面向VR头盔的实时真实感绘制系统

2020 年 10 月 26 日

摘要

1 介绍

1.1 光线追踪和光子映射

长期以来，快速绘制具有高质量真实感的画面一直是计算机图形学的目标。在诸多的算法中，光线追踪提供了一种成功的模拟全局光照明的方案。如 [1]所述，光线追踪算法通过以摄像机为起点，朝着屏幕每个像素点的方向发射光线，这些光线在场景中与表面进行交互与传播，最终击中一个光源或发光体，并由此计算像素点的颜色。

光子映射算法是光线追踪算法诸多变种中的一种。最早来源于反向光线追踪 [2]，然后Jensen使用k-NN最近邻估计对算法进行了改进 [3]。这是一种双通道方法，在第一个通道（光子追踪通道）中，光源向场景发射携带有能量的光子，然后这些光子在场景中与表面进行交互，并在相交的非光泽表面上被记录下来，存储在被称为光子图的数据结构中。第二个通道（渲染通道）从摄像机向场景中发射光线，在交点处使用光子图中邻近的光子来进行亮度的计算（这个过程被称为光子密度估计）。假设点x附近的区域是局部平坦的，这个点沿ω方向的辐射亮度可以用公式1来计算，其中r是包含了这n个光子的球的半径，$f_r(x,\omega_p,\omega)$是点x处的BRDF反射系数，$\Delta\Phi_p(x,\omega_p)$是入射方向为ω_p的光子的能量。

$$L_r(x,\omega) \approx \frac{1}{\pi r^2}\sum_{p=1}^{n} f_r(x,\omega_p,\omega)\Delta\Phi_p(x,\omega_p) \tag{1}$$

1.2 CUDA和OptiX

CUDA，（Compute Unified Device Architecture，统一计算架构）是由NVIDIA所推出的一种集成技术，是该公司对于GPGPU的正式名称。而OptiX是一个用于在GPU上实现最佳光线跟踪性能的应用程序框架，提供了一个简单，递归和灵活的管道来加速射线跟踪算法。通过可编程的相交，光线生成和阴影，将NVIDIA GPU的功能带入光线跟踪应用程序。这两者的编程语言都是CUDA特化的C++，这样接近硬件底层的抽象程度使得我们能最大程度地利用好现有的硬件能力，包括利用GPU上的可以指定的 __shared__ memory进行加速，而且NVIDIA Turing和Ampere架构显卡具有专门用于光线求交（包括与三角形和BVH遍历）的硬件电路，相比传统的在着色器里面自己实现的光线求教快数倍。

1.3 VR和OpenVR

1.4 我们的贡献

2 基本框架

3 方法

3.1 改善内存访问模式

传统的光子映射算法 [4]常常选用kd-树 [5]作为保存光子图的数据结构，它会将光子平均地进行划分然后保存在数据结构当中。光子的位置被看做三维的向量，每个分量都被看做作为划分依据的键值。在进行划分的时候，需要首先计算一个包含所有光子的包围盒的尺寸，然后以这个包围盒最长边所在维度最中间的光子作为结点。这个光子在划分维度上的键值形成了一个平面，将剩余的光子平均分成两部分。这两部分光子将被分别分配到两棵子树当中，继续递归执行划分的过程，直到不能继续划分为止。为了避免使用指针的开销，在实际实现中可以使用完全二叉树数组来表示kd-树。

在密度估计时需要搜索某个点附近的光子，整个搜索过程从kd-树的根节点开始，计算这个点和根节点光子之间的距离，如果小于设定的半径，那么就要计算根节点光子对结果的贡献。然后计算这个点相对于根节点的位置，和这个点同一侧的光子都是可能对最终结果有贡献的，在这部分光子对应的子树上递归调用搜索过程。如果这个点到根节点光子所产生的划分平面的距离小于设定的半径，那么另一侧的光子也有可能对最终结果产生贡献，也需要对另一侧光子对应的子树递归调用搜索过程；否则另一侧的光子和这个点的距离都一定大于设定的半径，不会产生贡献，可以直接抛弃。

使用kd-树可以避免对所有光子的搜索，从而提高了搜索的效率。但是这种数据结构的问题是访问的随机性太高，在搜索过程中先后进行搜索的两个光子在内存中的位置并没有什么相关性，甚至可能隔得很远，这样就没有办法充分利用硬件设备的带宽。在光子映射算法中，最近光子搜索占据了主要的开销 [6]，如果不能充分利用带宽，那么这个部分就会成为整个算法的瓶颈。

我们将光子图的数据结构改换为哈希网格 [7]，它的主要思想是这样的：将空间均匀划分为大量的小立方体，定义的哈希函数将位于小立方体内的坐标映射成这个小立方体的编号，同一个小立方体内的光子在内存中相邻保存，这样搜索某个点附近的光子时，只需要搜索可能有贡献的小立方体对应的一片片连续的内存空间即可。

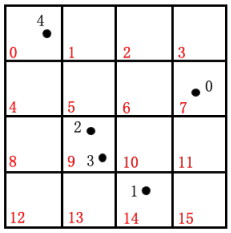
为此我们需要首先计算整个网格的大小，网格的原点被设定成各个分量都最小的光子的坐标gridOrigin（可以比这个坐标再小一点）。方便起见，每个小立方体的边长被设定为搜索半径的大小r，这样在搜索的时候每个点在搜索的时候只需要访问以它所在的小立方体为中心的$3 \times 3 \times 3$的区域即可。将整个网格的大小除以每个小立方体的边长并向上取整，可以获得网格各个维度的小立方体数量，记为gridSize。我们采用了 [6]中的做法，在每个维度的两端额外添加一层小立方体，用增大内存开销的方式避免了边界检查。哈希函数采用了简单易实现的线性映射：

$$hash(photon) = z \times gridSize.x \times gridSize.y + y \times gridSize.x + x$$

其中$x = \lfloor (photon.x - gridOrigin.x)/r \rfloor$，是光子相对于网格原点的位置除以小立方体边长向下取整的结果，y和z的计算与x类似。

获取了哈希函数的计算式之后，可以计算原始光子图中的每个光子的哈希值（也就是所在的小立方体的编号），然后根据哈希值从小到大的顺序将光子重新排序，这样就完成了光子图的构建。除此之外，还需要生成一个前缀和数组startIdx，记录每个小立方体所包含的光子的起始下标，编号为i的小立方体所包含的光子从下标startIdx[i]开始，一共startIdx[i+1]−startIdx[i]个。包括中心小立方体在内的27个小立方体的编号相对于中心小立方体的编号的偏移都是固定的，可以事先计算好，然后保存在一个偏移查看表中，在搜索的过程中查表就可以知道下一个待搜索的小立方体的编号的偏移。更高效的做法是利用相邻编号的小立方体的光子是紧挨着保存的特点，只在偏移查看表中储存9个条目。

重构光子图的过程如图1所示：左上图是场景中光子分布的示意，其中小立方体的编号用红色表示，光子的编号用黑色表示。右上图计算了场景中的每个光子的哈希值，在此之后依据哈希值从大到小将光子重新排序，下图是扫描得到的前缀和数组startIdx，它表明在0、7、9、14号小立方体中分别存在1、1、2、1个光子。



(光子编号, 小立方体编号)
(0, 7)
(1, 14)
(2, 9)
(3, 9)
(4, 0)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	1	1	1	1	1	2	2	4	4	4	4	4	5

图 1: 哈希网格光子图重构示意图

在哈希网格中进行搜索的过程如图2所示，蓝点代表需要进行密度估计的点，它所在的小立方体的编号为10，根据偏移查看表，光子图中下标从startIdx[5]开始到startIdx[8]之前，从startIdx[9]开始到startIdx[12]之前，从startIdx[13]开始到startIdx[16]之前的光子可能会对这个点产生贡献，也就是0，2，3，1号光子。最终0号光子由于距离过远被剔除，其他的光子参与最终贡献的计算。

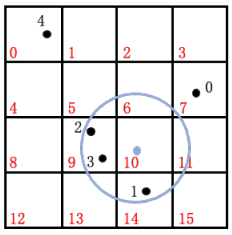


图 2: 哈希网格搜索

当kd-树和哈希网格所需要搜索的光子数差不多的时候，由于哈希网格访问的内存地址更加紧凑有序，这种方法将更加有效。这一方法存在的缺陷是，为了不漏掉搜索半径内的任何光子，必须完整搜索周围的27个小立方体，这就可能有大量不会产生贡献的光子被访问到。最糟糕的情况下，光子均匀分布在网格中，只

有$\frac{4}{3}\pi \div 27 \approx 15.5\%$被访问的光子产生贡献，这个情况是使用kd树时不会发生的。幸运的是，很多情况下这一效率并没有这么低，这部分效率的损失被紧凑的内存访问所弥补，最终表现出比kd-树更好的性能。

与kd-树相比，使用哈希网格需要额外的一个前缀和数组的空间开销。而且由于空间的均匀划分和场景的非均匀分布，小立方体内包含的光子数目差别很大，许多小立方体内部都是空的。

光子映射算法在进行密度估计时，一般使用KNN算法，并需要为此维护一个大小为K的最大堆 [4]，堆中保存了距离待搜索的点最近的K个光子，在堆中最远的光子的距离将作为估计时的半径，它一般会比事先给定的半径小 [8]。较小的堆会导致生成图像的偏差，而较大的堆则则不得不被分配到显卡的全局内存中，大量的全局内存访问和维护堆的操作开销将阻碍实时渲染的实现。为此，我们选择不使用堆，计算搜索半径内所有光子的贡献。这样做会使最终的估计半径会比使用堆时的估计半径大，一方面，最靠近带搜索点的光子的贡献会相应降低，另一方面，更多的光子会参与贡献的计算。

3.2 利用硬件能力加速

OptiX [9]是一个基于CUDA的高效的光线追踪引擎，但是在处理通用的并行任务的时候，它的效率可能不如CUDA高，为此，我们编写了两种内核函数并进行了对比：一种采用纯OptiX内核执行光线追踪和光子密度估计，另一种使用OptiX进行光线追踪，而采用CUDA进行光子密度估计。

通常，光子图存储在显卡的全局内存当中。负责执行任务的流多处理器可以使用位于芯片之上的共享内存，这部分内存的访问比全局内存更快，而且更重要的一点是，共享内存中的数据可以被一组线程共享。为了充分利用共享内存，提高带宽利用率，我们设计了如下方案：

首先，整个屏幕被划分成一些粗粒度的网格，每一个格子被分配给一个CUDA线程组，格子内的像素的计算任务被分配到单独的线程。首先由每一个线程计算当前像素对应的光线与场景的交点所处的哈希小立方体的编号，如果线程组内所有线程得到的编号都相同，说明网格内的所有像素光线都最终交于同一个哈希小立方体，它们将对完全相同的$3 \times 3 \times 3$区域内的光子进行遍历。现在，每个线程从所需要遍历的光子中选取一部分，将它们从全局内存中读入到共享内存当中，然后每个线程对共享内存内的光子再进行遍历。图3（右）是对这一过程的说明，最上方的两个小圆代表计算像素颜色的两个线程，假设它们一共需要访问全局内存中储存的4个光子，那么它们首先会将全局内存中的两个光子加载到共享内存中，每个线程加载一个光子（由黑色实线箭头表示），然后这两个线程会从共享内存中获取光子的信息，然后进行密度估计（黑色点虚线箭头表示）。然后这两个线程再次从全局内存中加载另外的两个光子到共享内存（蓝色实线箭头表示），然后再从共享内存中读取光子信息（黑色点虚线箭头表示）。

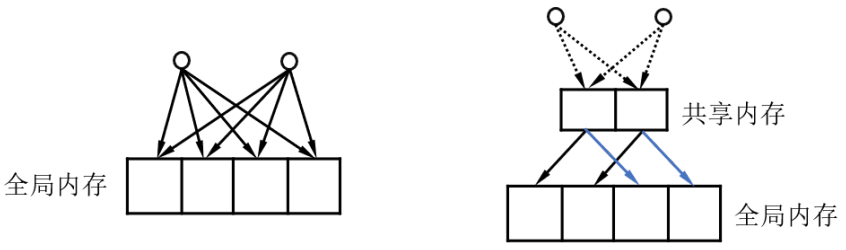


图 3: 共享内存利用示意图

不使用共享内存的做法图3（左）一共需要进行$2 \times 4 = 8$次全局内存的访问，而使用共享内存的做法图3（右）需要进行$2 \times 4 = 8$次共享内存的访问和4次全局内存的访问。由于共享内存比全局内存快的多，而且现在只需要数量更少的全局内存访问，可以更加有效利用带宽来加快速度。下面给出了算法伪代码：

Algorithm 1 利用共享内存进行密度估计算法伪代码

```
Input: array offset, array startIdx, array photonMap
photons := shared memory[group size]; /* 光子 */
hashValues := shared memory[group size]; /* 每个交点的哈希值 */
flag := shared variable; /* 是否启用算法的标志位 */
flag := 0;
hashValues[group index] := hash(hitPointPosition); /* 计算交点小立方体编号 */
group barrier;
if (hashValues[group index] != hashValues[(group index+1)%group size] then
  flag := 1;
end if
group barrier;
if flag == 1 then
  直接进行密度估计; /* 至少有一个线程的交点与其他线程不同 */
else
  /* 所有线程交点落在同一个小立方体内 */
  for i := 0; i ¡ 27; i += 1 do
    collectedCnt := 0; /* 当前线程已经访问的光子数 */
    start := hashValues[group index] + startIdx[group index];
    end := hashValues[group index] + startIdx[group index+1];
    photonCnt := end - start;/* 当前小立方体内光子数量 */
    paddedCnt := ((photonCnt - 1) / group size + 1) * group size;/* 对齐 */
    for j := group index; j ¡ paddedCnt; j += group size do
      if j ¡ photonCnt then
        photons[group index] := photonMap[start + j];
      end if
    end for
    group barrier;
    for k := 0; k ¡ group size and collectedCnt ¡ photonCnt; k += 1, collectedCnt += 1 do
      读取photons[k]计算贡献;
    end for
    group barrier;
  end for
end if
group barrier;
```

值得一提的是，如果线程组中的某个线程对应的光线没有与场景相交，它也可以协助进行从全局内存到共享内存的光子加载。

3.3 利用双眼图像相关性加速

现实世界中，我们人的双眼看到的图像是不一样的，因此如果VR设备想要提供真实感，就要对双眼所看到的图像进行分别渲染。一个简易的想法是，根据VR设备传入的眼睛的位置和投影矩阵，分别对左右眼进行图像的渲染和显示。但是事实上，左右眼看到的图像并不是独立的，有很大一部分图像是重合的。

参考文献 [10]当中采用的做法是渲染一张比目标图像更宽的图像，然后将左边的一部分图像绘制到左眼，将右边的一部分绘制到右眼。这样做是不正确的，即使是看同一个物体，左右眼的方向也不完全相同，这一现象称为“视差”，它决定了双眼看到的图像在平移后不能重合。“视差”是人感受深度的重要因素之一，忽略它将导致真实感的损失。

考虑到左右眼的光线可能会交于场景中的同一个点，那么这个点所对应的直接和间接光照计算就没有必要进行两次，可以将左眼的计算结果保存起来，在绘制右眼图像的时候直接读取对应的值即可。我们的方法是这样的：在渲染左眼图像的时候，连接左眼和每一个左眼屏幕上的像素，产生左眼光线，判断每一个左眼光线和场景的交点对右眼的可见性，如果这个交点是右眼可见的，计算这个交点在右眼屏幕上的对应的像素，然后保存在该像素对应的内存空间中。在渲染右眼图像的时候，检查当前像素对应的内存是否已经被绘制过了，如果已经被绘制过了，那就可以直接从该内存位置中读取颜色；否则进行直接和间接光照的计算。

整个过程如图4所示，为了计算左眼屏幕坐标为index的像素的颜色，一条左眼光线（蓝色实线）被发射到场景中，以它与场景的交点为出发点，朝右眼发射一条光线（红色虚线），做可见性判断，如果是可见的，就计算这个右眼光线对应的屏幕上的像素的坐标c_index。如果这个坐标位于屏幕像素的范围内，那么在c_image[c_index]中将记录该点在右眼中的颜色。这样当在绘制右眼屏幕坐标为c_index的像素的时候，查看c_image[c_index]发现已经有值了，于是直接将这个值读到帧缓存对应的位置中。

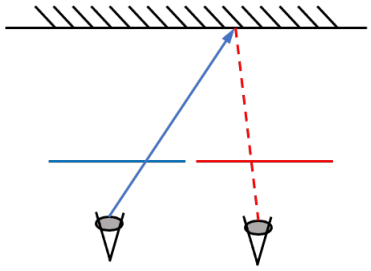


图 4: 双眼图像相关性示意图

c_index的计算方法是这样的：VR系统会提供一个矩阵M，以进行观察坐标系到世界坐标系的转换。用V_c表示观察坐标系下点的位置，V_w表示世界坐标系下点相对于眼睛的位置，有关系$M \times V_c = V_w$，由此可以计算得到：

$$V_c = M^{-1} V_w$$

如图5所示，一旦知道了V_c，就可以计算像素的位置：

$$\frac{V_c.z}{z_0} = \frac{V_c.y}{y_0} = \frac{V_c.x}{x_0}$$

$$index.x = \lfloor x_0 + \frac{size.x}{2} \rfloor, index.y = \lfloor y_0 + \frac{size.y}{2} \rfloor$$

对计算得到的像素位置，必须进行检查，确保只有在屏幕范围内的像素被绘制，否则可能会造成内存访问的错误。

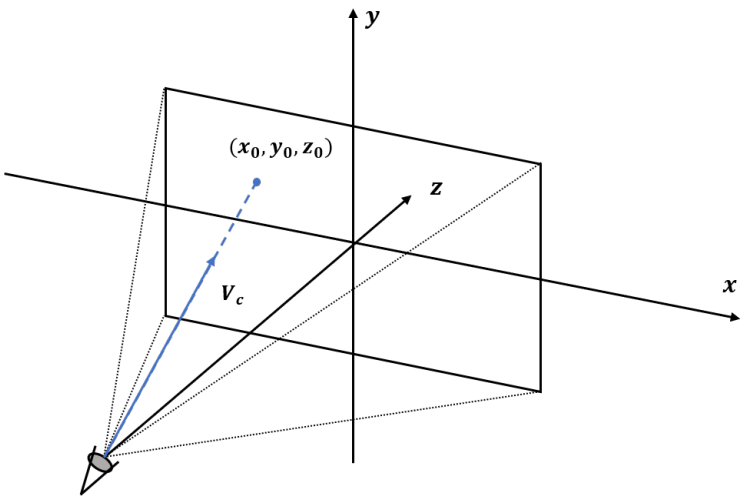


图 5: 像素位置计算示意图

4 实验

4.1 实验设备

我们使用的显卡是NVIDIA GeForce RTX 2060 SUPER。测试得到的帧时间都是连续绘制100帧然后取平均值。我们测试了使用不同的发射光子数和搜索半径的结果。每一项中得到的最好的结果用粗体表示

4.2 实验结果

我们首先比较了使用传统的kd-树和哈希网格作为光子图的数据结构时的绘制帧时间，结果列在表1中。可以看到，当发射的光子数和搜索半径不太大时，使用哈希网格会起到加速效果；当发射的光子数增多，半径也相应变大的时候。哈希网格的速度又比使用kd-树更慢了，如前文所述，哈希网格提供更紧凑的内存访问的代价是很多没有贡献的光子也会被访问到。在随后的实验中，我们都将使用哈希网格作为光子图数据结构。

然后我们比较了是否使用最大堆对绘制时间的影响，结果列在表2中。可以看到，这一步带来的加速是显著的，背后的原因主要是减少了位于全局内存中的堆的访问以及堆维护的操作开销。如前所述，这样做让那

表 1: 采用不同数据结构绘制帧时间的比较

发射光子数	搜索半径	kd-树	哈希网格
512×512	0.02	10.80ms	**6.15ms**
	0.05	112.30ms	**84.50ms**
	0.1	**470.31ms**	567.70ms
256×256	0.05	12.48ms	**7.05ms**
	0.1	67.91ms	**48.68ms**
	0.2	**314.21ms**	419.63ms
128×128	0.05	3.58ms	**3.03ms**
	0.1	8.17ms	**5.73ms**
	0.2	38.81ms	**27.16ms**

些本来最终保存在堆中的光子的贡献降低，同时会计算更多本来不会被保存在堆中的光子，这样做会使得最终得到的图像更加模糊。在后续的实验中都不使用堆。

表 2: 是否使用堆绘制帧时间的比较

发射光子数	搜索半径	使用堆	不使用堆
512×512	0.02	6.61ms	**3.27ms**
	0.05	84.87ms	**8.26ms**
	0.1	583.50ms	**23.16ms**
256×256	0.05	7.53ms	**3.50ms**
	0.1	49.30ms	**6.54ms**
	0.2	414.93ms	**20.16ms**

在表3中，我们比较了使用OptiX内核同时执行光线追踪和密度估计的方案（单一方案）与只使用OptiX内核进行光线追踪而使用CUDA内核执行密度估计的方案（混合方案）。可以看到，即使额外增加了OptiX内核和CUDA内核之间传递交点信息的通信开销，使用混合方案大体上也只需要单一方案的一半的时间。可能的解释是，OptiX在执行像密度估计这样更加通用的并行任务上的效率不如基础的CUDA高。在后续的实验中我们都将使用混合的方案。

在表4中展示了使用算法1的效果，起到的效果并不明显，查看profile发现整个程序运行的瓶颈并不在于内存访问，而在于密度估计中用到的大量的运算。尽管如此，算法1提供了利用共享内存对整个密度估计进行加速的独创思路。

表 3: 使用不同内核绘制帧时间的比较

发射光子数	搜索半径	单一方案	混合方案
512×512	0.05	7.59ms	**4.48ms**
	0.1	22.01ms	**10.46ms**
	0.2	77.92ms	**36.82ms**
256×256	0.05	3.47ms	**2.85ms**
	0.1	6.39ms	**4.12ms**
	0.2	20.04ms	**10.55ms**

表 4: 是否使用共享内存绘制帧时间的比较

发射光子数	搜索半径	不使用共享内存	使用共享内存
512×512	0.05	**4.63ms**	4.95ms
	0.1	11.86ms	**10.57ms**
	0.2	34.76ms	**32.01ms**
256×256	0.05	2.82ms	**2.79ms**
	0.1	**4.12ms**	4.25ms
	0.2	10.52ms	**9.30ms**

表 5: 两种绘制方式绘制帧时间的比较

发射光子数	搜索半径	双眼分开渲染	利用双眼相关性渲染
512×512	0.05		
	0.1		
	0.2		
256×256	0.05		
	0.1		
	0.2		

5 结论

6 不足和改进

参考文献

[1] 秦春林. 全局光照技术：从离线到实时渲染. 电子科技大学出版社, 2018.

[2] J ARVO. Backward ray tracing. *SIG-GRAPH'86 course notes*, 18(15):259–263, 1986.

[3] Henrik Wann Jensen. Global illumination using photon maps. In *Rendering Techniques 96, Eurographics Workshop in Porto, Portugal, August*, 1996.

[4] Henrik Wann Jensen and Niels Jørgen Christensen. A practical guide to global illumination using photon maps. *SIGGRAPH 2000 Course Notes CD-ROM*, 2000.

[5] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[6] Sanket Gupte. Real-time photon mapping on gpu. *University of Maryland Baltimore County*, 2011.

[7] Martin Fleisz. Photon mapping on the gpu. *Master's thesis, School of Informatics, University of Edinburgh*, 2009.

[8] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Graphics Hardware 2003*, 2003.

[9] Steven Gregory Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Patrick Luebke, David Kirk Mcallister, Morgan Mcguire, Keith Morley, and Austin and Robison. Optix: a general purpose ray tracing engine. *ACM Transactions on Graphics (TOG)*, 2010.

[10] Masahiro Fujita and Takahiro Harada. Foveated real-time ray tracing for virtual reality headset. *Poster, SIGGRAPH Asia*, 14, 2014.