LearnOptix-v7系列4-简单的场景

Dezeming Family

2023年8月22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 ex04 firstTriangleMesh

有了前面的基础知识，本文的学习会变得容易不少。我们本文一共介绍三个项目：ex04_firstTriangleMesh/ex05_fir
本节介绍ex04。

main.cpp中有两个函数值得注意：一是SampleWindow::render()；二是main()函数中创建三角Mesh和定义相机。当我们用鼠标按下并移动，那么GLFW窗口就会检测并设置相机发生了变动。Camera类有三个成员，分别表示相机位置，相机看向的点，相机的up向量。

要想真正进行射线追踪，我们需要生成真实的射线，并且进行追踪（device上）。并且要有真正的hit和miss程序来处理追踪的射线（device上）。此外，还需要在programs之间进行数据通信交流，因此需要每个射线的数据（Payload，即PRD，在device上）。还需要一些三角形并且构建加速结构。以及合理的program groups以及为mesh和miss program的SBT records（主机上）。

在我们的例子中，Miss program获得背景色，hit program获得每个基元的颜色，然后Raygen program把这些值写到frame buffer中。

1 1 Payload

devicePrograms.cu定义了两个helper函数，unpackPointer()和packPointer()。这两个函数用于将指针进行编码和解码，使得64位的指针可以被放入到两个32位的变量中。这个指针指向的位置是存放Ray的Payload的，在这个简单例子中，Raygen程序定义的Payload只是一个vec3类型的变量：

```
1 __raygen__renderFrame() {
2     // Payload
3     vec3f pixelColorPRD = vec3f(0.f);
4 }
```

也就是比如当光线与物体没有相交时，就给Payload赋值为背景色。然后，pixelColorPRD被编码为两个32位的数：

```
1     uint32_t u0, u1;
2     packPointer( &pixelColorPRD, u0, u1 );
```

然后就会传入到optixTrace(...)函数中作为参数来进行光线跟踪。

注意之前我们在SampleRenderer::createMoudle()中看到的这个2就是表示两个32位的数据：

```
1 pipelineCompileOptions.numPayloadValues    = 2;
```

在Miss program或者Hit program中，会先从Optix系统中获得这两个32位的整数，然后解码为64位的指针，该过程实现在了getPRD()函数里。解码得到指针以后，就可以解释为（转换为）vec3f类型的数据，然后对其赋值。比如：

```
1 extern "C" __global__ void __closesthit__radiance()
2 {
3   const int    primID = optixGetPrimitiveIndex();
4   vec3f &prd = *(vec3f*)getPRD<vec3f>();
5   prd = gdt::randomColor(primID);
6 }
```

1 2 发射参数

现在，发射参数LaunchParams包含了两个新增加的部分：相机参数和加速结构句柄。

相机参数包括相机原点/相机方向/相机（屏幕）竖直方向/相机（屏幕）水平方向，这些参数用于在Raygen程序中针对每个像素生成采样射线。

调用optixTrace(...)函数就是进行递归光线追踪，追踪结束后，就会把得到的pixelColorPRD赋值给framebuffer。

加速结构句柄是：

```
1    OptixTraversableHandle traversable;
```

该句柄会作为optixTrace(...)函数的第一个参数。optixTrace(...)的第二个和第三个参数分别是光线的起始点和方向，第四个和第五个参数分别是光线前进的最大t值和最小t值。其他参数后面用到再介绍。

1 3 三角形加载与加速结构的构建的流程

BLAS全称是Bottom-level acceleration structure。

在main()函数中，初始化了两个立方体盒子。每个立方体盒子都是8个顶点，6个面，由12个三角形构成的：

```
1 TriangleMesh model;
2 // 100x100 thin ground plane
3 model.addCube(vec3f(0.f,-1.5f,0.f),vec3f(10.f,.1f,10.f));
4 // a unit cube centered on top of that
5 model.addCube(vec3f(0.f,0.f,0.f),vec3f(2.f,2.f,2.f));
```

然后main()函数调用SampleWindow的初始化程序：

```
1 launchParams.traversable = buildAccel(model);
2 createPipeline();
3 buildSBT();
4 launchParamsBuffer.alloc(sizeof(launchParams));
```

model就是TriangleMesh对象，里面有两个数组，一个存顶点，另一个存顶点索引：

```
1 std::vector<vec3f> vertex;
2 std::vector<vec3i> index;
```

然后分别加载到两个CUDABuffer里：

```
1 vertexBuffer.alloc_and_upload(model.vertex);
2 indexBuffer.alloc_and_upload(model.index);
```

1 4 三角形加载与加速结构的构建的细节

过程主要涉及下面几个函数：

```
1 SampleRenderer::buildAccel(const TriangleMesh &model);
2 SampleRenderer::buildSBT();
```

buildAccel(...)返回的值给发射参数赋值。

加速结构需要先构建，然后再压缩“拍平”，使得存储空间更小，压缩的内存紧凑模式在遍历时也可以更快。先调用optixAccelBuild(...)函数构建加速结构，然后调用optixAccelCompact(...)函数压缩紧凑。构建时，OptixBuildInput对象triangleInput会被传入到optixAccelBuild(...)函数中作为参数。

关于OptixBuildInput对象triangleInput的SBT的相关内容我们暂时先忽略。

1 5 小结

目前，我们仍然只有一个光线类型，以及一个Mesh，因此只有一个hitgroup record。且我们也只有一个miss program。我们也没有每个program的data。

二 ex05 firstSBTData

本节介绍我们一直忽略的SBT的使用，我们使用SBT records来传递一些数据。

2 1 SBT

HitgroupRecord做了一些修改，添加了一个TriangleMeshSBTData结构，该结构有一个表示颜色的值，一个顶点指针和一个索引指针。

```
struct __align__( OPTIX_SBT_RECORD_ALIGNMENT ) HitgroupRecord
{
  __align__( OPTIX_SBT_RECORD_ALIGNMENT ) char header[
    OPTIX_SBT_RECORD_HEADER_SIZE];
  TriangleMeshSBTData data;
};
```

我们目前只有一种类型的Hitgroup对象（也就是三角形）：

```
int numObjects = 1;
std::vector<HitgroupRecord> hitgroupRecords;
for (int i=0;i<numObjects;i++) {
  // we only have a single object type so far
  int objectType = 0;
  HitgroupRecord rec;
  OPTIX_CHECK(optixSbtRecordPackHeader(hitgroupPGs[objectType],&rec));
  rec.data.vertex = (vec3f*)vertexBuffer.d_pointer();
  rec.data.index  = (vec3i*)indexBuffer.d_pointer();
  rec.data.color  = model.color;
  hitgroupRecords.push_back(rec);
}
hitgroupRecordsBuffer.alloc_and_upload(hitgroupRecords);
sbt.hitgroupRecordBase          = hitgroupRecordsBuffer.d_pointer();
sbt.hitgroupRecordStrideInBytes = sizeof(HitgroupRecord);
sbt.hitgroupRecordCount         = (int)hitgroupRecords.size();
```

为该三角形设置SBT，那么在求交时如果与三角形相交，就可以调用SBT里存储的内容进行一些计算了：

```
extern "C" __global__ void __closesthit__radiance()
{
    const TriangleMeshSBTData &sbtData
    = *(const TriangleMeshSBTData*)optixGetSbtDataPointer();

    // compute normal:
    const int    primID = optixGetPrimitiveIndex();
    const vec3i index  = sbtData.index[primID];
    const vec3f &A     = sbtData.vertex[index.x];
    const vec3f &B     = sbtData.vertex[index.y];
    const vec3f &C     = sbtData.vertex[index.z];
    const vec3f Ng     = normalize(cross(B-A,C-A));

```

```
    const vec3f rayDir = optixGetWorldRayDirection();
    const float cosDN  = 0.2f + .8f*fabsf(dot(rayDir,Ng));
    vec3f &prd = *(vec3f*)getPRD<vec3f>();
    prd = cosDN * sbtData.color;
}
```

2 2 小结

有人可能会好奇发射参数和SBT、Payload之间的使用区别。注意发射参数仅仅在发射时使用，而Payload是在调用optixTrace(...)函数时，任何anyhit或者miss等函数都可以去访问或者修改里面的值，Payload全名叫Ray Payload，也就是采样射线携带的信息。 SBT则可以为不同的对象进行绑定，以便不同的对象去访问属于自己的SBT。

三 ex06 multipleObjects

上个工程我们虽然有两个立方体，但是包含在了一个mesh里。这次我们把它们设置到两个mesh。但我们仍然只用一个Hit program group，也就是说这两个Mesh使用同一个Hit program group（共用同一种closet-hit和any-hit程序）。

3 1 两个mesh

OptixBuildInput对象triangleInput现在变成了一个vector对象，而不再是一个单一的对象。
为每个mesh设置OptixBuildInput对象的参数，然后为这两个mesh计算统一的加速结构。

3 2 设置SBT

现在要为每个对象（其实就是每个mesh）来设置不同的program：

```
int numObjects = (int)meshes.size();
std::vector<HitgroupRecord> hitgroupRecords;
for (int meshID=0;meshID<numObjects;meshID++) {
  HitgroupRecord rec;
  // all meshes use the same code, so all same hit group
  OPTIX_CHECK(optixSbtRecordPackHeader(hitgroupPGs[0],&rec));
  rec.data.color  = meshes[meshID].color;
  rec.data.vertex = (vec3f*)vertexBuffer[meshID].d_pointer();
  rec.data.index  = (vec3i*)indexBuffer[meshID].d_pointer();
  hitgroupRecords.push_back(rec);
}
hitgroupRecordsBuffer.alloc_and_upload(hitgroupRecords);
sbt.hitgroupRecordBase          = hitgroupRecordsBuffer.d_pointer();
sbt.hitgroupRecordStrideInBytes = sizeof(HitgroupRecord);
sbt.hitgroupRecordCount         = (int)hitgroupRecords.size();
```

我们目前只有一个程序组，即hitgroupPGs数组只有一个元素，包括三角形的closet-hit和any-hit程序。

构建的每个mesh都存储在OptixBuildInput对象triangleInput[2]中。每个triangleInput的元素都包含一个SBT record：

```
triangleInput[meshID].triangleArray.numSbtRecords = 1;
```

在构建加速结构时，mesh构建对象，也就是triangleInput，并没有绑定任何SBT，那么当射线和基元相交时，如何得知当前的SBT是哪个呢？甚至，求交以后应该调用哪个closet-hit程序呢？

注意，对于不同的mesh，其获取的SBT指针是不同的，这是Optix自己处理的内容：

```
const TriangleMeshSBTData &sbtData
  = *(const TriangleMeshSBTData*)optixGetSbtDataPointer();
```

mesh如何和SBT相互绑定呢？答案很意外，mesh和SBT并没有相互绑定，而是存在Hitgroup SBT和Hitgroup program之间的绑定关系。在求交以后，物体响应哪个SBT是根据我们在optixTrace(...)中设置的参数来决定的。我们用一小节来好好捋一捋它们之间的关系。

3 3 着色器绑定表

着色器绑定表可以理解为当响应了closest-hit或者any-hit程序以后，执行计算时需要访问的数据，比如使用哪个纹理/访问顶点数组。其实实际上，当物体与射线有交点时，去执行哪个closest-hit或者哪个any-hit程序，时根据物体绑定的SBT来决定的。

在构建加速结构时，每个OptixBuildInput对象（比如代表一个球体，或者代表一个mesh组）都对应一个物体ID。

当我们只有一种发射的射线类型时，每个物体绑定的一个SBT，该SBT对应一个Hitgroup程序就可以了。如果有两种发射的射线类型，而且每个物体都要实现对两种射线类型不同的着色计算，那么就需要两种类型的SBT，分别绑定两个不同的Hitgroup程序。当此时求交时是第一个SBT，那么第一个Hitgroup程序响应；当此时求交时是第二个SBT，那么第二个Hitgroup程序响应。

sbt.hitgroupRecordBase属性设置了Hitgroup的SBT的地址。用以GPU程序响应求交程序后访问SBT。

当设置有两种采样射线类型且需要访问两种不同的SBT时（采样辐射度射线/采样阴影的射线）时，所有OptixBuildInput对象对应的SBT都应该是两个，而不能有的是2个，有的是1个，否则就会在optixTrace(...)时因为SBT stride（optixTrace(...)是对整个场景的所有加载进去的物体进行光追）不一致而导致访问错误。我们再给两个例子加深一下印象。

3 3.1 第一个例子

假设我们只有一种类型的相机采样射线。

假如我们加载了5个meshes，每个mesh都包括不同数量的三角形。

假设初始化了5个hitgroup SBT。因为我们要设置optixTrace(...)的SBT offset为0，且SBT stride为1（每个mesh绑定的SBT数量都是1个），也就是说：第一个mesh相当于绑定了第一个hitgroup SBT；第二个mesh相当于绑定了第二个hitgroup SBT；以此类推。

不同的SBT可以绑定不同的hitgroup程序，通过下面函数即可绑定：

```
optixSbtRecordPackHeader(...)
```

3 3.2 第二个例子

假如我们加载了5个meshes，每个mesh都包括不同数量的三角形。

假设初始化了10个hitgroup SBT，设为HitgroupRecord[10]。我们希望追踪相机射线或者阴影射线时，响应不同的求交程序（closest-hit程序与any-hit程序）。

我们要设置optixTrace(...)的SBT stride为2（每个mesh绑定的SBT数量都是2个），也就是说：

当optixTrace(...)的SBT offset为0时；当相机射线与mesh[0]有交点，那么此时响应的Hitgroup SBT就是HitgroupRecord[0]；当相机射线与mesh[1]有交点，那么此时响应的Hitgroup SBT就是HitgroupRecord[2]；当相机射线与mesh[2]有交点，那么此时响应的Hitgroup SBT就是HitgroupRecord[3]；以此类推。

当optixTrace(...)的SBT offset为1时；当相机射线与mesh[0]有交点，那么此时响应的Hitgroup SBT就是HitgroupRecord[1]；当相机射线与mesh[1]有交点，那么此时响应的Hitgroup SBT就是HitgroupRecord[3]；当相机射线与mesh[2]有交点，那么此时响应的Hitgroup SBT就是HitgroupRecord[5]；以此类推。

第一个mesh相当于绑定了第一个和第二个hitgroup SBT；第二个mesh相当于绑定了第二个和第四个hitgroup SBT；以此类推。根据optixTrace的SBT offset参数来设置meshes应该去响应哪个SBT。

四　小结

本文通过一个比较系统的程序初步认识了Optix的几乎所有组件，但Optix 7可以设置和调节的模块有很多。但我们暂时先不花太多精力去了解，而是根据教程[1]来渲染一些更好看的场景。

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/

[7] https://puluo.top/optix_01/