LearnOptix-v7系列5-更真实的场景

Dezeming Family

2023年8月23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 ex07 firstRealModel

模型读取是根据tiny_obj_loader.obj实现的，Model结构有一个成语：

```
std::vector<TriangleMesh *> meshes;
```

每个TriangleMesh都包含了法向量数组/纹理坐标数组/顶点数组和索引数组。 loadOBJ(...)会加载并且读取模型的各个mesh，然后初始化到数组中。

其他部分和之前并没有本质区别。

二 ex08 addingTextures

本节实现纹理的加载。

2 1 发射参数相关

TriangleMeshSBTData结构增加了一点新的内容：

```
struct TriangleMeshSBTData {
    vec3f  color;
    vec3f *vertex;
    vec3f *normal;
    vec2f *texcoord;
    vec3i *index;
    bool                hasTexture;
    cudaTextureObject_t texture;
};
```

OptixBuildInput对象在构建mesh时，只需要顶点和索引。其他信息，比如纹理或者法向量，与求交等程序无关，因此并不需要加载到OptixBuildInput对象中。但是在一些程序，比如closest-hit或者any-hit中可以通过SBT来访问。

2 2 新增加或修改的函数

为了能够使用纹理，新增加了SampleRenderer::createTextures()函数，以及对SampleRenderer::buildSBT()函数进行了修改。原理也很简单，如果我们要进行纹理访问，那么纹理坐标是其一；其二是我们需要把纹理加载到GPU中；其三是我们需要知道我们访问的是哪块纹理。TriangleMeshSBTData中的纹理对象就记录了SBT对应的纹理。当然一个SBT可以存放多个纹理，比如镜面纹理/漫反射纹理等。

2 3 createTextures()函数

创建纹理比较简单，与OpenGL或者CUDA上基本是完全一致的，比如设置读取方式/插值方式，设置内存拷贝等。

注意有的mesh可能是没有纹理的，所以mesh序号不一定对应于纹理序号。

2 4 buildSBT()函数

需要为有纹理的mesh附加纹理编号：

```
if (mesh->diffuseTextureID >= 0) {
    rec.data.hasTexture = true;
    rec.data.texture    = textureObjects[mesh->diffuseTextureID];
```

```
} else {
    rec.data.hasTexture = false;
}
```

三 ex09 shadowRays

3 1 阴影光线

增加第二种光线类型：

```
enum { RADIANCE_RAY_TYPE=0, SHADOW_RAY_TYPE, RAY_TYPE_COUNT };
```

注意这些枚举类型其实表示偏置，比如阴影光线使用第二个SBT表。

在之前的Raygen程序中，optixTrace有这么一个参数：

```
    OPTIX_RAY_FLAG_DISABLE_ANYHIT
```

这是为了不响应任何any-hit程序。在closest-hit程序中，继续追踪阴影光线的optixTrace(...)函数的参数设置为：

```
OPTIX_RAY_FLAG_DISABLE_ANYHIT
| OPTIX_RAY_FLAG_DISABLE_CLOSESTHIT
| OPTIX_RAY_FLAG_TERMINATE_ON_FIRST_HIT
```

不但不响应any-hit程序，也不响应closest-hit程序。注意一开始Payload设置为vec3f(0.f)，当没有击中任何基元时，响应的Miss program中会给Payload设置为vec3f(1.f)。

当然也可以允许响应anyhit程序，并在anyhit程序中设置光线的终止，只不过我们目前暂时还不用该功能。

3 2 Program的构建

目前我们有两个Miss program，构建在OptixProgramGroup类型的missPGs[2]对象中，以及两个Hitgroup programs，构建在OptixProgramGroup类型的hitgroupPGs[2]对象中。

目前我们的几个prorgam如下，：

```
// hitgroupPGs[0]
void __closesthit__radiance();
// hitgroupPGs[0]
void __anyhit__radiance();
// hitgroupPGs[1]
void __closesthit__shadow();
// hitgroupPGs[1]
void __anyhit__shadow();
// missPGs[0]
void __miss__radiance();
// missPGs[1]
void __miss__shadow();
void __raygen__renderFrame();
```

在上一篇文章（系列4）中，我们介绍了SBT和Hitgroup program以及和基元之间的关系，因此这里就只介绍程序就可以了。

两个Miss SBT使用同样的SBT record结构，即MissRecord：

```
std::vector<MissRecord> missRecords;
for (int i=0;i<missPGs.size();i++) {
  MissRecord rec;
  OPTIX_CHECK(optixSbtRecordPackHeader(missPGs[i],&rec));
  rec.data = nullptr; /* for now ... */
  missRecords.push_back(rec);
}
```

对于Hitgroup程序也会构建两个，分别表示为hitgroupPGs[0]和hitgroupPGs[1]。注意代码中序号0和1就是枚举的RADIANCE_RAY_TYPE和SHADOW_RAY_TYPE进行赋值的。

在SampleRenderer::buildSBT()函数中，对每个mesh都要构建两种类型的Hitgroup program对应的SBT。SBT的编号也是用该枚举赋值的。

hitgroupRecords数组中的元素数量为mesh数乘以射线类型数（因为定义了两种射线，所以就是mesh数乘以2）。因此，关于hitgroup program和hitgroup SBT之间的对应关系为：hitgroupRecords[0]/hitgroupRecords[2]/hitg应了hitgroupPGs[0]；hitgroupRecords[1]/hitgroupRecords[3]/hitgroupRecords[5]...对应了hitgroupPGs[1]。

现在，在Raygen程序中，发射的采样射线optixTrace(...)的程序中的三个参数的意义就很容易理解了：

```
RADIANCE_RAY_TYPE,      // SBT offset
RAY_TYPE_COUNT,         // SBT stride
RADIANCE_RAY_TYPE,      // missSBTIndex
```

对于numMeshes个mesh构成的场景，OptixBuildInput对象triangleInput[]有numMeshes个。SBT和OptixBuildInput对象的绑定是自动的：相机发出的初始射线与物体相交得到最近交点，因为使用的是第0个类型的光线采样，对应了SBT的偏置为0。

如果我们有很多种类的基元（不只是三角形，还包括球形等），我们就需要在closest-hit或者any-hit程序中进行一些特殊处理。但是本文暂不介绍。

当然，在源码中，我们完全不需要两个hitgroup program，一个就足够了。教程最后也没有把第二个hitgroup program利用起来。

四　ex10 softShadows

本节没有什么特别的部分，无非就是定义了一个随机数生成器（每个像素根据其像素坐标来对生成器进行不同的初始种子设置），然后利用Payload将随机数生成器传递给后续射线来使用。

定义了一个四边形，用于采样软阴影。

五　ex11和ex12

因为去噪确实比较简单，所以这两个例子就放在同一个章节下讲。

LaunchParams里多设置了一个albedo输出和一个normal输出，用于记录相机射线遇到的第一个交点处的法向量和albedo：

```
float4    *normalBuffer;
float4    *albedoBuffer;
```

在devicePrograms.cu文件里，Payload定义如下：

```
struct PRD {
```

```
    Random random;
    vec3f  pixelColor;
    vec3f  pixelNormal;
    vec3f  pixelAlbedo;
};
```

发射相机射线以后，在__closesthit__radiance()：

```
prd.pixelNormal = Ns;
prd.pixelAlbedo = diffuseColor;
prd.pixelColor = pixelColor;
```

在Raygen函数里进行赋值。

```
optixLaunchParams.frame.colorBuffer[fbIndex] = (float4)rgba;
optixLaunchParams.frame.albedoBuffer[fbIndex] = (float4)albedo;
optixLaunchParams.frame.normalBuffer[fbIndex] = (float4)normal;
```

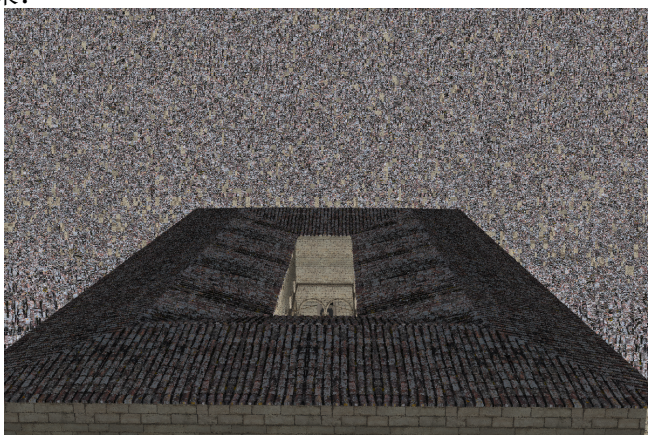
然后去噪的工作都是在下面的函数里实现的：

```
SampleRenderer::render()
```

其实就是把各个buffer丢到去噪器里，得到输出图像，然后再在toneMap.cu文件中进行色调映射得到最终输出。需要注意的是，我们也可以只对没有albedo的成分进行去噪，然后再在toneMap.cu里将albedo与去噪后的illumination部分相乘得到最后结果：

```
float4 f4 = denoisedBuffer[pixelID];
f4 = clamp(sqrt(f4));
f4 = make_float4(
    f4.x * AbbedoBuffer[pixelID].x,
    f4.y * AbbedoBuffer[pixelID].y,
    f4.z * AbbedoBuffer[pixelID].z,
    1.0f);
```

但是会得到这样的结果：

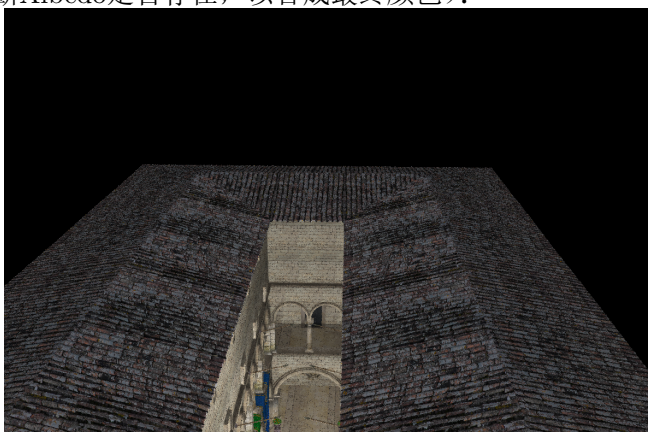


这是因为Payload里的Albedo没有初始化，导致如果没有启动closest-hit程序时，Payload就是没有被赋值过的“脏数据”。初始化一下就好了：

```
PRD prd;
prd.random.init(ix+optixLaunchParams.frame.size.x*iy,
                optixLaunchParams.frame.frameID);
prd.pixelColor = vec3f(0.f);
```

```
// add this to __raygen__renderFrame()
prd.pixelAlbedo = vec3f(0.f);
```

得到（注意此时Miss程序得到的值已经没有什么作用了，可以再增加点代码，比如通过Albedo的第四通道，使得程序可以判断Albedo是否存在，以合成最终颜色）：



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/