# LearnOptix-v7系列2-熟悉Optix系统

Dezeming Family

2023年8月19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　代码介绍

源码的Common下有三个目录： 3rdParty、gdt和glfWindow。其中，glfWindow是作者封装的glfw显示窗口。3rdParty是第三方库，包括glfw/stb库/ply模型加载工具/obj模型加载工具。 gdt是GPU Developer Toolbox，即开发GPU代码的一些工具。

关于gdt，简单介绍一些文件的含义，其他的以后用到再说：

- random.h里实现了线性同余发生器（Linear congruential generator），简称LCG，是一种能产生具有不连续计算的伪随机序列的分段线性方程的算法，它代表了最古老和最知名的伪随机序列生成器算法之一，其理论相对容易理解，并且易于实现和快速。
- LinearSpace.h包括$2 \times 2$矩阵变换和$3 \times 3$矩阵变换。
- AffineSpace.h包括物体的平移旋转等操作。
- Quaternion.h包括矩阵的四元数操作。
- fixedpoint.h是关于定点数（fixed-point），即约定机器中所有数据的小数点位置是不变的。在计算机中通常有两种简单的约定：将小数点的位置固定放在数据的最高位之前，或者固定在最低位数据之后，一般称前者为定点小数，后者为定点整数。

本文只介绍程序流程，为的是初步了解整个过程（这也是我当时的学习方法，因为Optix 7的细节部分实在是太多了），然后我会在下一篇文章中介绍各种细节和代码原理。本文需要配合源码来阅读，源码部分需要反复观赏以把流程先搞明白。

# 二　ex01:helloOptix

名称空间osc表示Optix Siggraph Course。

这个工程里只有两个文件，optix7.h和main.cpp。optix7.h只有一个函数，就是打印运行时错误的函数。

main.cpp只有一句关键代码：

```
1 // initial optix
2 OPTIX_CHECK(optixInit());
```

optixInit返回的结果是一个OptixResult枚举，该枚举包括各种CUDA运行时的错误信息，除了OPTIX_SUCCESS表示正确执行了函数，其他都是错误信息。

本例程正确运行以后得到的输出显示结果类似于：

```
1 #osc: initializing optix...
2 #osc: found 1 CUDA devices
3 #osc: successfully initialized optix... yay!
4 #osc: done. clean exit.
```

本例子比较简单，但第二个例子和第三个例子的工作量都比较大。

# 三　ex02:pipelineAndRayGen

要想运行Optix程序，你需要一个Optix的Context，pipline，moudle以及SBT。

pipline是由三种程序(OptixProgramGroup)构成的：光线生成程序Raygen/击中基元程序Hitgroup以及与场景没有交点的程序Miss。

moudle是一些设置项，比如最大追踪深度。

第二个例子的主体main()函数比较简单：

```
// define a renderer
SampleRenderer sample;
// set frame buffer size
const vec2i fbSize(vec2i(1200,1024));
sample.resize(fbSize);
// execute rendering process
sample.render();
// load the rendered buffer and save to a image file
std::vector<uint32_t> pixels(fbSize.x*fbSize.y);
sample.downloadPixels(pixels.data());
const std::string fileName = "osc_example2.png";
stbi_write_png(fileName.c_str(),fbSize.x,fbSize.y,4,
               pixels.data(),fbSize.x*sizeof(uint32_t));
```

导出渲染的buffer其实就是调用内存拷贝函数，（其实就是以前的cudaMemcpy，而不是C++标准库的memcoy）。

```
Memcpy((void *)t, d_ptr, count*sizeof(T), cudaMemcpyDeviceToHost)
```

createContext()只初始化编号为0的设备。

## 3 1 一些自定义的类和功能

LaunchParams是一个自定义的结构，定义在LaunchParams.h头文件里，当前只存储了frameID以及buffer的尺寸和数据指针。

CUDABuffer类是一个自定义的结构，定义在CUDABuffer.h头文件里，该类用于申请/释放以及拷贝内存。

SampleRenderer类定义在SampleRenderer.h头文件里，是一个渲染器类。该类的protected函数都是初始化时调用的，用于创建环境并且创建一些程序。

建立管线和发射raygen程序可以分为四大步：

- 创建设备端的 raygen 程序来计算像素颜色。
- 创建Optix的pipeline。
- 创建Shader Binding Table (SBT)。
- 创建frame buffer，并且 launch raygen program。

## 3 2 创建raygen程序

### 3 2.1 raygen程序

CUDA执行的代码在devicePrograms.cu文件中。

在以前的Optix 6中，代码是这么编写的：

```
// "global" variants as parameters to raygen programr
// tBuffer<> type wraps device buffers
rtBuffer<uint32_t> framebuffer;
int2 fbsize;
// program accesses "global" vars:
RT_PROGRAM void raygen(){
```

```
    framebuffer[rtLaunchIndex()] = ...;
}
```

现在变为了（注意再提一句，LaunchParams是用户自己定义的类，用于传递参数，不是Optix内置类）：

```
// only one global: user-supplied LaunchParams struct
extern "C" __constant__ LaunchParams LaunchParams;
// raygen program:
extern "C" __global__ void __raygen__renderFrame()
{
    launchParams.framebuffer[....] = ...;
}
```

我们输出的数据可以直接存在launchParams结构定义的指针里，而不需要再使用rtBuffer¡¿了。

#### 3 2.2 编译.cu文件

添加Cmake规则编译.cu文件，并在二进制文件中嵌入PTX，见example2/CMakeLists.txt:

```
include "cmake/configure_optix.cmake"
cuda_compile_and_embed(deviceCode, "deviceCode.cu")
add_executable(example2
    ${deviceCode}
    ...
)
```

cuda_compile_and_embed提供了规则：

- invokes CUDA compiler to compile devicecode
- embeds generated PTX code in a global string

add_executable中，把生成的嵌入的device code嵌入到二进制文件中（可执行程序）。

### 3 3 创建Optix的pipeline

过程有下面几步：

- 初始化Optix。
- 创建Optix Context。
- 创建Optix Module。
- 设立ProgramGroups。
- 创建管线。

前两步已经介绍过了，第三步步骤如下：

- 设置OptixModuleCompileOptions对象。
- 设置OptixPipelineCompileOptions对象。
- 设置OptixPipelineLinkOptions对象。
- 调用optixModuleCreateFromPTX(...)函数，从嵌入的ptx设备代码中创建Moudle。

第四步见下面这几个函数：

```
SampleRenderer::createRaygenPrograms()
SampleRenderer::createMissPrograms()
SampleRenderer::createHitgroupPrograms()
```

它们都调用了optixProgramGroupCreate(...)函数来创建程序，这些程序就是实现光线追踪的主要代码，不同程序功能定义在OptixProgramGroupDesc的kind成员中：

```
OPTIX_PROGRAM_GROUP_KIND_RAYGEN
OPTIX_PROGRAM_GROUP_KIND_MISS
OPTIX_PROGRAM_GROUP_KIND_HITGROUP
```

第五步创建管线：

```
SampleRenderer::createPipeline()
```

createPipeline是把programGroups对象的数组作为参数，创建得到OptixPipeline对象。

代码中也使用了optixPipelineStackSize(...)函数。如果要查询各个程序组的堆栈要求，会使用optixProgramGroup
使用此信息可计算NVIDIA OptiX程序的特定调用图所需的总堆栈大小。要设置特定管线的堆栈大小，需要使用optixPipelineStackSize。对于其他参数，辅助函数可用于实现这些计算。

### 3 4 创建Shader Binding Table (SBT)

SBT一般可以写为下面的形式：

```
struct __align__( OPTIX_SBT_RECORD_ALIGNMENT ) RaygenRecord
{
  __align__( OPTIX_SBT_RECORD_ALIGNMENT ) char header[
    OPTIX_SBT_RECORD_HEADER_SIZE];
  // just a dummy value - later examples will use more interesting
  // data here
  void *data;
};
```

raygen program，miss program和hitgroup program都有一个SBT结构。

SampleRenderer::buildSBT()用来创建SBT，为raygen program，miss program和hitgroup program都创建SBT。

创建时，还需要使用CUDA将数据传到设备上：

```
// ...
raygenRecordsBuffer.alloc_and_upload(raygenRecords);
// ...
missRecordsBuffer.alloc_and_upload(missRecords);
// ...
hitgroupRecordsBuffer.alloc_and_upload(hitgroupRecords);
```

alloc_and_upload(...)就是把调用CudaMalloc和CudaMemcpy函数。

### 3 5 创建frame buffer，发射raygen program

源码中，frame buffer名为colorBuffer，在resize(...)函数中初始化。回忆一下之前main()函数中的代码：

```
const vec2i fbSize(vec2i(1200,1024));
sample.resize(fbSize);
sample.render();
```

发射部分见SampleRenderer::render()函数。

之前说过，发射参数是一个常量内存：

```
extern "C" __constant__ LaunchParams optixLaunchParams;
```

发射参数/SBT以及图像长宽（意味着线程数）都要被传送到发射函数optixLaunch(...)中。

注意由于发射是异步的(async)，所以需要设置同步，这些实现到了CUDA_SYNC_CHECK()函数中：

```
cudaDeviceSynchronize();
cudaError_t error = cudaGetLastError();
if( error != cudaSuccess ) {
    ...
}
```

## 四 ex03:testFrameInWindow

该工程与上一个工程并没有太多不同，只是调用了GLFW来作为显示器，GLFWindow::run()函数运行每一帧计算，每帧的帧编号在发射参数中获取：

```
const int frameID = optixLaunchParams.frameID;
```

根据帧编号来生成每帧不同的内容。

其他内容与上一个工程基本相同，所以不再赘述。

## 五 小结

本文主要介绍了主要代码流程，在Optix 6系列文章中我们是先介绍的基本知识，然后再串成代码。之所以我先介绍代码，主要是因为Optix 7涉及的细节太多，如果不先把代码跑起来，那么细节上也会很难理解透彻（不过我直到现在也理解的不是很透彻）。

下一篇文章我们主要介绍Optix的各个函数和结构的意义。

# 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/