文章编号：2095-2163(2020)06-0164-04 中图分类号：TP732 文献标志码：A

基于 UE4 实时射线追踪技术的研究与探讨

蒋莉珍
（桂林航天工业学院，广西 桂林 541001）

摘 要：在过去的十几年中，科技与经济的迅猛发展使得虚拟引擎的越发的普及，已经从单纯的游戏制作发展到了动画可视化等等各个方面，同时也推进了相关产业的迅猛发展。提出了利用 UE4 进行跟踪集成的方式来提高动画质量；同时通过组建新射线追踪硬件，混合渲染技术，新的去噪算法这样一个数量级的工程工作组合来提高 GPU 射线追踪时的性能。

关键词：UE4 射线追踪；混合渲染

Real-time Ray Tracing of UE4

JIANG Lizhen
（Guilin Institute of Aerospace Technology, Guilin Guangxi 541001, China）

【Abstract】 In the past ten years, the rapid development of science and technology and economy has made the virtual engine more and more popular, which has developed from the simple game production to the film visualization and other aspects, and also promoted the rapid development of related industries. This paper proposes the way of tracking integration with UE4 to improve the animation quality.At the same time, gpu-ray tracking performance can be improved through an order of magnitude combination of engineering work such as building new ray tracking hardware, hybrid rendering technology and new denoising algorithm.

【Key words】 UE4 ray tracing; mixed rendering

0 引 言

用射线跟踪生成的图像质量比用光栅化高很多[1]，一个典型的例子就是近年来动画中大多数的特技效果都用路径跟踪来完成的。但是使用射线跟踪是有挑战性的实时应用程序，例如在游戏中的应用。首先像包括包围卷层次(BVH)构造、BVH 遍历、和射线/原始交叉测试等射线跟踪算法的步骤的代价是非常高昂的。其次在射线跟踪技术中采用随机抽样也非常困难，因为通常需要数百到数千的样本，以产生一个收敛的图像，这远远超出了现代实时渲染技术的预算的计算范围，现代实时渲染技术只能按几个数量级来进行计算，并且，直到最近，实时图形 api 还没有光线跟踪支持使光线追踪集成在当前的游戏中很难被利用。随着在 DirectX 12 和 Vulkan 中射线跟踪支持的宣布彻底的改变了此前的状况。本文主要采用了 DirectX 射线追踪(DXR)并将其集成到 UE4 中，这使得可以重用现有的材质着色器代码。

1 Ue4 中集成射线跟踪

将光线跟踪框架集成到大型应用程序如虚幻引擎中是非常困难的，对于 ue4 来说已经算是非常大的构架更改了[2]。将光线追踪集成到 UE4 主要为了实现的目标如下：

(1)性能。这是 UE4 的一个关键方面，通过此方面来判断光线跟踪功能是否可以符合用户的期望。通过使用现有的基于光栅化的技术计算 g 缓冲区，然后通过对比可以帮助我们更好进行的判断，在进行计算时最重要的是射线是用来计算特定的通道，如反射或面积光的阴影。

(2)兼容性。射线跟踪通过的输出与现有的 UE4 的着色和后处理管道必须是兼容的。

(3)光的阴影一致。UE4 使用的阴影模型必须准确使用光线追踪来产生一致的阴影效果。具体来说时需要严格遵循相同的数学规则在现有的着色器代码中对 UE4 提供的各种遮阳模型做 BRDF 评估，重要性抽样，和 BRDF 概率分布函数。

(4)最小化中断。现有的 UE4 用户必须找到易于理解和扩展的集成，因此，这些集成必须遵循 UE 设计范例。

(5)多平台支持。最初在 UE4 中实时光线跟踪是完全基于 DXR，如果需要 UE4 的多平台特性支持必须设计新系统，这样在以后进行平台移动时就不必过多的关注重构技术了。

整合分为两个阶段。第一个阶段是原型阶段，本文利用了“反射”(《星球大战》的部分)和“光速”

作者简介：蒋莉珍(1990-)，女，硕士，助教，主要研究方向：数字媒体技术。

收稿日期：2019-11-19

(保时捷)演示探讨了光线跟踪技术在计算机辅助设计中的应用。以此来帮助我们获取到了有关性能、API、集成延迟着色管道渲染硬件接口中需要的更改(RHI)、一个从每个硬件的细节中抽象出用户的薄层平台、着色器 API 的变化、可扩展性等等集成中最具挑战性的方面的结论。

在找到第一阶段出现的大多数问题的答案后，进入第二阶段。这包括对 UE4 高级版本的重大重写渲染系统，除了提供了一个更好的实时光线追踪的集成，也提升了整体性能提高了着色器 API 的可扩展性等等各方面的好处[3]。

下面简述一下在高层次上将射线跟踪集成到基于栅格的实时系统中渲染引擎的几个步骤：(1)注册将被 ray 跟踪的几何图形这样保证了当图形改变时可以加速结构可以建立或更新。(2)创建命中着色器，使任何时候的射线击中一块几何图形，都可以像在基于栅格化的渲染计算它的材料参数。(3)创建光线生成着色器，跟踪阴影反射等各种用途的光线情况。

2　DIRECTX 射线追踪加速结构的背景

要理解第一步，首先要理解由 DirectX 射线跟踪 API 公开的两级加速结构(AS)。在 DXR 中两种类型的加速结构形成了一个两级层次结构：顶级加速结构(TLAS)和底层加速结构(BLAS)[4]。如图 1 所示，TLAS 构建在一组实例之上，并且每个实例都有一个有自己的变换矩阵和指向 BLAS 的指针。每个 BLAS 都建立在一组几何原语之上，几何原语可以是三角形，也可以是 AABBS，这些其中 AABBS 被用来封装自定义几何图形，由于沿着射线可以找到 AABBS 所以这些几何图形将使用自定义交叉着色器在加速结构遍历过程中进行交叉。TLAS 通常为动态场景重建每个帧，每个 TLAS 为每个独特的几何图形至少构建一次[5]。如果几何图形是静态的，则在初始构建之后不需要额外的 BLAS 构建操作。不过对于动态几何，BLAS 将需要更新或完全重建。当输入原语的数量发生变化时，需要重新构建完整的 BLAS 比如三角形或 AABBS 需要添加或删除。对于 BVH(这是 NVIDIA RTX 实现所使用的)来说 BLAS 更新需要一个 refit 操作，其中树中的所有 AABBS 都从叶子更新到根，但树结构保持不变。

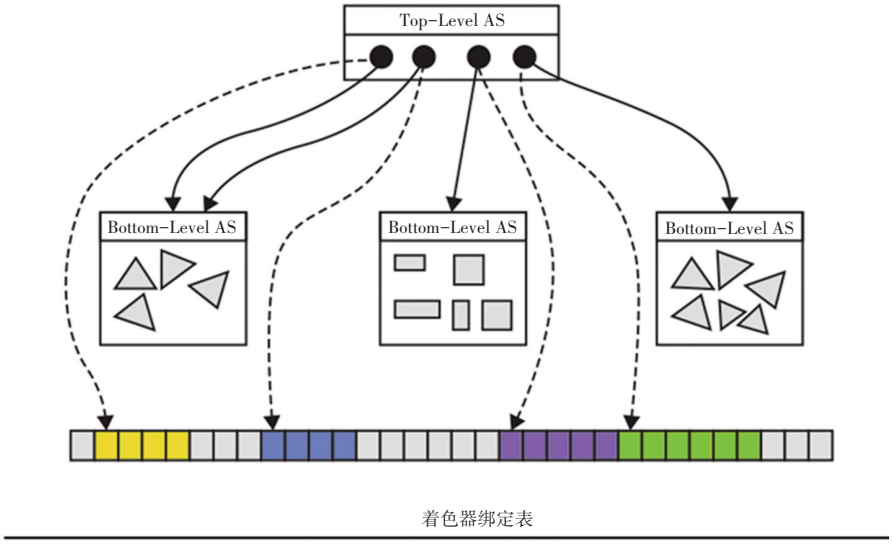

图 1　加速结构的两级层次结构

Fig. 1　A two-level hierarchy of accelerated structures

3　UE4 RHI 的实验性扩展

在实验 UE4 实现中，扩展渲染硬件接口(RHI)的抽象灵感来自于 NVIDIAOptiX API，但略有不同把整个的抽象做了相应的简化。这个抽象由 3 种对象类型组成：rtScene，rtObject，和 rtGeometry。rtScene 是由多个有效的 rtObjects 组成的实例，每个实例都指向一个 rtGeometry。rtScene 封装了一个 TLAS，而 rtGeometry 封装了多个 BLAS。rtGeometry 和任何指向给定 rtGeometry 的 rtObject 都可以由多个部分组成，所有这些都属于同一个 UE4 原始对象(静态网格或骨架网格)并且共享相同的索引和顶点缓冲区，但是他们可以使用不同的(材质)点击着色器。rtGeometry 本身没有与之关联的着色器，因此我们可以在 rtObject 部分设置着色器及其参数。

引擎材质着色系统和 RHI 也被扩展到支持 DXR 中新的射线跟踪着色器类型,在实验中主要的着色阶段为射线生成、最近命中、任意命中、交叉、错过。与这些着色器阶段的射线跟踪管道如图 2 所示。除了最近命中和任何命中的着色器,本文还扩展了引擎支持使用现有的顶点着色器(VS)和像素着色器(PS)。利用了一个扩展开源微软 DirectX 的实用工具编译器来提供一种从预编译的 VS 和 PS 的 DXIL 表示生成最近命中和任意命中着色器的机制。此实用程序将 VS 代码、输入组装阶段的输入布局(包括顶点和索引缓冲区格式和步长)和 PS 代码作为输入。给定这个输入,它可以生成执行索引缓冲区提取、顶点属性提取、格式化的最优代码,接下来是 VS 求值(对于三角形中的三个顶点),然后使用击中时的重心坐标插值 VS 输出,其结果作为输入提供给 PS。该实用程序还能够生成一个最小的任意点击着色器来执行 alpha 测试。这使得引擎中的渲染代码可以继续使用顶点和像素着色器,就好像它们将 被用来采用光栅化 G-buffer 设置着色器参数一样。

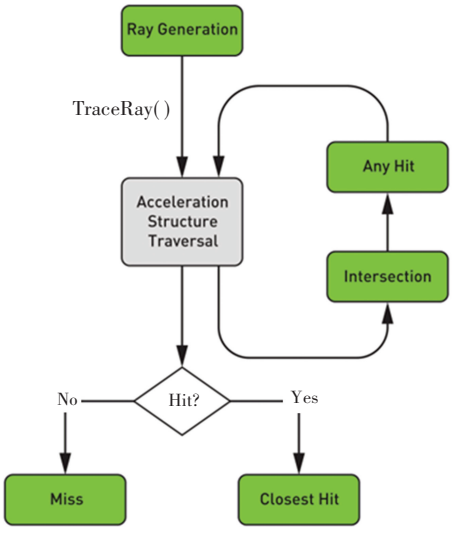

图 2 射线跟踪管道

Fig. 2 Ray tracing pipeline

在这个实验性的 RHI 抽象下的实现有两个额外的职责:编译射线跟踪着色器(到特定于 gpu 的表示)以及管理着色器绑定表,管理着色器绑定表指向需要用于每个几何图形块的着色器代码和着色器参数。

4 有效地编译射线跟踪着色器

在引入 RTX 和 DirectX 射线跟踪之前,图形 api(用于栅格化和计算)中现有的管道抽象只需要提供少量的着色器(1 到 5 个)来创建所谓的管道状态对象(PSO),该对象它封装了用于输入着色器的已编译的特定于机器的代码。这些图形 api 允许在需要时并行创建许多管道状态对象,每个对象用于不同的材质或渲染管道。然而,射线跟踪管道在很大程度上改变了这一点。射线跟踪管道状态对象(RTPSO)必须与所有可能需要在给定场景中执行的着色器一起创建,比如当一个射线命中任何对象时,不管什么样子的对象系统都可以执行与之相关联的着色器代码。在当今游戏的复杂内容中,这很容易就意味着成千上万着色器。编译数以千计的射线跟踪着色器成机器码如果顺序进行,一个 RTPSO 可能非常耗时。幸运的是,这两个 DirectX 射线跟踪和所有其他由 RTX expose 启用的 NVIDIA 射线跟踪 api 能够编译利用当今 cpu 的多核在并行多射线跟踪着色机的机器码。在实验中 UE4 实现时,可以通过简单地调度单独的任务来使用这个功能,它们各自编译一个射线跟踪着色器或命中组,形成 DXR 所谓的集合。在每一帧,如果没有其他着色器编译任务已经在执行,检查是否存在那种需求但不能用的着色器。如果找到任何这样的着色器,我们就开始新一轮的着色器编译任务。我们还检查了每一帧之前是否有着色器编译任务已经完成。如果完成了创建了一个新的 RTPSO,替换了之前的 RTPSO。在任何时候,一个 RTPSO 都用于一个帧中的所有 DispatchRays()调用。任何旧的 RTPSO 如果不在帧框架中使用时都会被新的 RTPSO 所取代。当构建 TLAS 时,当前 RTPSO 中尚未提供所需着色器的对象会被删除(跳过)。

5 着色器绑定表

这个表是一个由多个记录组成的内存缓冲区,每个记录都包含一个不透明的着色器标识符和一些着色器参数,它们相当于 DirectX 12 调用的根表(用于图形管道绘制或计算管道调度)。

由于第一个实验实现的目的是得到每一帧更新场景中每个对象的着色器参数,所以着色器绑定表管理很简单,就像命令缓冲区一样。着色器绑定表被分配为一个 n 缓冲的线性内存缓冲区,大小由场景中的对象数量决定。在每个框架中,我们利用可见的 GPU 以及可以分配缓冲区的 CPU 来编写整个着色器绑定表,然后我们将 GPU 副本从这个缓冲区加入到 GPU 本地副本的队列中。

(下转第 170 页)