LearnOptix 系列 1-Optix 使用入门介绍

Dezeming Family

2023 年 1 月 8 日

DezemingFamily 系列书和小册子因为是电子书，所以可以很方便地进行修改和重新发布。如果您获得了 DezemingFamily 的系列书，可以从我们的网站 [https://dezeming.top/] 找到最新版。对书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。

进入 Nvidia 官网的光线追踪栏目 [1]，我们可以看到很多相关工具和技术，我们会在本文中简单讲解，然后重点介绍一下我们要掌握的工具：Optix。

OptiX 本身不是渲染器，它是用于构建基于光线跟踪的应用程序的可扩展框架。OptiX 引擎由两个共生部分组成：1）基于主机的 API，用于定义光线跟踪的数据结构；2）基于 CUDA C++ 的编程系统，可以生成新光线，使光线与曲面相交，并对这些相交做出响应。这两个部分一起为“原始光线跟踪”提供了低级支持。这允许用户编写的应用程序使用光线跟踪进行 collision detection, sound propagation, visibility determination 等。

在可能的情况下，OptiX 引擎避免了光线跟踪行为的规范，而是提供了执行用户提供的 CUDA C 代码的机制，以实现着色（包括递归光线）、相机模型，甚至颜色表示。因此，OptiX 引擎可用于 Whitted 式光线跟踪、路径跟踪、碰撞检测、光子映射或任何其他基于光线跟踪的算法。它可以独立运行，也可以与 OpenGL 或 DirectX 应用程序一起运行，用于混合光线跟踪光栅化应用程序。

2023-04-09: 补充了两种自己构建开发环境的方法

目录

一　多种 Nvidia 光追模块介绍

本节分文两个小节：一是介绍 Nvida 中 RTX 相关的常用模块和工具；二是支持 RTX 的工具。

1 1　Nvida 中 RTX 相关的常用模块和工具

RTXGI：RTX Global Illumination SDK 利用光线跟踪提供了可扩展的解决方案来计算多反弹间接光照，而无需 bake 时间（计算前烘焙）、光泄漏或昂贵的每帧成本。RTXGI 在任何支持 DXR(DirectX Raytracing) 的 GPU 上都受支持。目前在 Unity 和 UE4 都能够应用。

RTXDI：比如游戏环境中添加数百万个动态光源，而不必担心性能或资源限制（比如微型 LED、时代广场广告牌，甚至是爆炸的火球）。RTXDI 在实时渲染的同时实现了这一点，并且很容易将用户生成的模型中的照明结合起来。任何形状的几何体都可以发光、投射适当的阴影，并可以自由动态地移动。

DLSS：深度学习超级采样是一种神经网络图形技术，它使用 AI 提高性能，通过图像重建创建全新的帧并显示更高的分辨率，同时提供一流的图像质量和响应能力。

NRD：Real-Time Denoisers 是一个 spatio-temporal、API-agnostic 的去噪库，旨在处理低每像素光线追踪信号。它使用输入信号和环境条件来提供与 ground-truth 图像相当的结果。

Micro Mesh：是一种用于 extreme geometry 和实时光线追踪的图形图元。通过 NVIDIA GeForce RTX40 系列 GPU 的加速，该技术可有效存储不透明度和位移，并允许资产 (assets)（比如一些模型或者定义的纹理）在直接光栅化或光线跟踪中以完全保真的方式使用。Micro Mesh 提高了渲染复杂几何体（如扫描的工件、生物、岩石和树木）的实时性能和内存压缩。

1 2　支持 RTX 的工具

NVIDIA RTX 平台结合了光线追踪、深度学习和光栅化，利用 NVIDIA Turing GPU 上的 RT cores，增强了内容创作者和开发人员的创作过程。软件开发人员可以使用 OptiX API 来利用此平台。OptiX AI 去噪器使用 NVIDIA Tensor Cores，通过减少生成无噪声图像所需的时间来加速光线跟踪。注意开普勒 GPU 不支持 RTX 模式，需要 Maxwell GPU 或更高版本。

在 [1] 网页靠下方可以找到 RTX-Enabled Applications，表示一些支持 RTX 的应用，包括支持插件的游戏引擎（Unreal Engine NvRTX 和 Unity）、支持插件的可视化应用（autodesk arnold、blender、octane render 和 vray）以及图形工具（NVIDIA OPTIX、NVIDIA MDL 和 NanoVDB）。我们这里只讲这三个图形工具。

OPTIX：NVIDIA 的可编程 GPU 加速光线跟踪管道，提供围绕性能和易用性构建的高度可扩展渲染解决方案。

MDL：材质定义语言 (Material Definition Language) 允许基于物理的材质和光源库在应用程序之间无缝交换，同时保持其外观。

NanoVDB：为 OpenVDB 提供实时渲染的 GPU 支持。OpenVDB 是一个获得奥斯卡奖的开源 C++ 库，包括一个新颖的分层数据结构和一套工具，用于高效存储和处理三维网格上离散化的稀疏体数据。它由梦工厂动画公司开发，用于剧情片制作中常见的 volumetric 应用，目前由学院软件基金会（ASWF）维护。

最近，高性能 API 设计已经向提供较低级别的资源管理和执行调度控制转变。这种转变使经验丰富的开发人员能够完全控制其应用程序，同时仍能充分利用高度优化的 API 的优势。较低级别的控制还为开发人员提供了更大的灵活性，以便 API 的使用能够更好地满足其应用程序的需求。这种转变的例子可以在当前的 DirectX 和 Vulkan SDK 中看到。在最新的 7.0 版本中，OptiX 加入了这一转变，提供了由 OptiX 运行时内部管理的核心功能的直接控制，包括主机和设备端内存分配、多 GPU 工作分配和异步调度。

光线跟踪 SDK 有多种选择，它们利用 NVIDIA RTX 技术栈和专用光线跟踪 RT-Core 硬件；Microsoft 的 DXR API 在 DirectX 环境中提供光线跟踪功能；VK_NV_ray_tracing 扩展为 Vulkan API 添加了类似的支持；NVIDIA 的 OptiX SDK 为 CUDA 程序带来了光线跟踪。前两个 API 专注于与实时应用程序

的紧密集成，因此其功能集受到限制。另一方面，OptiX 以产业级渲染为目标，为运动模糊和多级变换层次等高级功能提供内置支持。

1 3 Optix 简介

OptiX API 已在多种应用中使用多年。它经过了多次实战测试，是将 GPU 光线追踪推向世界的关键工具。自 NVIDIA 的 RT-Core 硬件发布以来，人们越来越希望将 GPU 加速应用于更大、更复杂的光线跟踪工作负载，例如故事片的最终帧 (final-frame) 渲染和超大数据集的交互式预览。为此，Nvidia 重新设计了 OptiX API，以在实现此类应用程序时提供更大的灵活性。这个新的 API（Optix 7）级别较低，类似于 DXR 或 Vulkan 光线追踪的抽象级别，但保留了 OptiX 中一直存在的许多关键概念。

在 SIGGRAPH 2019 上，NVIDIA 发布了两个新版本的 OptiX API：OptiX 7.0，其中包含新的低级 API；以及 OptiX 6.5，这是对经典 API 的更新。通过 [3] 中的表可以看到，对底层控制和硬件性能支持最好的就是 Optix 7。

OptiX 的核心是一个简单但强大的光追抽象模型，该光线跟踪器使用用户提供的程序来控制光线的起始、光线与曲面的相交、材质的着色以及新光线的生成。光线携带用户指定的有效数据，用于描述每条光线的属性，如颜色、递归深度、重要性或其他属性。开发人员以基于 CUDA C 的功能的形式向 OptiX 提供这些功能。由于光线跟踪是一种固有的递归算法，OptiX 允许用户程序递归生成新光线，内部执行机制管理递归堆栈的所有细节。OptiX 还提供了灵活的动态函数调度和复杂的变量继承机制，从而可以非常通用和紧凑地编写光线跟踪系统。

Optix 的另一个很好的功能是，光线跟踪的执行通常是“次线性 (sub-linear)”的，因为如果场景中对象数量提高两倍，运行时间不会增加两倍。这是通过将对象组织成一个加速结构来实现的，该加速结构可以在整个基元组不是与任何给定光线相交的候选者时快速拒绝它。对于场景的静态部分，此结构可以在应用程序的整个生命周期中重复使用。对于场景的动态部分，OptiX 支持在需要时重建加速结构。该结构只查询其包含的任何几何对象的边界框，因此可以任意添加新类型的基元，并且只要新的基元可以提供边界框即可使用该加速结构。

二 Optix 使用入门

在 [3] 可以找到如何开始使用的方法，以及 [4] 有比较详细的资源列表：

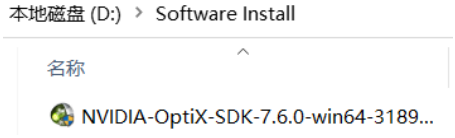
Overview
Programming guide
API documentation
Developer forum
SDK download
Release notes

NVIDIA OptiX API 是用于在 GPU 上实现最佳光线跟踪性能的应用程序框架。它为加速光线跟踪算法提供了简单、递归和灵活的管道。OptiX 可以将 NVIDIA GPU 的强大功能应用于光线跟踪应用程序，包括可编程 intersection、ray generation 和 shading，以及 RTX 的性能改进。

本节讲解如何搭建 Optix 开发环境，并且跑出一个最简单的例程。下一节将介绍一个最简单的例子。

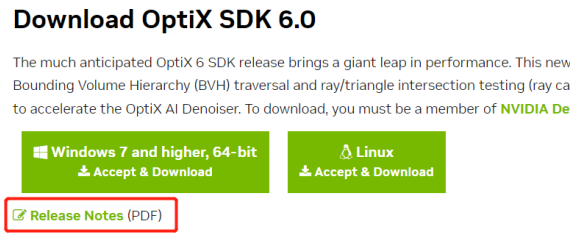
2 1 下载与安装

在上面所示的资源列表中下载，大家可以在官网中看到对 Optix 支持的硬件要求。如果自己的电脑不支持 Optix 7（尤其是 GTX 1080 以下的 GPU），就从 [6] 下载老版本。比如当前的最新版本是：



但由于目前我的笔记本显卡不支持 Optix 7，所以暂时用 Optix 6 做实验，等后面有实验环境了再写关于 Optix 7 的实验方法。大家还要注意安装对应的 CUDA 版本，这里不再赘述，在 [6] 中不同版本的

Release Notes 上可以找到支持的软硬件配置：



安装好以后可以在 [5] 找到编程指南，但是编程指南中对于 Optix 的细节描述太多（尤其是有不少关于 GPU 底层的内容），我觉得还是得先更快入门动手操作，否则就容易摸不清头脑。

我们打开安装路径，比如我的路径时默认路径：

```
C:\ProgramData\NVIDIA Corporation\OptiX SDK 6.0.0\SDK-precompiled-samples
```

然后就能看到很多例子，我们可以随便运行一些，看看自己的电脑是否硬件支持。我们可以在该目录下看到 ptx 子目录，ptx 目录里面有很多 ptx 文件，这是用来生成 Optix 光线追踪的光线的文件，是在编译中生成的，如果你编译时没有保留这些文件，那么就无法运行 Optix 代码。

下面的目录里有不少工程文件，以及一个 CMakeLists.txt：

```
C:\ProgramData\NVIDIA Corporation\OptiX SDK 6.0.0\SDK
```

我们把目录下的工程源文件进行 CMake，得到一系列的 Optix 样例代码。我把样例编译到了下面的目录：

```
D:\Develop\C++\Optix-Examples
```

编译通过则可以自己先把这些示例代码跑一跑，粗略看一下代码结构和一些函数。

我尝试过用 VS2019 以及 CUDA v12.0 版本来编译 Optix 6.0.0，结果都失败了，可见 Optix SDK 的编译对版本要求是很高的（也有可能是安装 SDK 时自动选择了自己的计算机的 CUDA 版本，但由于我的计算机有多个 CUDA 版本，所以导致只有一个可以编译成功）。最终我用 VS2015 和 CUDA v10.1 编译成功。

2 2　建立自己的工程

我们的目标是新建一个工程，可以编译和运行示例中的程序。我们本节建立的工程不能自己编译和生成.ptx 文件（类似于生成的 shader），而是运行给的例子中的.ptx 文件。

新建一个 Visual Studio 的控制台应用工程，然后添加.cu 文件到该工程下。之后在 VS 里设置工程包含的头文件目录、库目录和可执行文件目录。

首先是头文件目录，包括 CUDA 目录和 Optix 目录：

```
C:\ProgramData\NVIDIA Corporation\OptiX SDK 6.0.0\include
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.1\include
```

然后是一些示例的 utility 工具的头文件目录（sutil 和 OpenGL）：

```
C:\ProgramData\NVIDIA Corporation\OptiX SDK 6.0.0\SDK
C:\ProgramData\NVIDIA Corporation\OptiX SDK 6.0.0\SDK\cuda
C:\ProgramData\NVIDIA Corporation\OptiX SDK 6.0.0\SDK\sutil
C:\ProgramData\NVIDIA Corporation\OptiX SDK 6.0.0\SDK\support\freeglut\
    include
```

注意 sutil 中有些头文件还需要包含相应的.c 和.cpp 文件，注意这些文件，比如 sutil.cpp 并不需要自己的工程去引用，该文件中定义的函数都是要生成 dll 库的，在我们当前的项目中由于是要生成可执行文件，所以不能加入进来，我们需要把它们在示例程序中编译生成的 dll 和 lib 文件拷贝到当前工程下，然后在 VS 的附加依赖项中输入：

```
sutil_sdk.lib
freeglut.lib
optix.6.0.0.lib
optix_prime.6.0.0.lib
optixu.6.0.0.lib
```

由于我们使用了 freeglut.lib，我们也不需要再导入 glew.c 到当前工程下（它们都是 OpenGL 工具库，我们只需要一个即可）。

然后还有一个 sampleConfig.h，是编译生成例子时生成的头文件（不同的环境下生成的该头文件的不同的，不可混用），在生成的样例的目录下：

```
D:\Develop\C++\Optix-Examples\sampleConfig.h
```

也复制过来。

然后需要包含库和可执行文件目录（为了方便可以直接把这些目录里的文件打包放在了一个目录下）：

```
C:\ProgramData\NVIDIA Corporation\OptiX SDK 6.0.0\bin64
C:\ProgramData\NVIDIA Corporation\OptiX SDK 6.0.0\lib64
C:\ProgramData\NVIDIA Corporation\OptiX SDK 6.0.0\SDK\support\freeglut\win64
    \Release
```

Visual Studio 中随便打开一个编译好的例子的属性，然后选择 C/C++，然后选择“预处理器”，找到“预处理器定义”，可以看到工程预定义的宏：

```
WIN32
_WINDOWS
_USE_MATH_DEFINES
NOMINMAX
GLUT_FOUND
GLUT_NO_LIB_PRAGMA
CMAKE_INTDIR="Debug"
```

这些宏也要加到我们自己的工程里。

这些配置看起来比较繁琐,我也是花了很长时间才把配置都搞清楚的,这主要是因为需要包含 OpenGL 工具作为显示输出窗口。但注意此时我们的项目并不会编译 cuda 文件，而是直接使用示例中早已生成的 ptx 文件来进行渲染。后面我们解释一些基本的 Optix 编程模型以后，再介绍如何建立一个比较完备的工程（生成并使用 ptx 文件）。

三　如何利用 Optix 开发

本节分为两个部分，第一部分我们建立一个最基本的工程文件，glut 窗口会不断地变换颜色；第二部分我们开始使用 Optix 来通过光追渲染一个物体。

由于 Optix 6 中各个版本区别不是很大，所以可以参考 [7] 的编程指南。

3 1　基本工程文件

程序代码见 1-3-1 目录，我们只有一个 main.cpp 文件，这里并不需要任何 sutil 等其他工具，只需要最基本的 OpenGL 和 Optix 库（但我们暂时不使用 Optix 的功能）即可。

glutRun() 函数中调用包括渲染程序和事件响应回调函数。glutDisplay() 负责渲染，我们仅仅改变窗口的颜色，让窗口颜色不断变化。

3 2 编程模型

Optix 是一个基于对象的 C 语言 API，编程模型分为主机代码和 GPU 设备程序。主机会定义对象、程序和变量，然后在 GPU 中使用。

Optix 编程模型包括对象模型和组件程序 (Component programs)，这两部分并不是独立的，对象也包括一些功能。

对象模型

主要对象有：

- Context：用于运行 Optix 引擎的实例。
- Program：CUDA 函数，编译为 NVIDIA PTX 虚拟汇编语言 (virtual assembly language)。
- Variables：一种变量，用于将数据传入 Optix 程序。
- Buffer：绑定到一个变量的多维数组。
- TextureSampler：Buffer 的插值机制。
- Geometry：ray 可以相交的基元，比如三角形或者用户自定义类型。
- Material：材质程序，当 ray 相交与基元相交时执行。
- GeometryInstance：绑定 Geometry 和 Material。
- GroupNode：一系列安排在层次结构 (hierarchy) 中的对象。
- TransformNode：一个层次节点 (hierarchy node) 用于变换几何和 ray。
- SelectorNode：可编程层次节点，用来选择去遍历的子对象。
- AccelerationStructure：绑定到层次节点 (hierarchy node) 的加速结构对象。

这些对象是用 C 语言 API 创建、销毁、修改和绑定的。

组件程序

OptiX 提供的光线跟踪管线包含几个可编程组件，在执行通用的光线跟踪算法期间，在 GPU 上的特定时间段调用这些程序。有以下类型的程序：

- Ray generation programs：光追管线的入口，由每个像素样本或者用户自定义的任务（比如一些自定义的特殊算法）来调用。
- Exception programs：异常控制。
- Closest hit programs：当 ray 找到最近交点时调用，在该程序中执行比如材质着色等程序。
- Any hit programs：当 ray 找到一个新的潜在的最近交点时调用，比如对于计算阴影时有用。
- Intersection programs：实现一个 ray 和基元交点测试，在 traversal 时调用。
- Bounding box programs：计算基元的世界空间包围盒，当构建一个新的加速结构时调用。
- Miss programs：当追踪的光线错过了所有几何结构时调用。
- Attribute programs：当与内置三角形相交时调用，用于为任何命中和最近命中的程序提供三角形特定属性。

这些程序的输入语言是 PTX（parallel-thread-execution）。OptiX SDK 还提供了一组用于 NVIDIA C 编译器（nvcc）的包装类和头文件，可以使用 CUDA C 生成适当的 PTX。

Variable 与执行模型

OptiX 具有灵活而强大的 variable 系统，用于将数据传输到 program。当 OptiX program 引用一个 variable 时，将查询一组被很好地定义的作用域以获取该变量的定义，这将根据查询定义的范围启用变量定义的动态覆盖。

例如，closest hit program 可以引用一个名为 color 的 variable。然后可以将该 program 附加到多个 Material 对象，这些对象依次附加到 GeometryInstance 对象。closest hit program 中的变量首先查找直接附加到其 Program 对象的定义，然后依次查找 GeometryInstance、Material 和 Context 对象。这将使“Material”对象上存在默认颜色定义，但使用该材质替代默认颜色定义的特定实例。

一旦所有这些 objects、programs 和 variables 被组装到有效的 context 中，就可以启动光线生成程序。Launches 采用维度和大小参数，并多次调用光线生成程序，次数等于指定的大小。

一旦 ray generation program 被调用，可以查询特殊的语义变量以提供标识光线生成程序调用的运行时索引。例如，一个常见的情况是启动二维调用，其宽度和高度等于要渲染的图像的大小（以像素为单位）。

3 3　编译的设置项

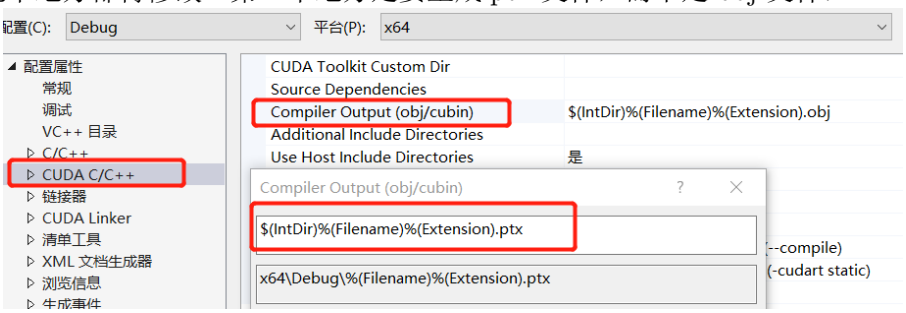
有了前面的基本概念，我们开始设置自己的工程。

本节我们的目标是构建一个不依赖示例中使用的 sutil 中的工具的工程，也就是构建一个最简洁的工程（但我们还需要 OpenGL 来作为显示窗口）。

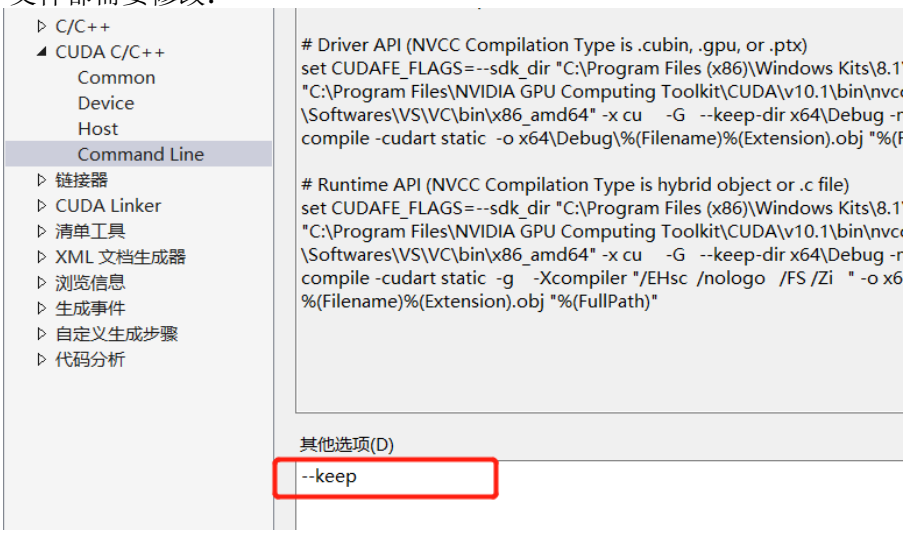
一开始不知道为什么生成.ptx 文件一直有错误，而且资料实在是太少。最终，在 [8, 9] 这两个博客的帮助下终于解决了。

我们需要先能够编译和生成.ptx 文件，右键工程-> 生成依赖项-> 生成自定义，然后选择我们的 cuda。之后右击.cu 文件，选择属性，把.cu 文件都从“不参与生成”改为“CUDA C/C++”。

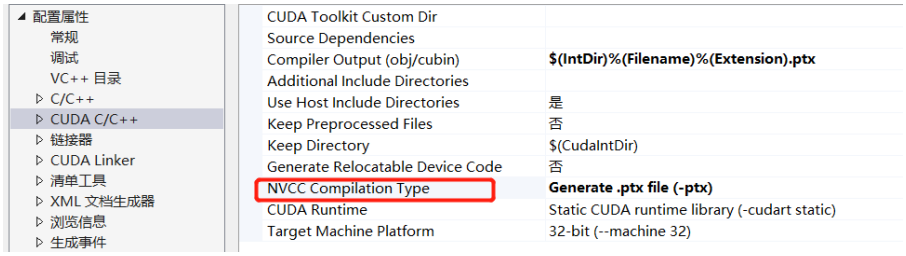
然后下面几个地方都得修改。第一个地方是要生成.ptx 文件，而不是 obj 文件：



第二个地方是保留生成的中间文件，我们需要这里生成的.ptx 文件（相当于 shader），注意每个写光追 shader 的.cu 文件都需要修改：



第三个地方是选择 NVCC 编译类型，从混合编译变为生成.ptx 文件（注意下面的目标机器平台需要设置为自己电脑的相应位数，Win10 都是 64 位的）：



然后注意右键该工程-> 属性->CUDA C/C++，在这里的 Device 中的 Generate GPU debug information 和 host 中 Generate host debug information 都设置为“否”。这一步很关键，如果不设置，则输出的 ptx 文件会非常大，而且无法正确被 Optix 程序读取。

CUDA C/C++ 下的 Common 中的 Generate Relocatable Device Code 也要设置为“否”。

附加依赖项里还要加上 nvrtc.lib 和 cudart.lib。

3 4 集成 OpenGL 显示功能

我们的基本工程文件中虽然已经有了 OpenGL 显示窗口，但这里的显示 Buffer 是 OpenGL 中的，我们希望将 Optix 渲染得到的内容显示到 OpenGL 窗口上，因此需要一些 Buffer 转换的功能。

程序代码见 1-3-2 目录。该目录下有四个文件，其中，main.cpp 和 render_test.cu 是我们自己写的代码，glew.c 和 random.h 是 OpenGL 源文件和生成随机数相关的头文件。

我在代码 main.cpp 和 render_test.cu 中做了很详细的注释，生成和转换 OpenGL 的 Buffer 的代码部分如下：

```
1 Buffer getOutputBuffer();
2 enum bufferPixelFormat;
3 GLenum glFormatFromBufferFormat(bufferPixelFormat pixel_format, RTformat
    buffer_format);
4 void displayBuffer(RTbuffer bufferInput);
```

在 displayBuffer() 函数中，当还没有 OpenGL 的 Buffer 时就先创建。

生成 OpenGL 窗口的函数以及用于窗口交互的函数如下：

```
1 void glutInitialize(int* argc, char** argv);
2 void glutResize(int w, int h);
3 void destroyContext();
4 void registerExitHandler();
```

这些代码都是移植的 sutil.cpp 中的功能，具体实现步骤大家可以不用了解。

3 5 Optix 工程

程序代码仍然是见 1-3-2 目录。

生成 OpenGL 的 Context 的代码：

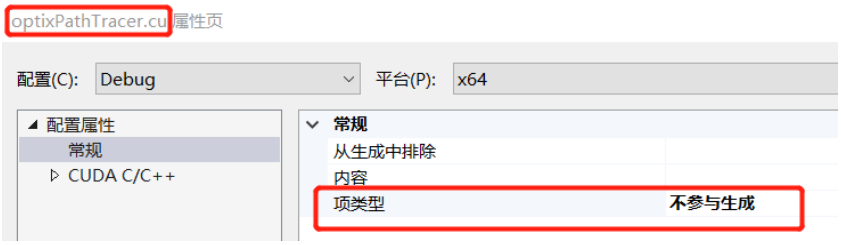
```
1 const char* const Project_NAME = "optixTest";
2 using namespace optix;
3 Context        context = 0;
4 Buffer result_buffer;
5 Buffer getOutputBuffer();
6 void createContext();
```

在 createContext() 函数中，我们把使用的 ptx 文件定义为我们工程生成的 ptx 文件。在 glutDisplay() 中，通过 launch() 函数来调用 Optix 程序在 GPU 中进行渲染。我们不讲解具体意义，大家可以根据我们

实现的结果和手册 [7] 来理解，这个流程挺简单的。（简单提一句，示例中使用的 optix::Context 是面向对象 C++ 版的环境类，还可以直接使用 RTcontext 这种更底层的结构，这也是在使用手册中给出的，不过 optix::Context 对其进行了封装，所以使用更简单。）

四　构建使用 sutil 的工程

我们上一章的构建方式不依赖 sutil，但借助 sutil 可以通过 sutil::getPtxString() 函数直接把.cu 文件转换为.ptx 格式的字符串读取，而不用再生成.ptx 文件。在这种设置下，我们的工程里的.cu 文件就可以设置为“不参与生成”了。



本工程的预定义宏与第二章的工程是相同的，不需要做改动。

我们在本工程下需要的链接库：

```
optix.6.0.0.lib
optix_prime.6.0.0.lib
optixu.6.0.0.lib
nvrtc.lib
cudart.lib
sutil_sdk.lib
```

相比于上一章的代码，我们不需要再假如 glew.c 这个 OpenGL 源文件了，这是因为 sutil_sdk.lib 里包含了 OpenGL 的相关功能函数（已经被封装到了该链接库中）。

这个工程的代码和之前上一章的代码的区别就是在调用 GPU 程序时，本工程使用 sutil 把.cu 文件直接读取然后转换为.ptx 格式的字串并用于程序：

```
const char *ptx = sutil::getPtxString( SAMPLE_NAME, "optixPathTracer.cu" );
```

而上一章的代码中是需要读取生成好的.ptx 文件作为 GPU 程序：

```
std::string ptx = "./x64/Debug/render_test.cu.ptx";
optix::Program ray_gen_program = context->createProgramFromPTXFile(ptx, "
    draw_solid_color");
```

我们在后面讲解 Optix 时，都会按照使用 sutil 的方式来介绍程序。

五　小结

本文初步介绍了 Optix 的编程和使用方法，对于 Optix 开发来说，真的是“万事开头难”，建立开发环境远比学习如何使用它要难得多。

本文没有详细介绍 Optix 的各个模块和功能，主要在于构建开发环境和初步了解和使用 Optix。后面的系列中我会从实例构建入手，通过实例来讲解 Optix 项目的实现方案。

参考文献

[1] https://developer.nvidia.com/rtx/ray-tracing

[2] https://developer.nvidia.com/rtx/ray-tracing/optix

[3] https://developer.nvidia.com/blog/how-to-get-started-with-optix-7/

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

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

[6] https://developer.nvidia.com/designworks/optix/downloads/legacy

[7] https://raytracing-docs.nvidia.com/optix6/guide_6_5/index.html#guide#

[8] https://www.cnblogs.com/chen9510/archive/2019/10/25/11737941.html

[9] https://blog.csdn.net/novanova2009/