LearnOptix 系列 2-Optix 的基本功能与代码结构

Dezeming Family

2023 年 4 月 16 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。

2023=04=18：完成本文第一版。

目录

一　基本工程说明

如果安装有问题，需要从 [?] 中的各个版本的 Release Notes 中查看环境的需求（对 CUDA 以及 CUDNN 的需求，还有对显卡的需求等）。

1 1　SDK 的相关库

在我们默认的安装路径下，下面两个文件夹里的内容共同生成 sutil_sdk 库（sutil_sdk.lib 和 sutil_sdk.dll）：

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

除此之外我们需要的库都是 cuda 或者 optix 的，所有需要的库如下：

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

其中，cuda 目录下的内容主要都是一些头文件和 optix 的.cu 文件，这些头文件有着特殊功能，比如 random.h 可以用来产生随机数。

1 2　我们的第一个示例代码

见我们给出的代码示例 1-1，里面有两个文件，分别是 main.cpp 和 render_test.cu。

我们需要加载.cu 文件在程序内自动转换为.ptx 格式并使用，但是注意在执行这句话时：

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

这里的 sutil::getPtxString 中的 getCuStringFromFile 会从两种地址来搜索文件可能的位置，搜索的优先级是，如果提供了样例名 sample_name（我们知道 SDK 里有很多样例，比如 optixWhitted、optixPathTracer 等；这里的 sample_name 就是我们调用 getPtxString 时的参数 Project_NAME），就从“base_dir/样例名”所在目录中搜索；否则就从“base_dir/cuda”目录下搜索：

```
// Potential source locations (in priority order)
if( sample_name )
    source_locations.push_back( base_dir + "/" + sample_name + "/" +
        filename );
source_locations.push_back( base_dir + "/cuda/" + filename );
```

基目录 base_dir 就是样例的目录，安装时选择默认设置，则该目录就是：

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

因此，我们要把 render_test.cu 放入到对应目录下，两种目录都可以（如果要选择“base_dir/样例名”，就需要在基目录下创建 Project_NAME 目录）。可以改一下 sutil.cpp 源文件的 getCuStringFromFile() 函数使其能够去查找当前工程下的目录：

```
// 添加下句代码后重新编译
source_locations.push_back(std::string("./") + filename);
```

二　第一个示例代码的详细解读

本节详细解读代码示例 1-1。示例代码包含了四个部分：(1) 场景初始化函数、(2) 渲染相关函数、(3)OpenGL 窗口生成和交互函数以及 (4) 生成和转换 Optix 的 Buffer 为 OpenGL 的 Buffer。

2 1　OpenGL 相关功能

OpenGL 会通过 glutInitialize() 函数进行初始化，然后创建环境，之后就调用 glutRun()。

glutRun() 会先进行一些窗口的设置，然后注册各种回调函数，并且将 glutDisplay() 函数进行注册，注册后函数就能够循环不断执行 glutDisplay() 函数了。

glutResize() 函数与窗口大小变化有关，对窗口大小变量 width 和 height 进行调整。

displayBuffer() 是渲染完一帧图像以后调用的，将 Optix 光追管线渲染得到的 buffer 转换为 OpenGL 的 PBO（像素缓冲对象）的 Buffer，用于在 OpenGL 窗口中显示。这些内容我们可以暂时先不过多追究，因为跟 Optix 核心代码无关。

2 2　创建场景

createContext() 就是创建场景的函数。在《Optix 使用入门介绍》一文中我们介绍了“对象模型”、“组件程序”和“Variable”等 Optix 的基本构成，大家如果忘记了可以去回顾一下。

我们简单说一下 RTcontext 和 Context 的区别。它们的定义分别是：

```
1 // c:\ProgramData\NVIDIA Corporation\OptiX SDK 6.0.0\include\optix_host.h
2 typedef struct RTcontext_api* RTcontext;
3 // c:\ProgramData\NVIDIA Corporation\OptiX SDK 6.0.0\include\optixu\
    optixpp_namespace.h
4 typedef Handle<ContextObj> Context;
```

注意上面注释中的 optixu 目录是 OptiX API 的便利函数，ContextObj 内封装了 RTcontext 对象：

```
1 RTcontext m_context;
```

使得操作起来更便利。比如对于 ContextObj::setRayTypeCount() 函数，其实就是调用：

```
1 checkError( rtContextSetRayTypeCount( m_context, num_ray_types ) );
```

OptiX 支持光线类型的概念，这有助于区分为不同目的追踪的光线。例如，渲染器可能会区分用于计算颜色值的光线和专门用于确定光源可见性的光线（阴影光线）。这种概念上不同的射线类型的适当分离不仅增加了程序模块性，而且使 OptiX 能够更有效地操作。不同光线类型的数量及其行为都完全由应用程序定义。要使用的光线类型的数量是通过 rtContextSetRayTypeCount 设置的。

每个 context 可以具有多个计算入口点。context 入口点与单个光线生成程序以及异常程序相关联。给定上下文的入口点总数可以使用下面的函数设置：

```
1 inline void ContextObj::setEntryPointCount(unsigned int  num_entry_points) {
2     checkError( rtContextSetEntryPointCount( m_context, num_entry_points ) )
        ;
3 }
```

每个入口点的关联程序由下面两个函数设置：

```
1 rtContextSetRayGenerationProgram
2 rtContextSetExceptionProgram
```

并由下面两个函数查询：

```
rtContextGetRayGenerationProgram
rtContextGetExceptionProgram
```

在使用之前，必须为每个入口点分配一个光线生成程序；然而，异常程序是一个可选程序，它允许用户在各种错误条件下指定行为。多入口点机制允许在多个渲染算法之间切换，以及在单个 OptiX 上下文上高效实现诸如多遍渲染之类的技术。

对于多入口点，比如在 optixMotionBlur 示例中，就能看到：

```
// raygen, timeview
context->setEntryPointCount( 2 );
context->setRayGenerationProgram( 0, ray_gen_program );
context->setRayGenerationProgram( 1, timeview_program );
```

对于多入口点，我们先不过多讲解，以后遇到再介绍。

2 3 CUDA 函数内的功能

rtDeclareVariable 函数

rtDeclareVariable() 函数的四个参数分别是 (type,name,semantic,annotation)（类型, 变量名称, 语义, 注释）。

rtDeclareVariable 声明指定类型的变量名。默认情况下，变量名将与使用当前程序的查找层次结构 (lookup hierarchy) 在 API 对象上声明的变量相匹配。

type 可以是基元类型或用户定义的结构（详情可见 rtVariableSetUserData）。除了光线有效负载和属性外，声明的变量将是只读的。该变量对当前.cu 文件中定义的所有 cuda 函数都是可见的。

使用 semanticName，该变量可以绑定到内部状态、与射线关联的有效载荷 (payload)，或在 intersection 点（光线与表面的交点）和材质程序之间通信的属性。附加的可选注释 (annotation) 也可以用于将特定于应用程序的元数据 (metadata) 与变量相关联。合法的 semantic 有如下几种，先初步了解一下：

- rtLaunchIndex-启动调用索引。类型必须是 unsigned int、uint2、uint3、int、int2、int3 之一，并且是只读的。
- rtLaunchDim-启动的每个维度的大小。这些值的范围从 1 到该维度中的启动大小。类型必须是 unsigned int、uint2、uint3、int、int2、int3 之一，并且是只读的。
- rtCurrentRay-当前活动的光线，仅当对 rtTrace 的调用处于活动状态时有效。vector 不能保证被归一化。类型必须是 optix::Ray，并且是只读的。
- rtCurrentTime-当前光线时间。类型必须是浮点型并且是只读的。
- rtIntersectionDistance-当前最近的命中距离，仅当对 rtTrace 的调用处于活动状态时有效。类型必须是浮点型并且是只读的。
- rtRayPayload-传递到最近的 rtTrace 调用中的结构，是可读可写的。
- attribute name-从 intersection 程序传递给最近命中或任何命中程序 (closest-hit or any-hit program) 的命名属性。两组程序中的类型必须匹配。此变量在最近命中或任何命中程序中都是只读的，并在 intersection 程序中写入。

关于 attribute name 的一个例子：

```
rtDeclareVariable(float3, geometric_normal, attribute geometric_normal, );
rtDeclareVariable(float3, shading_normal, attribute shading_normal, );
```

在我们的 glutDisplay() 中会不断修改变量 color_set，使颜色不断变化：

```
context["color_set"]->setFloat(color1, 1.0f - color1, 0.0f);
```

rtBuffer

rtBuffer 声明了一个类型为 type、维度为 Dim 的缓冲区。Dim 必须介于 1 和 4 之间（包括 1 和 4），如果未指定，则默认为 1。生成的对象通过 [] 索引运算符提供对缓冲区数据的访问，其中索引为 1、2、3 或 4 维缓冲区的 unsigned int、uint2、uint3 或 uint4。此运算符可用于读取或写入指定索引处的结果缓冲区。

命名缓冲区遵循 rtDeclareVariable 中描述的运行时名称查找语义：

```
// host程序，createBuff程序里设置为2维
result_buffer = context->createBuff(RT_BUFFER_OUTPUT, RT_FORMAT_FLOAT4,
    width, height);
context["result_buffer"]->set(result_buffer);
// device声明
rtBuffer<float4, 2> result_buffer;
```

如果命名缓冲区未绑定到缓冲区对象，或者绑定到类型或维度不正确的缓冲区对象时，将导致编译错误。写入只读缓冲区的行为是未定义的。只有在同一线程以前写入过值的情况下，才能很好地定义从只写缓冲区读取。

rtBuffer 声明必须出现在.cu 文件范围内（不在函数内），并且对同一编译单元内的所有 RT_PROGRAM 实例都可见。

RT_PROGRAM

RT_PROGRAM 定义 GPU 设备内的函数。此函数可以使用 rtProgramCreateFromPTXString 或 rtProgramCreateFROPTXFile 绑定到特定的程序对象，随后将绑定到不同的可编程绑定点。所有程序都应具有“void”返回类型。

对于该函数的参数，注意除了下面两种类型的程序，所有其他程序都不接受任何参数：Bounding box 程序将有一个基元索引和边界框引用返回值的参数（类型为 nvrt::AAbb&）。Intersection 程序将有一个 int primitiveIndex 参数。

```
RT_PROGRAM void intersect(int primIdx){...}
RT_PROGRAM void bounds (int, float result[6]){...}
```

三　第二个 Optix 程序

本节我们做一个简单的任务，在场景中设置一个球，然后显示出来。要想显示几个球，有不少新的内容需要设置，比如我们需要定义一个相机（这里我们根据 Peter Shirley 的方式来定义就好了 [9]）；我们需要定义一个球体类，使得光线可以与之相交。

完整代码见 3-1 目录，大家也可以根据本文自己去写，如果没有修改 sutil.cpp 的搜索目录，那么就别忘了将.cu 文件放在指定的目录下（SDK 的 cuda 目录下）。

3 1　相机的定义

针孔相机其实只需要一个视点以及看向的方向即可。在 createContext() 函数中有相关实现，很简单，因此不再赘述。相应的在 camera.cu 里也有对应的变量和功能。

注意.cu 文件里包含的头文件 helpers.h,这里有一些额外实现的功能,比如 float3 转 uchar4（make_color() 函数）、光线微分、采样 Phong 等。make_color() 函数是 helper.h 里面的一个函数，可以将 float3 类型的输入数据每个位都乘以 255.99 后作为 uchar4 的前三个位，并且 uchar4 的第 4 个位设置为 255u。

3 2 与球体求交

我们与球体求交的代码直接定义在了 camera.cu 里。其实现完全是参考自 [9] 的第一本书的前几个小节，为了让球显示的更好看，我们还添加了 Phong 着色。

由于程序非常简单，所以不再赘述。

四 小结

虽然我们能够用 Optix 来渲染一个球体了，但我们其实本质上并没有借助太多 Optix 的强大功能，比如自动构建加速层次结构、光线递归反弹等。

我们会在下一本小书中开始探索和使用 Optix 的内在光追机制，实现一个 Whitted 光线追踪器。

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://raytracing-docs.nvidia.com/optix6/api_6_5/index.html

[9] https://raytracing.github.io/books/RayTracingInOneWeekend.html