LearnOptix-v7系列3-Optix基本语义

Dezeming Family

2023年8月21日

DezemingFamily系列文章因为都是免费的电子文档，所以可以很方便地进行修改和重新发布。如果您获得了DezemingFamily的系列文章，可以从我们的网站[https://dezeming.top/]找到最新版。对文章的内容建议和出现的错误欢迎在网站留言。

全局光照(GI)有很多种解决方案，比如VXGI、Lumen、DDGI、SSGI、IBL、PRT、SurfelsGI等，其中，越来越火的Nvidia的RTX技术也是一些软硬件结合的实时光追解决方案。

目录

一 Optix 7结构的介绍 **1**

1 1 总览 . . . 1

1 2 基本定义 . . . 1

1 2.1 Program . . . 1

1 2.2 Shader binding table . . . 2

1 2.3 Ray payload . . . 2

1 2.4 Attribute . . . 2

1 2.5 Buffer . . . 2

1 2.6 加速结构 . . . 2

1 2.7 Opacity micromaps . . . 3

1 2.8 Traversing the scene graph . . . 3

1 2.9 光线追踪 . . . 3

二 Optix的组件 **4**

2 1 Context . . . 4

2 2 Program pipeline的创建 . . . 4

2 2.1 Program input . . . 5

2 2.2 Module creation . . . 5

2 2.3 Pipeline launch parameter . . . 5

2 2.4 Program group creation . . . 5

2 2.5 管线链接 . . . 6

2 3 着色器绑定表 . . . 7

2 3.1 Records . . . 7

2 4 Ray generation的发射 . . . 8

三 小结 **8**

参考文献 **9**

一　Optix 7结构的介绍

本文的内容主要来自于[4, 5]。

1 1　总览

Optix 7支持场景的轻量级表示，可以表示实例化、基于顶点和变换的运动模糊，具有内置三角形、内置swept curves、内置球体和用户定义的基元。API还包括用于基于机器学习的去噪的highly-tuned kernels和神经网络。

NVIDIA OptiX的Context控制单个GPU。Context不包含大容量CPU分配，但与CUDA一样，可以在设备上分配调用启动所需的资源。它可以容纳少量句柄对象(handle objects)，用于管理昂贵的基于主机的状态。当Context被破坏时，这些句柄对象会自动释放。句柄对象（如果存在）会消耗少量主机内存（通常小于100KB），并且与所使用的GPU资源的大小无关。有关此规则的例外情况，请参阅[5]的“程序管线创建”章节。

该应用程序调用加速结构的创建（称为builds）、编译和主机设备内存传输。所有API函数都使用CUDA流，并在适用的情况下异步调用GPU函数。如果使用多个流，则应用程序必须通过使用CUDA事件来避免GPU上的竞争条件，从而确保满足所需的依赖关系。比如我们需要确保渲染都结束，然后再显示到屏幕上刷新，这时需要确实是否同步执行完毕。

应用程序可以通过一些不同的配方指定多GPU功能。多GPU功能，如有效的负载平衡或通过NVLINK共享GPU内存，必须由应用程序开发人员处理。

为了提高效率和一致性，与CUDA内核不同，NVIDIA OptiX运行时允许在任何时间点将一个任务（如单个光线）的执行移动到不同的通道(lane)、warp或流式多处理器（SM）。（请参阅CUDA工具包文档中的“内核焦点(Kernel Focus)”部分。）因此，应用程序不能在提供给OptiX的程序中使用共享内存、同步、barriers或其他SM线程特定的编程结构。

NVIDIA OptiX编程模型为未来的应用程序提供了API：随着新NVIDIA硬件功能的发布，现有程序可以使用这些功能。例如，当添加支持时，基于软件的光线跟踪算法可以映射到硬件，或者当底层算法或硬件支持这样的改变时，可以映射到软件。

1 2　基本定义

1 2.1　Program

在NVIDIA OptiX中，Program是GPU上代表特定着色操作的可执行代码块。这在DXR和Vulkan中被称为着色器。为了与先前版本的NVIDIA OptiX保持一致，当前文档中使用了“Program”一词。这个术语也提醒我们，这些可执行代码块是系统中的可编程组件，可以做的不仅仅是着色。Program有以下几类：

- Ray generation：光线跟踪管线的入口点，由系统为每个像素、采样或其他用户定义的工作分配并行调用。
- Intersection：实现在遍历过程中调用的光线-基元相交测试。
- Any-hit：当被跟踪的光线找到新的、可能的closest的交点时调用，例如用于阴影计算。
- Closest-hit：当跟踪的光线找到最近的交点时调用，例如用于材质着色。
- Miss：当跟踪的光线未命中所有场景几何体时调用。
- Exception：异常处理程序，针对堆栈溢出和其他错误等情况调用。
- Direct callables：与常规CUDA函数调用类似，直接可调用函数会立即被调用（编程规范可以与Optix的Program不同）。

- Continuation callables：与直接可调用程序不同，连续可调用程序由调度程序执行（编程规范可以与Optix的Program不同）。

光线跟踪“pipline”基于八种Programs的互连调用结构及其与通过场景中的几何数据进行搜索的关系，称为traversal。下图是这些关系的示意图：

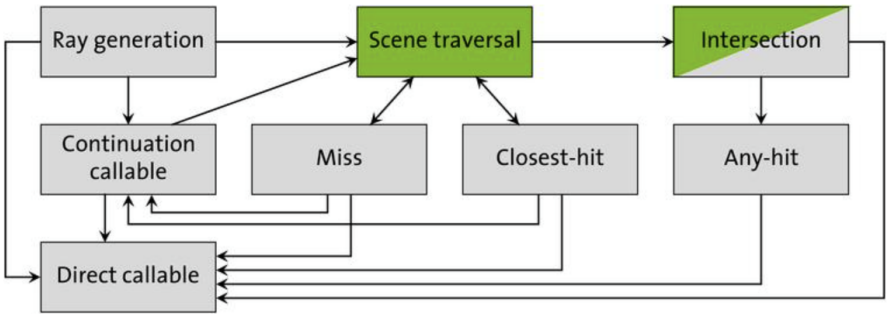


Fig. 2.1 - Relationship of NVIDIA OptiX programs. Green represents fixed functions; gray represents user programs.

绿色表示固定功能、硬件加速操作，灰色表示用户程序。Intersection一半绿一半灰是因为有些求交程序是用户自己定义的。如果启用了异常，则在发生异常的情况下，可以从任何Program或场景遍历调用内置或用户提供的异常程序。

1 2.2 Shader binding table

着色器绑定表将几何数据连接到programs及其参数。record是着色器绑定表的一个组件，在执行过程中通过使用在创建加速结构时和运行时指定的offsets来选择。一条record包含两个数据区域，标题(header)和数据(data)。

Record header：对应用程序不透明，由optixSbtRecordPackHeader填写。NVIDIA OptiX用于识别编程行为。例如，基元将在header中标识该基元的交点、any-hit和closest-hit行为。

Record data：对NVIDIA OptiX不透明，与基元或header中引用的程序相关联的用户数据可以存储在此处，例如程序参数值。

1 2.3 Ray payload

光线有效负载用于在optixTrace和光线遍历期间调用的程序之间传递数据。有效负载值被传递到optixTrace并从optixTrace返回，并遵循copy-in/copy-out语义。有效载荷值的数量有限，但这些值中的一个或多个也可以是指向基于堆栈的本地内存或应用程序管理的全局内存的指针。

1 2.4 Attribute

Attributes用于把数据从intersection programs传递到any-hit和closest-hit programs。

Triangle的intersection为重心坐标(barycentric coordinates (U,V))提供了两个预定义的属性。用户为他们自己的基元定义的intersections可以定义有限数量的其他属性.

1 2.5 Buffer

NVIDIA OptiX用指向GPU内存的指针表示GPU信息。本文档中对术语“缓冲区”的引用是指该GPU内存指针和相关的内存内容。与NVIDIA OptiX 6不同，缓冲区的分配和传输由用户代码明确控制。

1 2.6 加速结构

NVIDIA OptiX加速结构是建立在设备上的不透明数据结构。通常，它们基于边界体层次模型，但这些结构的实现和数据布局可能因GPU架构而异。

NVIDIA OptiX提供两种基本类型的加速结构：

- 几何加速结构：基于基元构建（三角形、曲线、球体或用户定义的基元）。

- 实例加速结构：构建在其他对象上，例如加速度结构（类型）或运动变换节点；允许使用每个实例的静态转换进行实例化。

1 2.7 Opacity micromaps

NVIDIA OptiX的opacity micromaps是建立在设备上的不透明数据结构。

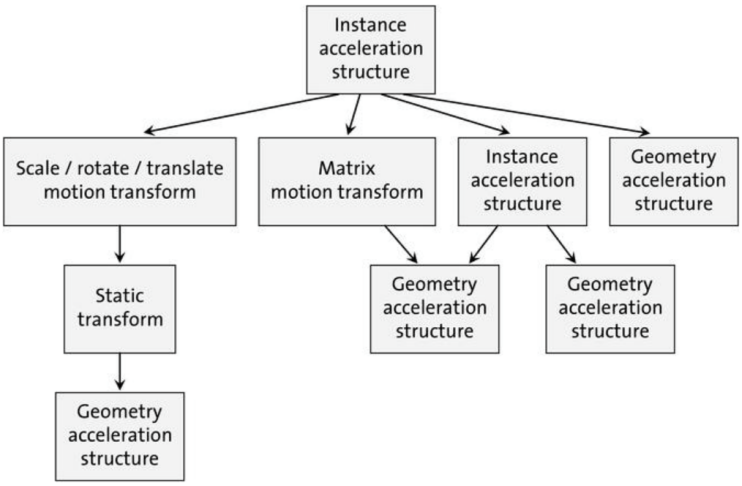
1 2.8 Traversing the scene graph

为了确定几何数据与光线的交点，NVIDIA OptiX搜索由加速结构和变换组成的节点图。这种搜索被称为遍历；节点图中的节点称为traversable objects 或 traversables。有下面的几种traversables：

- An instance acceleration structure
- A geometry acceleration structure (as a root for graph with a single geometry acceleration structure)
- Static transform
- Matrix motion transform
- Scaling, rotation, translation (SRT) motion transform

对于transformation traversables，相应的转换应用于所有子traversables（由transformation traversable的子代跨越的子图）。

transformation traversable应仅在运动的情况下使用，因为将变换应用于几何体是顺序相关的，而运动变换是时间相关的。静态变换是可用的，因为由于时间依赖性，它们不能与任何运动变换合并，而是应该与实例变换（如果需要作为实例化的子对象）或任何其他静态变换合并（即，在运动变换之后最多应该有一个静态变换）。例如，下图结合了这两种类型：



OptiX使用句柄作为traversable对象的引用。这些traversable句柄是由图形节点的设备内存指针生成的64位不透明的值。句柄标识这些对象的连接。对optixTrace的所有调用都从一个traversable句柄开始。

1 2.9 光线追踪

光线跟踪系统通过组合以下步骤中描述的四个组件来实现：

在场景中的一个或多个几何体网格和这些网格的实例上创建一个或更多个加速结构。

创建一个程序programs，其中包含将在光线跟踪启动过程中调用的所有程序。

创建一个包含对这些程序及其参数的引用的着色绑定表，并选择与加速结构中实例和几何体的隐式着色器绑定表记录选择相匹配的数据布局。

启动一个device-side内核，该内核将通过调用optixTrace的大量线程调用光线生成程序，以开始执行遍历和执行其他程序。

光线跟踪工作可以与其他CUDA工作交织在一起，以生成数据，在设备之间移动数据，并将数据移动到其他图形API。协调GPU上的所有工作是应用程序的责任。NVIDIA OptiX不与任何其他工作同步。

二　Optix的组件

虽然由于我们还没有一个完整的光线追踪器，所以介绍起来有些抽象，但也应该补充一下。在Optix 6系列文章中有比较全面和完整的介绍，应该更容易理解一些。

2 1　Context

Context由optixDeviceContextCreate创建，用于管理单个GPU。NVIDIA OptiX设备Context是通过指定与设备关联的CUDA Context来创建的。为了方便起见，可以传递0，NVIDIA OptiX将使用当前CUDA Context。

```
// Initialize CUDA for this device on this thread
OptixDeviceContext context = nullptr;
cudaFree(0);
CUcontext cuCtx = 0;
// Zero means take the current context
optixDeviceContextCreate(cuCtx, 0, &context);
```

还可以使用OptixDeviceContextOptions指定其他创建时间选项，包括用于指定回调函数、日志和数据的参数。

存在一小组Context属性，用于确定大小和限制。这些是使用optixDeviceContextGetProperty查询的。这样的属性包括最大跟踪深度、最大可遍历图深度、每个构建输入的最大基元以及每个加速结构的最大实例化数。

Context可以保留启动光线跟踪内核所需的任何GPU资源的所有权。某些API对象将保留主机内存。这些是用API中的创建/销毁模式定义的。应用程序必须调用optixDeviceContextDestroy来清理与Context关联的任何主机或设备资源。如果Context被销毁时，与此Context关联的任何其他API对象仍然存在，则它们也将被销毁。

Context可以保存解密密钥。当指定时，Context要求传递到API的用户代码使用适当的会话密钥进行加密。这最大限度地减少了输入代码受到安全攻击的风险。NVIDIA可根据要求提供Context解密功能。

只要数据传输和同步处理得当，应用程序可以组合任何支持的GPU的混合。一些应用程序可以选择通过限制这些混合的种类来简化多GPU处理，例如，通过仅混合相同流多处理器版本的GPU来简化数据共享。

如果启用了缓存，则在创建OptixModule、OptixProgramGroup和OptixPipeline对象时，输入程序的编译将缓存到磁盘。后续编译可以重用缓存的数据，以缩短创建这些对象的时间。缓存可以在多个OptixDeviceContex象之间共享，NVIDIA OptiX将负责确保对缓存的正确多线程访问。如果不希望在OptixDeviceContext对象之间共享，则可以为每个OptixDevice Context设置不同的缓存路径。通过将环境变量OPTIX_CACHE_MAXSIZE设置为0，可以完全禁用缓存。通过环境变量禁用缓存不会影响现有缓存文件或其内容。

NVIDIA OptiX验证模式有助于发现错误，否则这些错误可能会被检测不到，或者只是间歇性发生，难以定位。验证模式可在应用程序执行期间启用其他测试和设置。这种额外的处理可能会降低性能，因此只能在调试期间或已完成应用程序的最终测试阶段使用。

2 2　Program pipeline的创建

程序首先被编译为OptixModule类型的模块。组合一个或多个模块以创建OptixProgramGroup类型的程序组。然后将这些程序组链接到GPU上的OptixPipeline中。这类似于软件开发中常见的编译和链接过程。程序组还用于初始化与这些程序相关联的SBT记录的header。

2 2.1 Program input

NVIDIA OptiX程序编码在OptiX-IR（一种专有的中间表示）或PTX（一种用于并行线程执行的指令集）中。OptiX IR包含更丰富的代码表示，可以进行符号调试，并提供增强优化和未来功能的机会。OptiX IR是一种只能由NVIDIA工具读取的二进制格式，与以纯文本格式存储的PTX不同。

输入 PTX 应包括一个或多个 NVIDIA OptiX 程序。程序的类型会影响在管线执行期间程序的使用方式。通过在程序名称前面加上以下内容来指定这些程序类型：

Program type	*Function name prefix*
Ray generation	`__raygen__`
Intersection	`__intersection__`
Any-hit	`__anyhit__`
Closest-hit	`__closesthit__`
Miss	`__miss__`
Direct callable	`__direct_callable__`
Continuation callable	`__continuation_callable__`
Exception	`__exception__`

2 2.2 Module creation

一个Moudle可以包括任何程序类型的多个程序。两个选项结构控制编译过程的参数：OptixPipelineCompileOptio
对于用于创建单个管线中链接的程序组的所有模块，必须相同。OptixModuleCompileOptions：可能因同一管线中的模块而异。

在SampleRenderer::createModule()中设置了上面这些选项，我们目前只创建了一个Moudle，将全部的Program都塞到了这个Moudle中。

2 2.3 Pipeline launch parameter

LaunchParams就是在Optix进行光线追踪时会访问的参数，这里面可以包含指向最终Buffer的指针，以便于我们获得渲染结果。

2 2.4 Program group creation

OptixProgramGroup 对象由一到三个 OptixModule 对象创建，用于填充 SBT 记录的header。一共有五种Program group：

```
1 OPTIX_PROGRAM_GROUP_KIND_RAYGEN
2 OPTIX_PROGRAM_GROUP_KIND_MISS
3 OPTIX_PROGRAM_GROUP_KIND_EXCEPTION
4 OPTIX_PROGRAM_GROUP_KIND_HITGROUP
5 OPTIX_PROGRAM_GROUP_KIND_CALLABLES
```

只有第四个可以为closest-hit, any-hit, 和 intersection programs指定最多三个programs，其他都只能最多指定一个programs（下面的代码中，为其指定了两个programs）。

我们代码中的三种Program group的对象名分别是：missPGs/raygenPGs/hitgroupPGs，它们都是由我们创建的那一个Moudle来创建的。

Moudle可以包含多个Programs。Moudle中的Program由其入口函数名称指定，作为传递给optixProgramGroupCr
构的一部分。我们以我们的代码中创建Raygen program为例：

```
1 void SampleRenderer::createHitgroupPrograms() {
2     // for this simple example, we set up a single hit group
3     hitgroupPGs.resize(1);
4
5     OptixProgramGroupOptions pgOptions = {};
```

```
    OptixProgramGroupDesc pgDesc     = {};
    pgDesc.kind                      = OPTIX_PROGRAM_GROUP_KIND_HITGROUP;
    pgDesc.hitgroup.moduleCH             = module;
    pgDesc.hitgroup.entryFunctionNameCH = "__closesthit__radiance";
    pgDesc.hitgroup.moduleAH             = module;
    pgDesc.hitgroup.entryFunctionNameAH = "__anyhit__radiance";

    char log[2048];
    size_t sizeof_log = sizeof( log );
    OPTIX_CHECK(optixProgramGroupCreate(optixContext,
                                        &pgDesc,
                                        1,
                                        &pgOptions,
                                        log,&sizeof_log,
                                        &hitgroupPGs[0]
                                        ));
    if (sizeof_log > 1) PRINT(log);
}
```

注意每个我们自己定义的基元类型或者内置类型都需要定义为OptixProgramGroup对象。对于内置类型，则不需要用户自己写求交程序，而是可以用optixBuiltinISModuleGet()获取内置交集程序。但是对于内置的三角形和位移微网格三角形基元，求交程序不需要定义（也不需要optixBuiltinISModuleGet()获取），就算定义了也会被忽略。

Moudle中的Program可以在任意数量的 OptixProgramGroup 对象中使用。生成的程序组可用于填充任意数量的SBT记录。只要编译选项匹配，也可以跨管线使用程序组。

Moudle的生存期必须扩展到引用该Moudle的任何OptixProgramGroup的生存期。

2 2.5 管线链接

当 OptixPipeline 链接时，可以根据 OptixPipelineLinkOptions 和 OptixPipelineCompileOptions 选择一些固定的函数组件。这些选项以前用于编译管线中的模块。链接选项包括递归光线追踪的最大递归深度设置，以及用于调试的管线级别设置。但是，最大递归深度的值具有覆盖链接选项设置的限制的上限。

下面的代码中，使用了一个Miss program（raygenMiss[0]），一个Raygen program（raygenMiss[1]）和一个sphere的求交program（sphereGroup）。注意我们这个管线只设置了一种求交基元类型（sphereGroup中）

```
OptixPipeline pipeline = nullptr;
OptixProgramGroup programGroups[3] =
    { raygenMiss[0], raygenMiss[1], sphereGroup };
OptixPipelineLinkOptions pipelineLinkOptions = {};
pipelineLinkOptions.maxTraceDepth = 1;
optixPipelineCreate(
    optixContext,
    &pipelineCompileOptions,
    &pipelineLinkOptions,
    programGroups,
    3,
    logString, sizeof(logString),
    &pipeline);
```

关于管线堆栈的计算，相对来说比较涉及底层，大家可以参考一些现成的基于Optix开发的渲染器来理解，直接看文档有些抽象。

2 3 着色器绑定表

着色器绑定表（SBT）是一个数组，其中包含有关程序位置及其参数的信息。SBT驻留在设备内存中，由应用程序管理。比如，通过SBT可以获取击中物体的材质和几何信息。

2 3.1 Records

在我们的例子中，定义了三个Record（见SampleRenderer.cpp文件），分别是：

```
struct __align__( OPTIX_SBT_RECORD_ALIGNMENT ) RaygenRecord {
    ......
}
struct __align__( OPTIX_SBT_RECORD_ALIGNMENT ) MissRecord {
    ......
}
struct __align__( OPTIX_SBT_RECORD_ALIGNMENT ) HitgroupRecord {
    ......
}
```

record是 SBT 的数组元素，由header和数据块组成。 header内容对应用程序不透明，包含遍历执行访问的信息，以识别和调用程序。数据块并不由 NVIDIA OptiX 使用，而是保存可在程序中访问的任意程序特定应用程序信息。 header大小由OPTIX_SBT_RECORD_HEADER_SIZE宏（当前为32字节）定义。

```
struct __align__( OPTIX_SBT_RECORD_ALIGNMENT ) RaygenRecord
{
    __align__( OPTIX_SBT_RECORD_ALIGNMENT ) char header[
       OPTIX_SBT_RECORD_HEADER_SIZE];
    // user
    void *data;
};
```

API函数optixSbtRecordPackHeader和给定的OptixProgramGroup对象用于填充SBT record的header。SBT记录必须在NVIDIA OptiX启动之前上传到设备。 SBT标头的内容是不透明的，但可以复制或移动。如果在多个SBT记录中使用相同的程序组，则可以使用纯设备端内存副本来复制SBT标头。

官方代码：

```
OptixProgramGroup missPG;
......
MissRecord missRecord;
missRecord.data = nullptr;
optixSbtRecordPackHeader(missPG, &missRecord);
CUdeviceptr deviceRaygenSbt;
cudaMalloc((void**)&deviceRaygenSbt, sizeof(MissRecord));
cudaMemcpy((void**)deviceRaygenSbt, &missRecord, sizeof(MissRecord),
   cudaMemcpyHostToDevice);
```

我们样例的代码（实际执行的内容是一致的，只是放上来方便查看）：

```
OptixProgramGroup missPG;
......
MissRecord missRecord;
OPTIX_CHECK(optixSbtRecordPackHeader(missPG,&missRecord));
missRecord.data = nullptr;
missRecordsBuffer.alloc_and_upload(missRecord);
sbt.missRecordBase          = missRecordsBuffer.d_pointer();
sbt.missRecordStrideInBytes = sizeof(MissRecord);
sbt.missRecordCount         = 1;
```

只要Moudle和Program之间的编译选项匹配，就可以在管线之间重用SBT标头。可以使用optixGetSbtDataPointer备功能在设备上访问SBT record的数据部分。

着色器绑定表分为五个部分，其中每个部分表示一个唯一的程序组类型：

Group	*Program types in group*	*Value of enum* `OptixProgramGroupKind`
Ray generation	ray-generation	`OPTIX_PROGRAM_GROUP_KIND_RAYGEN`
Exception	exception	`OPTIX_PROGRAM_GROUP_KIND_EXCEPTION`
Miss	miss	`OPTIX_PROGRAM_GROUP_KIND_MISS`
Hit	intersection, any-hit, closest-hit	`OPTIX_PROGRAM_GROUP_KIND_HITGROUP`
Callable	direct-callable, continuation-callable	`OPTIX_PROGRAM_GROUP_KIND_CALLABLES`

SBT record的选择取决于程序类型，并使用相应的基指针。由于只能对光线生成程序和异常程序进行一次调用，因此这两种程序组类型不需要stride，传入的指针应指向所需的SBT record（对于Miss program，我目前的理解是，在计算阴影光线中可能也会不与任何内容求交，所以会得到多次调用）。对于其他类型，SBT record在对于给定group-type的索引sbt-index由下式计算：

group-typeRecordBase + sbt-index * (group-type)RecordStrideInBytes

例如，miss group的第三个record应该是：

missRecordBase + 2 * missRecordStrideInBytes

2 4 Ray generation的发射

射线生成发射是 NVIDIA OptiX API 的主要主力。启动调用设备上的 1D、2D 或 3D 线程数组，并为每个线程调用光线生成程序。当光线生成程序调用 optixTrace 时，将调用其他程序来执行遍历、交集、任意命中、最近命中、未命中和异常程序，直到调用完成。

管线每次启动都需要设备端内存。此空间由 API 分配和管理。由于启动资源可以在管线之间共享，因此只有在销毁 OptixDeviceContext 时才能保证释放这些资源。

三 小结

本文介绍了Optix 7的一些基本语义，作为上一篇文章的一些知识性的补充。其中有些语义我自己也不是完全明白，但为了完备性也整理到了文章中。

下一篇文章我们会介绍包含一些三角形的简单场景。

参考文献

[1] https://github.com/ingowald/optix7course

[2] https://owl-project.github.io/

[3] https://casual-effects.com/data/

[4] https://raytracing-docs.nvidia.com/optix7/guide/index.html#preface#

[5] https://raytracing-docs.nvidia.com/optix7/api/modules.html

[6] https://raytracing-docs.nvidia.com/