划分方法，来快速确定光线与片元是否发生相交



1. Kd-tree(k-dimensional tree)：不断使用超平面将空间分为两个子空间。

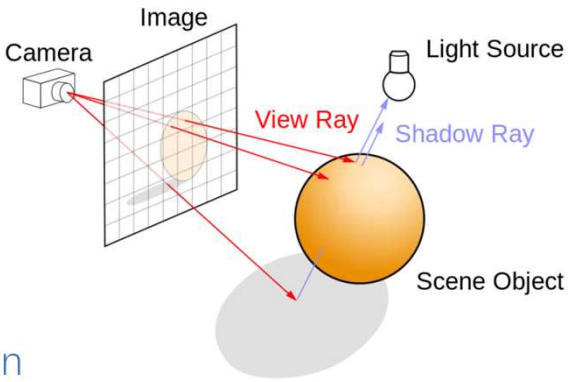
2. BVH(Bounding volume hierarchy):使用嵌套的包围盒将空间细分成更小的子空间。



BVH比起Kd-tree需要更小的内存带宽，更紧凑的遍历状态，成为了光追加速结构的事实标准(Vulkan/DirectX12)

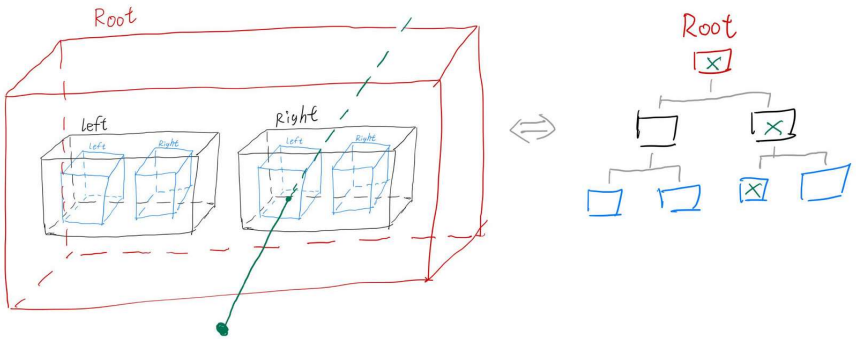
时间复杂度可以降低到O(logN)

1.原理： 如何加速Trace过程?

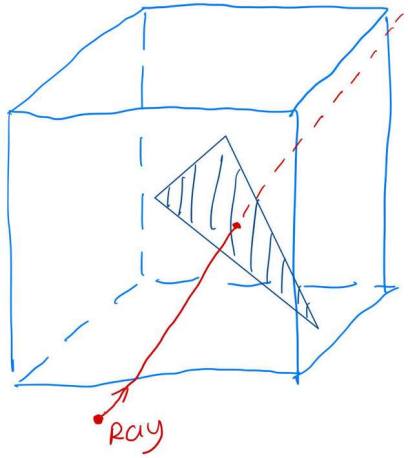


所以可以将Trace过程分解成：

1. Ray Traverse
遍历求交包围盒



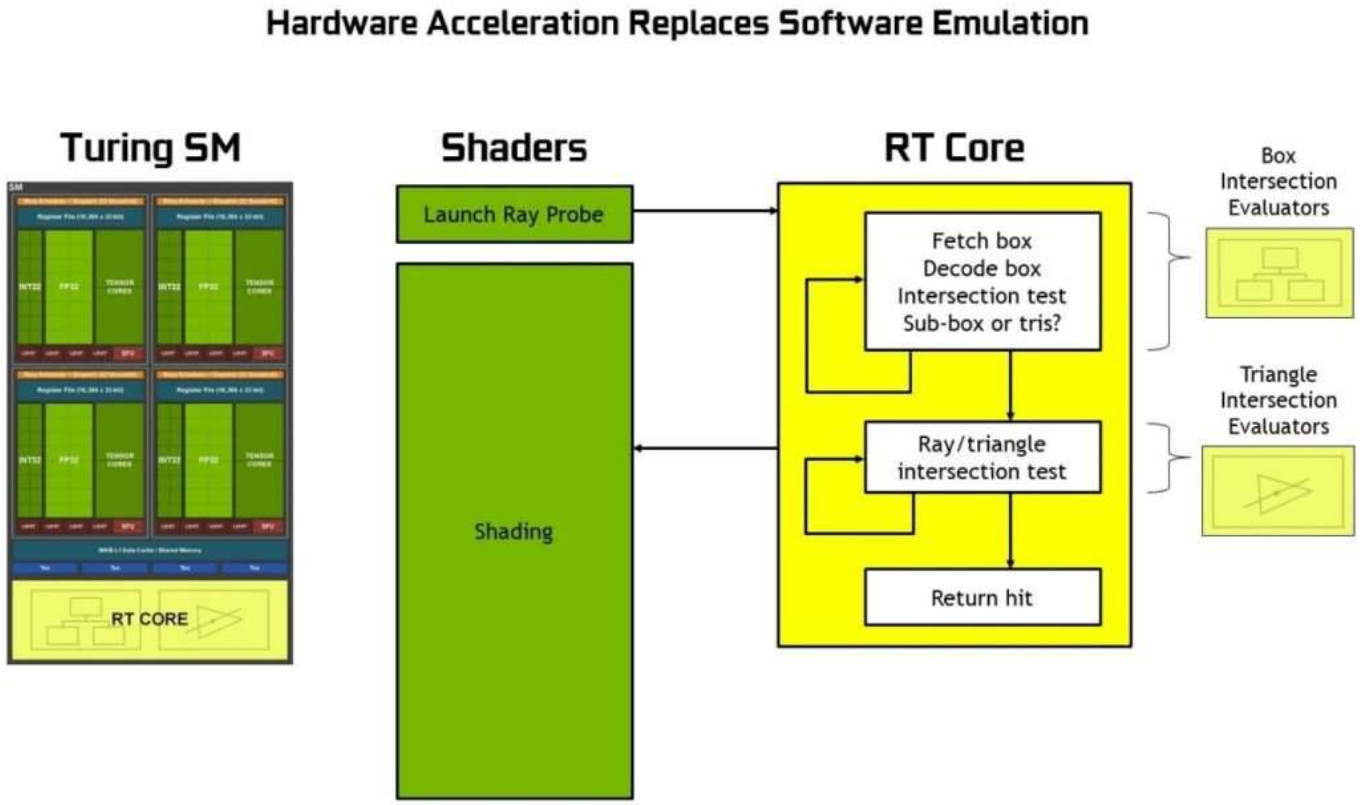
2. Triangle Intersection
对包围盒内的三角形计算其交点



与之相对应的，我们可以分别设计两个专用部件分别对Ray Traverse、 Triangle Intersection 进行加速

2.设计： Nvidia的RT Core

在Nvidia的RTX系列显卡中的光线追踪硬件RT Core也是分别实现了1. Ray Traverse 2. Triangle Intersection



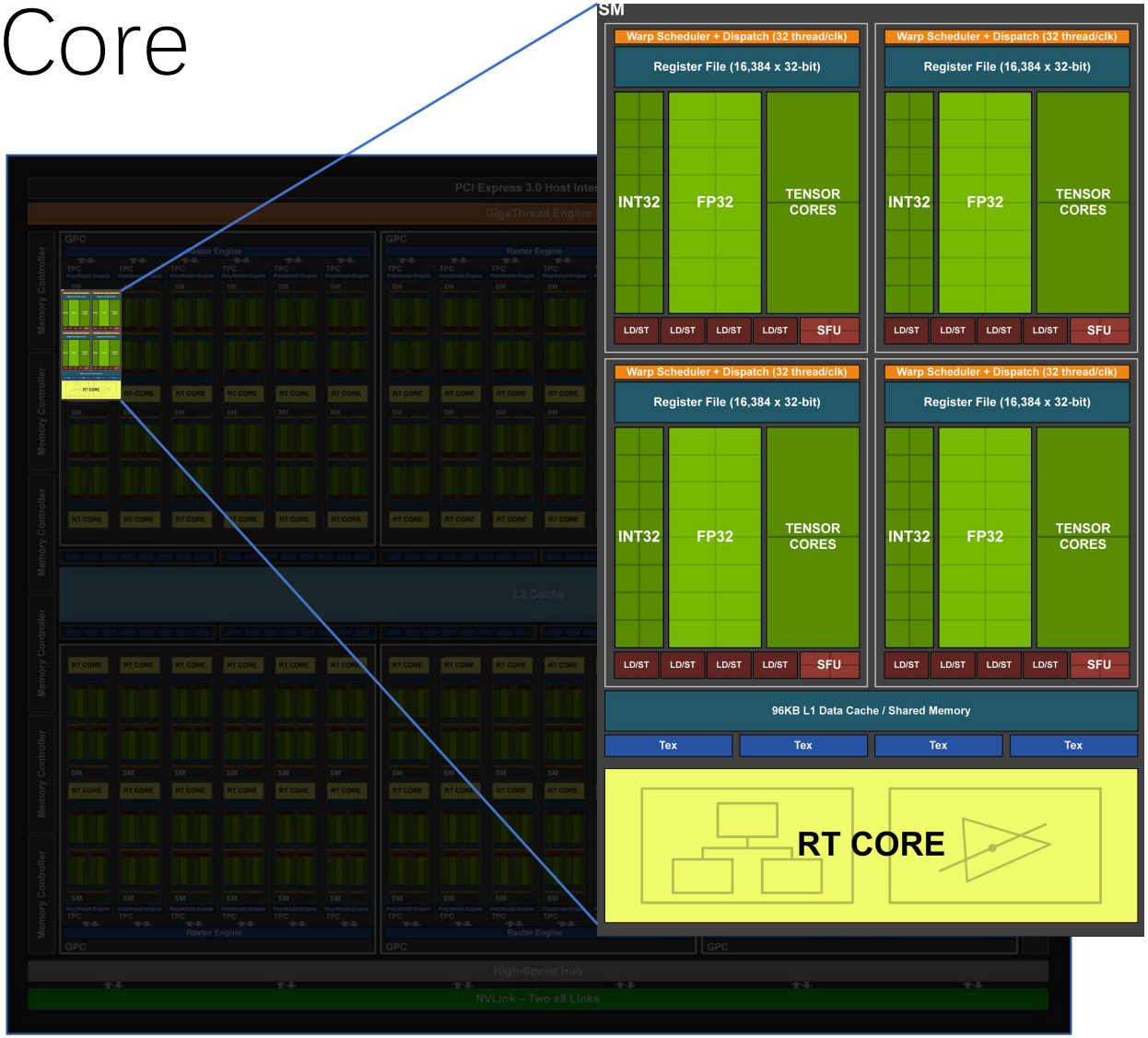
https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/

2.设计： Nvidia的RT Core

Turing架构：TU102 GPU

- 6 GPC（图形处理簇）
- 36 TPC（纹理处理簇）
- 72 SM（流多处理器）
- 72 RT核
- 每个GPC有6个TPC，每个TPC有2个SM，每个SM有1个RT核

目前性能表现最好，
但没有提供设计细节



2.设计：硬件加速光追的一种实现

RT Engine: An Efficient Hardware Architecture for Ray Tracing

Run Yan 1 , Libo Huang 1,*, Hui Guo 1 , Yashuai Lü 2, Ling Yang 1 , Nong Xiao 1, Yongwen Wang 1, Li Shen 1 and Mengqiao Lan 1

1 School of Computer, National University of Defense Technology, Changsha 410005, China

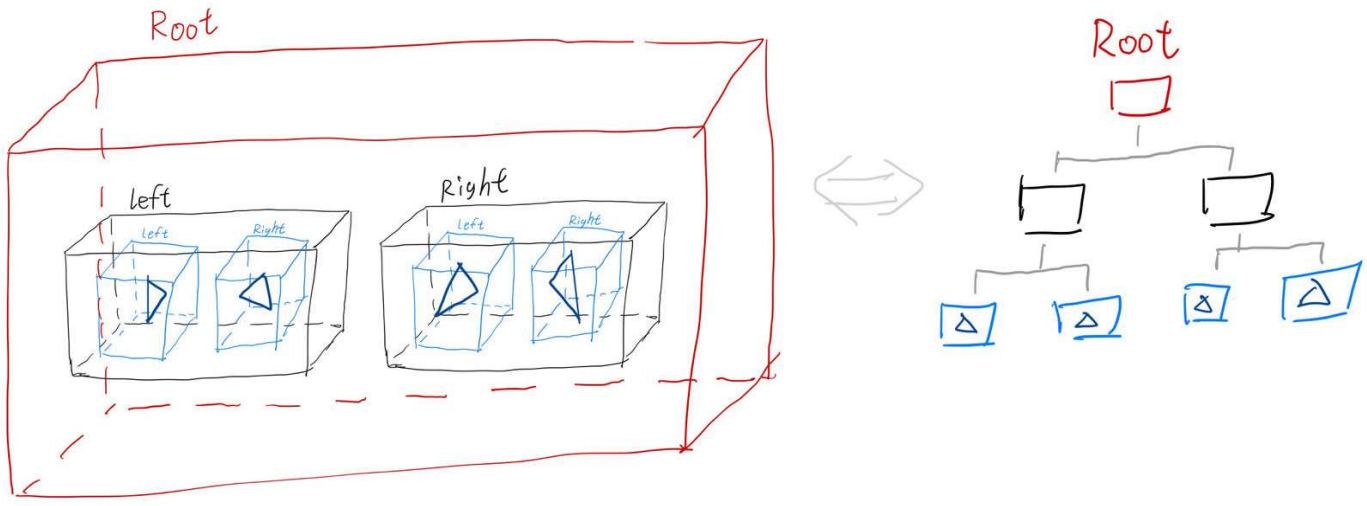
2 Huawei 2012 Labs, Beijing 100089, China

使用硬件描述语言Verilog实现了一个加速光追的专用硬件,主要加速了 Ray Traverse和 Triangle Intersection，在性能与面积比上达到现有设计中的最高。

2.设计： 核心加速数据结构acceleration structure (AS)

使用binary-based BVH作为加速结构AS，每个节点是一个AABB包围盒，其中叶节点内包含实际的三角面片。

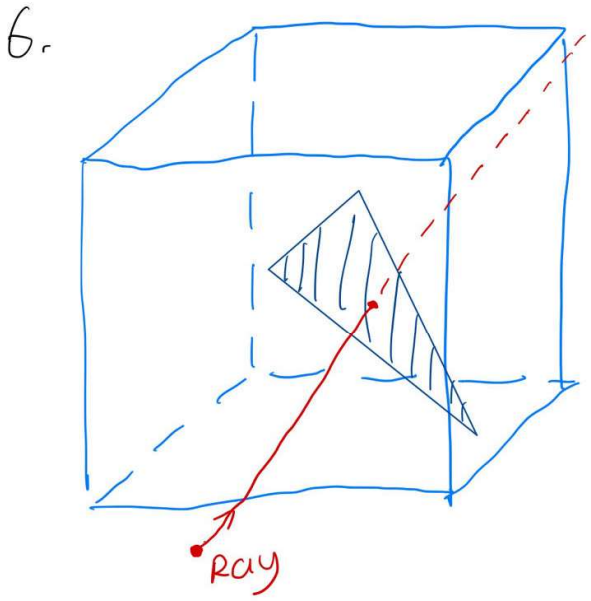
AABB(axis-aligned bounding box):对齐于坐标轴的六面长方体



2.设计：核心加速算法

基于栈的binary-based BVH Traverse算法

- 对内部节点：进行AABB盒的求交并遍历
- 对叶节点：对节点内的三角形求交



Algorithm 1: BVH Ray Tracing Algorithm

```
Input: ray, rootNode of the BVH
Output: intersection results
1  hit ← false
2  curNode ← rootNode
3  Stack ← ϕ
4  while curNode ≠ ϕ do
5      while curNode is a internal node do
6          for each child in curNode do
7              if curRay hits curNode.left and does not hit curNode.right then
8                  Traverse(curRay, curNode.left)
9              if curRay hits curNode.right and does not hit curNode.left then
10                 Traverse(curRay, curNode.right)
11             if curRay hits curNode.right and curNode.left then
12                 if hit_distance_right ≤ hit_distance_left then
13                     Traverse(curRay, curNode.right)
14                     Stack.push(curNode.left)
15                 if hit_distance_right > hit_distance_left then
16                     Traverse(curRay, curNode.left)
17                     Stack.push(curNode.right)
18             if curRay does not hit curNode.right or curNode.left then
19                 curNode = Stack.pop
20     while curNode is a leaf node do
21         for triangles in the node do
22             hitnode← Intersection(ray,traiangle)
23             if hit then
24                 return closest hit
25             break
26         curNode ← Stack.pop()
```

1 2 3 4 5 6

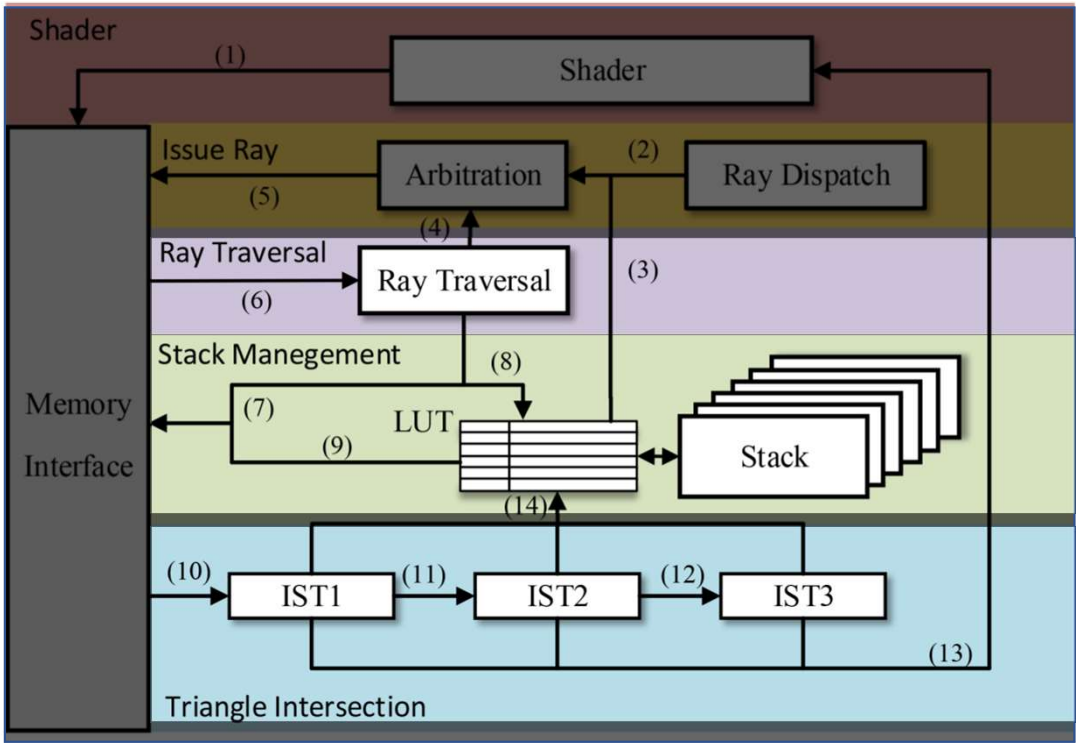
2.设计： 硬件总架构图

main contributions：

1. Traverse、Intersection 专用硬件化
2. 多栈设计，可以同时运算多个光线
3. Approximation Method for Reciprocal

Algorithm 1: BVH Ray Tracing Algorithm

```
Input: ray, rootNode of the BVH
Output: intersection results
hit ← false
curNode ← rootNode
Stack ← ϕ
while curNode ≠ ϕ do
    while curNode is a internal node do
        for each child in curNode do
            if curRay hits curNode.left and does not hit curNode.right then
                Traverse(curRay, curNode.left)
            if curRay hits curNode.right and does not hit curNode.left then
                Traverse(curRay, curNode.right)
            if curRay hits curNode.right and curNode.left then
                if hit_distance_right ≤ hit_distance_left then
                    Traverse(curRay, curNode.right)
                    Stack.push(curNode.left)
                if hit_distance_right > hit_distance_left then
                    Traverse(curRay, curNode.left)
                    Stack.push(curNode.right)
            if curRay does not hit curNode.right or curNode.left then
                curNode = Stack.pop
    while curNode is a leaf node do
        for triangles in the node do
            hitnode← Intersection(ray,traiangle)
            if hit then
                return closest hit
            break
        curNode ← Stack.pop()
```



2.设计:Triangle Intersection

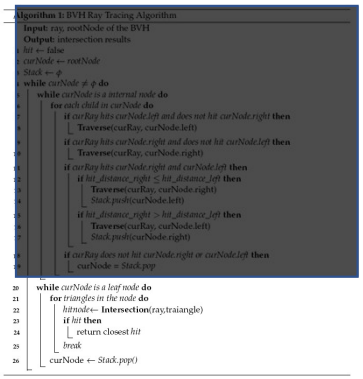
硬件化三角形与光线求交的过程

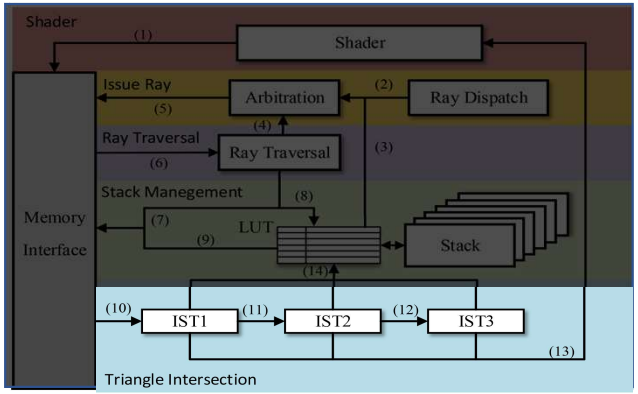
```
20  while curNode is a leaf node do
21      for triangles in the node do
22          hitnode← Intersection(ray,traiangle)
23          if hit then
24              return closest hit
25          break
26      curNode ← Stack.pop()
```

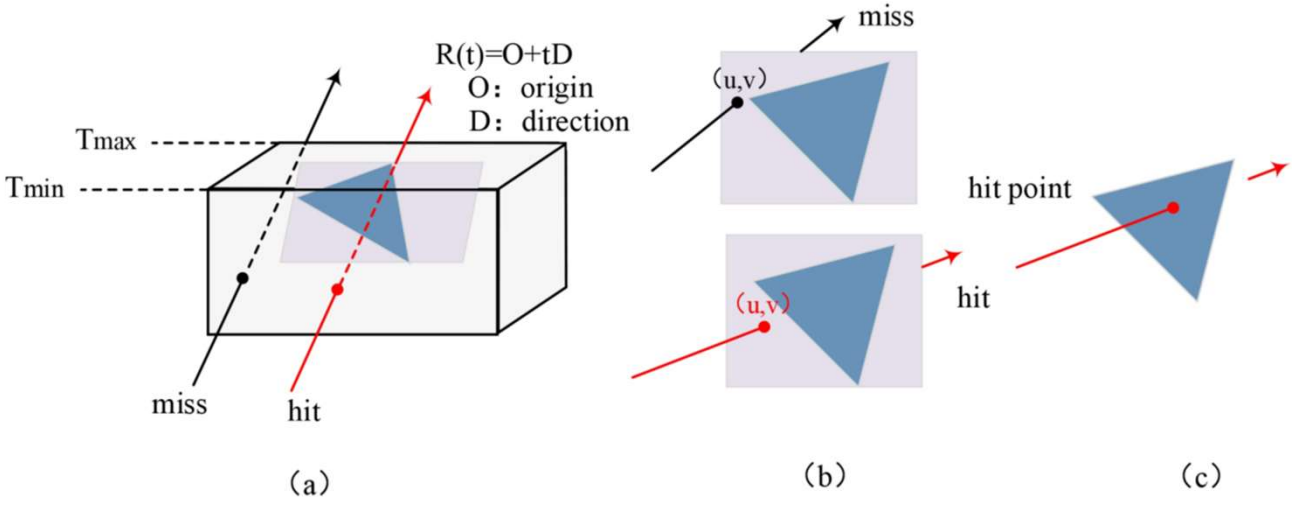
IST1: (a)过程,判断光线与三角形所在的平面是否相交,得到t

IST2: (b)过程,计算交点的关于三角形的重心坐标, 计算u， 判断是否u∈[0,1]

IST3: (c)过程,计算重心坐标v,判断是否v∈[0,1]
向Shader模块返回t,u,v



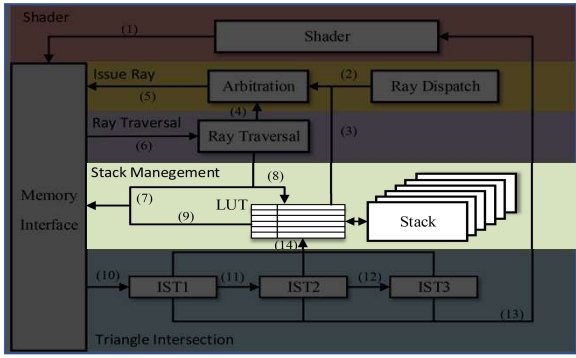




Triangle intersection test.
(**a**) Ray-plane test, (**b**) barycentric test and (**c**) final hit point calculation.

2.设计： 多栈结构

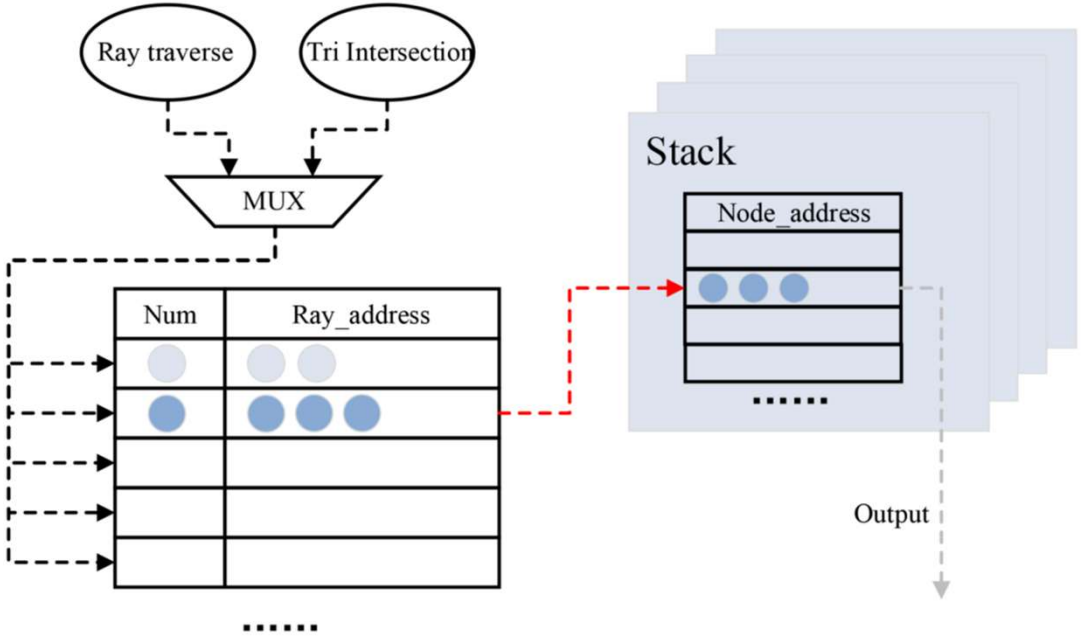
通过LUT（Look Up Table）与多栈设计，同时发射多个光线，
相较单栈设计，提升9.99–20.78x



Algorithm 1: BVH Ray Tracing Algorithm

Input: ray, rootNode of the BVH
Output: intersection results

```
1  hit ← false
2  curNode ← rootNode
3  Stack ← ϕ
4  while curNode ≠ ϕ do
5      while curNode is a internal node do
6          for each child in curNode do
7              if curRay hits curNode.left and does not hit curNode.right then
8                  Traverse(curRay, curNode.left)
9              if curRay hits curNode.right and does not hit curNode.left then
10                 Traverse(curRay, curNode.right)
11             if curRay hits curNode.right and curNode.left then
12                 if hit_distance_right ≤ hit_distance_left then
13                     Traverse(curRay, curNode.right)
14                     Stack.push(curNode.left)
15                 if hit_distance_right > hit_distance_left then
16                     Traverse(curRay, curNode.left)
17                     Stack.push(curNode.right)
18             if curRay does not hit curNode.right or curNode.left then
19                 curNode = Stack.pop
20     while curNode is a leaf node do
21         for triangles in the node do
22             hitnode← Intersection(ray,traiangle)
23             if hit then
24                 return closest hit
25             break
26         curNode ← Stack.pop()
```



每个光线将自己的内存地址送入LUT，来取得对应stack的索引

2.设计： Approximation Method for Reciprocal

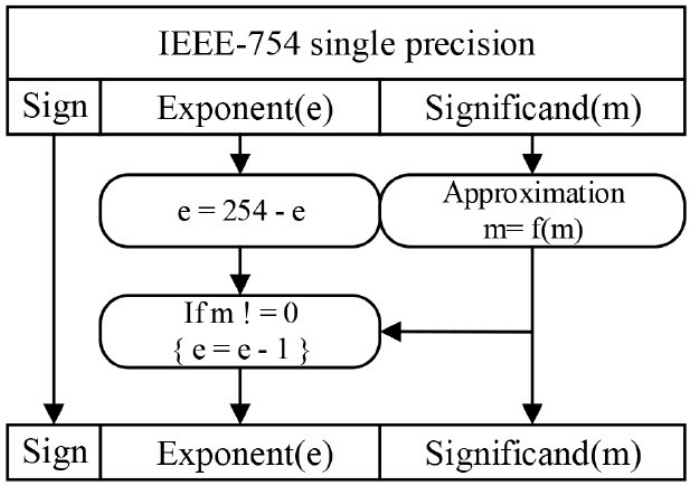
计算光线与三角形的交点的重心坐标需要求一次倒数

该设计使用Parabolic Synthesis与二次插值结合的方法实现浮点数倒数计算

1. 保留Sign位

2.对指数位取反：
(1) 去除偏移：$e' = e - 127$
(2) 取反符号再加偏移：$e'' = 127 - e'$
(3) 组合得到：$e'' = 254 - e$

3.对Significant使用Parabolic Synthesis 近似求倒数



Parabolic Synthesis：是一种低开销、低时延的硬件近似方法，
可以近似除法、三角函数、对数、平方根等函数。

并行求解多个子函数，每个函数都是二阶函数，最后通过乘法合成。

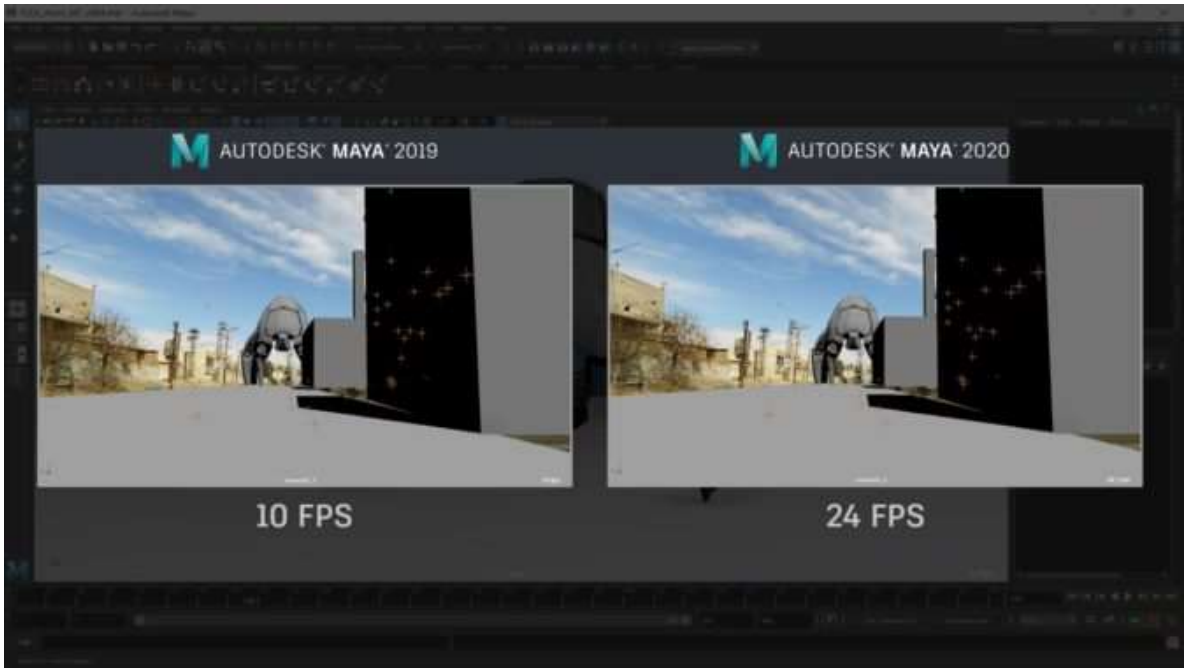
2.设计：结果比较

Table 7. Performance comparison against previous approaches.

	Clock Rate	Acceleration Structure	Performance (MRPS)	Area (mm^2)	Process (nm)	Performance/ Area (MRPS /mm^2)
T&I engine SIGGRAPH'11 [17]	500 MHz	Kd-tree	198	9.04	65	21.90
SGRT SIGGRAPH'13 [18]	500 MHz	BVH	184	7.2	65	25.56
RayCore TOG'14 [19]	500 MHz	Kd-tree	193	18	28	10.72
Two-AABB SIGGRAPH'14 [21]	500 MHz	BVH	297.6	6.82	28	43.63
HART TVCG'15 [20]	500 MHz	BVH	602	7.68	65	78.39
STRaTA CGF'15 [22]	1 GHz	BVH	365.6	57.1	65	6.40
MBVH SIGGRAPH'16 [23]	500 MHz	BVH	88	3.12	45	28.21
Dual Streaming HPG'17 [24]	1 GHz	BVH	345.6	57.1	65	6.05
Mach-RT TVCG'20 [25]	2 GHz	BVH	284.25	52	65	5.47
RT engine	850 MHz	BVH	92.74	0.48	28	193.21

在性能与面积比上达到现有设计中的最高。

3. 应用:加速电影特效制作



AUTODESK

Arnold Maya

https://blogs.nvidia.com/blog/2019/12/10/rtx-autodesk-maya-arnold/

3. 应用:加速求解KNN问题

RTNN: Accelerating Neighbor Search Using Hardware Ray Tracing (PPoPP 2022： 并行计算顶会)

利用Nvidia RTX系列显卡的RT Core 单元的加速BVH求交功能，来对2/3维的KNN（K-Nearest Neighbors） 问题进行求解。

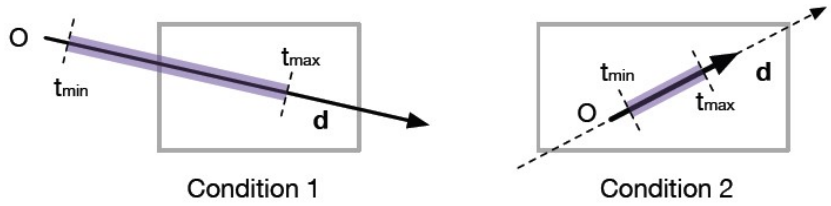
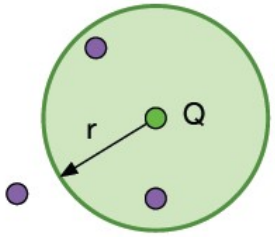
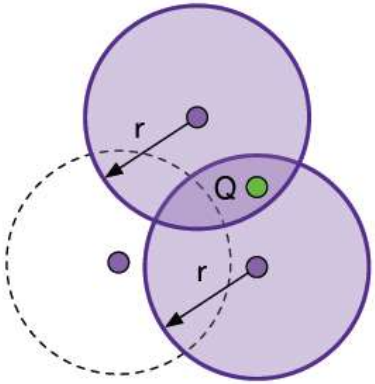


Fig. 2. Two conditions for ray-AABB intersection.



(a) Original neighbor search for query Q.



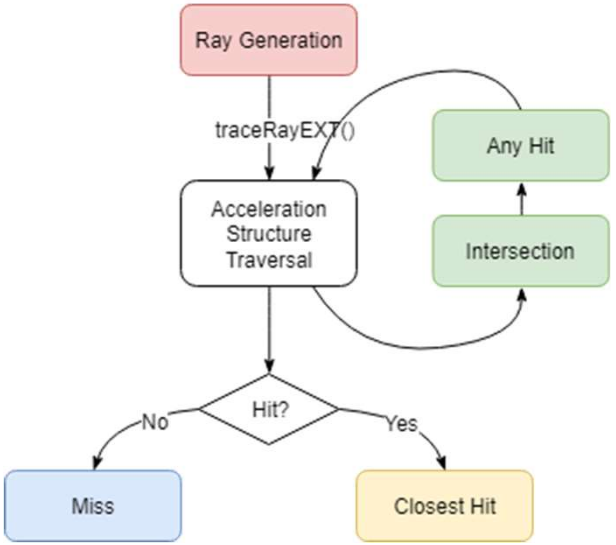
(b) Reversed neighbor search.

位置Q半径R中有几个点？ 等价于 Q位置在几个半径为R的球中

3. 应用:实时光线追踪

渲染器、游戏

实际使用时，使用Vulkan/DX12，
首先使用GPU构建加速结构（BVH），
然后编写5个shader程序即可使用：
Ray Generation、
Any Hit、
Intersection、
Miss和
Closest Hit



演示： NVIDIA Vulkan Ray Tracing Tutorials

参考

- 一篇光线追踪的入门

https://zhuanlan.zhihu.com/p/41269520

- https://github.com/nvpro-samples/vk_raytracing_tutorial_KHR
- 深入GPU硬件架构及运行机制

https://www.cnblogs.com/timlly/p/11471507.html

谢谢！