LearnOptix 系列 3-Whitted 光线追踪器

Dezeming Family

2023 年 4 月 19 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。

2023-04-23：完成第一版。

目录

一　知识点介绍

注意除了基本的 Optix 函数，optixu 还进行了进一步的封装和优化，我会在本文中详细说明。注意使用 optixu 的函数都是在间接调用 Optix 函数，所以本质上没有任何区别。

1 1　Ray 类型

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。

Ray 的类型数通过 rtContextSetRayTypeCount() 来设定，或者使用 optixu 里的：

```
// 实质也是调用rtContextSetRayTypeCount()函数
context->setRayTypeCount(...);
```

不同的 Ray 类型，可能会有不同的 ray payload、closest-hit programs、any-hit programs 以及 miss programs 属性。

ray payload：ray payload，即射线有效载荷，是与每条射线相关联的任意用户定义的数据结构。例如，它通常用于存储结果颜色、光线的递归深度、阴影衰减因子等。它可以被视为光线在被跟踪后传递的结果，但也可以用于在递归光线跟踪期间，光线生成之间存储和传播数据。

closest-hit programs：在调用 rtTrace 函数后，OptiX 一旦识别出从原点沿射线相交的最近基元，就会调用 closest-hit program。最近命中程序可用于执行基元相关处理，该处理应在光线可见性建立后进行。closest-hit program 可以通过修改每射线数据或写入输出缓冲器来传达其计算结果。它也可以递归地调用 rtTrace 函数。例如，计算机图形应用程序可能使用 closest-hit program 来实现表面着色算法。

any-hit programs：应用程序可能希望对 rtTrace 函数期间沿射线投射发生的任何基元相交执行一些计算，而不是最接近相交的基元；那么可以使用任何 any-hit programs 来实现。例如，渲染应用程序可能需要在每个表面相交处沿着射线累积一些值。

miss programs：当 rtTrace 函数跟踪的光线没有与基元相交时，会调用一个未命中程序 miss programs。未命中程序可以访问用 rtPayload 语义声明的变量，访问方式与最近命中程序和任何命中程序相同。

比如，在 Whitted 风格的光追中，用于计算颜色的光线，Payload 参考下面的 RadiancePL，closest-hit program 会计算颜色、跟踪下一次递归的光线，any-hit program 并不需要执行内容，miss program 采样 Environment map 的辐射度；阴影光线则不同，Payload 参考下面的 ShadowPL，closest-hit program 不执行内容，any-hit program 计算阴影衰减，并根据击中物是否是透明的来选择是否终止光线，miss program 不做任何操作。

```
struct RadiancePL {
    float3 color;
    int recursion_depth;
};
struct ShadowPL {
    float attenuation;
};
```

在调用 rtContextLaunch 时，光线生成程序将 radiance rays 跟踪到场景中，并将传递的结果（在有效载荷的颜色字段中找到）写入输出缓冲区以进行显示：

```
RadiancePL payload;
payload.color = make_float3( 0.f, 0.f, 0.f );
payload.recursion_depth = 0;
// 使用optix::Ray
```

```
optix::Ray ray(ray_origin, ray_direction, RADIANCE_RAY_TYPE, scene_epsilon);
// 追踪光线
rtTrace( top_object, ray, payload );
// 把payload.color结果写入到输出上
......
```

optix::Ray 定义在下面的目录下：

```
OptiX SDK 6.0.0/include/internal/optix_datatypes.h
```

我们需要设置光线的类型，这个类型是跟其能执行的程序相关联的（比如阴影光线就不会递归反射跟踪）。

与 radiance rays 相交的基元将执行 closest-hit program，该程序计算光线的 color，并可能跟踪阴影光线和反射光线。阴影光线部分显示在以下代码片段中：

```
ShadowPL shadow_payload;
shadow_payload.attenuation = 1.0f;
// 创建阴影光线
optix::Ray shadow_ray = optix::make_Ray( hit_point, L, SHADOW_RAY_TYPE,
    scene_epsilon, Ldist );
// 跟踪阴影光线
rtTrace( top_object, shadow_ray, shadow_payload );
// 衰减入射光
float3 rad =
 light.radiance * shadow_payload.attenuation;
// 把贡献加到radiance ray的payload上：
 payload.color += rad;
```

为了正确衰减阴影光线，所有材质都需要有“any-hit programs”程序，该程序可以调整衰减并终止光线遍历。假设材质为不透明材质，则会将衰减设置为 0：

```
shadow_payload.attenuation = 0;
rtTerminateRay();
```

1 2 设备函数 rtTrace

rtTrace() 函数是与追踪光线有关的函数，该函数是在下面的文件内定义的：

```
OptiX SDK 6.0.0/include/optix_device.h
```

是两个模板函数：

```
template<class T>
static inline __device__ void rtTrace(rtObject topNode, optix::Ray ray, T&
    prd, RTvisibilitymask mask=RT_VISIBILITY_ALL, RTrayflags flags=
    RT_RAY_FLAG_NONE);
template<class T>
static inline __device__ void rtTrace( rtObject topNode, optix::Ray ray,
    float time, T& prd, RTvisibilitymask mask=RT_VISIBILITY_ALL, RTrayflags
    flags=RT_RAY_FLAG_NONE);
```

它们分别调用不同的函数（一个携带了 Ray 的当前时间，另一个没有携带 Ray 的当前时间）。

rtTrace 从对象 topNode 处跟踪光线。对 prd（每射线数据）的引用将传递给所有最近命中的程序 (closest-hit) 以及在调用跟踪期间执行的任何命中程序 (any-hit)。topNode 必须引用类型为 RTgroup、RTselector、RTgeometrygroup 或 RTtransform 的 OptiX 对象。

可选的参数 time 设置光线用于运动场景的遍历和着色时的时间。射线时间 (ray time) 在用户程序中可用作 rtCurrentTime 语义变量。如果忽略该参数，则光线将继承触发当前程序的父光线的 time；在没有父光线的光线生成程序中，时间默认为 0.0。

可选的 visibility mask 控制与用户可配置的对象组 (user-configurable groups of objects) 的相交。将 groups 和几何图形的 visibility mask 与此 mask 进行比较。如果两个集合中都存在至少一个同位比特（(group_mask & ray_mask) != 0），则计算交点。注意 visibility 目前仅限于八个组，只考虑 mask 的低八位。

1 3 Entry points

每个 context 可以具有多个计算入口点（就相当于要启动的 GPU 程序）。context 入口点与单个光线生成程序以及异常程序相关联。给定 context 的入口点总数可以使用下面的函数设置：

```
1 rtContextSetEntryPointCount
2 // 使用optixu的函数：
3 ContextObj::setEntryPointCount(1);
```

每个入口点由以下函数设置关联程序：

```
1 rtContextSetRayGenerationProgram
2 rtContextSetExceptionProgram
3 // 使用optixu的函数：
4 ContextObj::setRayGenerationProgram
5 ContextObj::setExceptionProgram
```

并由下面的函数进行查询：

```
1 rtContextGetRayGenerationProgram
2 rtContextGetExceptionProgram
```

在使用之前，必须为每个入口点分配一个射线生成程序；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 context 上高效实现诸如多遍渲染之类的技术。

1 4 其他全局设置

除了光线类型和入口点计数外，OptiX 的 context 中还封装了其他几个全局设置。这些设置很多我们暂时还不会用到，但是大家应该先提前了解一下有哪些功能，有个初步印象即可。

每个 context 都包含许多属性，这些属性可以使用下面的函数进行查询和设置：

```
1 rtContextGetAttribute
2 rtContextSetAttribute
```

例如，可以通过指定下面作为属性参数来查询 OptiX context 在主机上分配的内存量。

```
1 RT_context_ATTRIBUTE_USED_host_memory
```

为了支持递归，OptiX 使用与每个执行线程相关联的一小堆内存。下面的函数可用于设置和查询此堆栈的大小。

```
rtContextGetStackSize
rtContextSetStackSize
```

应小心设置堆栈大小，因为不必要的大堆栈会导致性能下降，而过小的堆栈会导致光线引擎内溢出。堆栈溢出错误可以用用户定义的异常程序来处理。RTX 模式下不支持上述函数，在 RTX 模式下，不是直接设置堆栈大小，而是使用最大递归深度值来估计堆栈大小。下面的函数用于指定和查询最大跟踪递归深度，以及设置可调用程序链（就是说程序递归或者程序 1 调用程序 2、程序 2 又调用程序 3 时）的最大调用深度：

```
// 设置光线可递归跟踪的最大深度，该值只用于计算堆栈大小
rtContextSetMaxTraceDepth
rtContextGetMaxTraceDepth
// 设置可调用程序链的最大调用深度，该值只用于计算堆栈大小
rtContextSetMaxCallableProgramDepth
rtContextGetMaxCallableProgramDepth
```

当然，在一些光线追踪程序中，为了保险起见（难以预料此时运行是不是 RTX 模式），会同时设置：

```
context->setStackSize( 2800 );
context->setMaxTraceDepth( 12 );
```

rtContextSetPrint 函数用于从 OptiX 程序中启用 C 样式打印，使程序更容易调试。下面的 CUDA C 函数进行全局打开或关闭打印：

```
rtContextSetPrintEnabled
```

而下面的函数切换单个计算网格单元的打印：

```
rtContextSetPrinteLaunchIndex
```

打印语句对性能没有不利影响，打印默认是全局禁用的。打印请求在内部缓冲区中进行缓冲，其大小可以通过下面的函数指定：

```
rtContextSetPrintBufferSize
```

此缓冲区溢出将导致输出流被截断。在所有计算完成之后，但在 rtContextLaunch 返回之前，将输出流打印到标准输出。

context 也是 OptiX 变量的最外层作用域。通过下面的函数，声明的变量可用于与给定 context 关联的所有 OptiX 对象（可以通俗的理解为，一个大型程序里通常一个 context 可以关联到很多个.cu 文件，我们希望 context 关联到的这些.cu 文件都能看到该变量）。

```
rtContextDeclareVariable
// 使用optixu的函数：
ContextObj::declareVariable
```

为了避免名称冲突，可以使用下面的函数查询现有变量：

```
rtContextQueryVariable（按名称）
rtContextGetVariable（按索引）
```

并使用下面的函数删除现有变量：

```
rtContextRemoveVariable
```

本部分内容在下一小节会详细介绍。

rtContextValidate 可以在设置过程中的任何时候使用，以检查 context 及其所有相关 OptiX 对象的状态有效性。这将包括检查是否存在必要的程序（例如，几何节点 (geometry node) 的 intersection 程序）、无效的内部状态（如图节点 (graph nodes) 中未指定的子级）以及是否存在所有指定程序引用的变量。在 context 启动时，验证总是会隐式地执行。使用 optixu 时，用如下函数实现：

```
ContextObj::validate()
```

1.5 变量设置

在给 context 中的变量进行设置时，比如创建一个输出 buffer，使用 optixu 的函数可以简化为如下步骤：

```
result_buffer = context->createBuffer(RT_BUFFER_OUTPUT,
    RT_FORMAT_UNSIGNED_BYTE4, width, height);
context["result_buffer"]->set(result_buffer);
```

使用 Optix 底层代码则需要写为：

```
RT_CHECK_ERROR( rtBufferCreate( context, RT_BUFFER_OUTPUT, &buffer ) );
RT_CHECK_ERROR( rtBufferSetFormat( buffer, RT_FORMAT_FLOAT4 ) );
RT_CHECK_ERROR( rtBufferSetSize2D( buffer, width, height ) );
RT_CHECK_ERROR( rtContextDeclareVariable( context, "result_buffer", &
    result_buffer ) );
RT_CHECK_ERROR( rtVariableSetObject( result_buffer, buffer ) );
```

上一小节讲过 rtContextDeclareVariable。注意 context[] 操作符重载是：

```
template<class T>
Handle<VariableObj> Handle<T>::operator[](const std::string& varname)
{
  Variable v = ptr->queryVariable( varname );
  if( v.operator->() == 0)
    v = ptr->declareVariable( varname );
  return v;
}
```

而 declareVariable 其实本质上就是调用：

```
rtContextDeclareVariable
```

上面的 set() 函数其实就是调用：

```
rtVariableSetObject
```

除了给 context 设置变量，还可以单独给一个程序来设置变量：

```
const char *ptx = sutil::getPtxString( "optixHello", "draw_color.cu" );
RT_CHECK_ERROR( rtProgramCreateFromPTXString( context, ptx, "
    draw_solid_color", &ray_gen_program) );
RT_CHECK_ERROR( rtProgramDeclareVariable( ray_gen_program, "draw_color", &
    draw_color ) );
RT_CHECK_ERROR( rtVariableSet3f( draw_color, 0.462f, 0.725f, 0.0f ) );
```

```
5 RT_CHECK_ERROR( rtContextSetRayGenerationProgram( context, 0,
      ray_gen_program ) );
```

其中，下面的调用也可以用 optixu 来实现：

```
1 // Optix函数：
2 rtProgramDeclareVariable
3 // 使用optixu的函数（间接调用Optix函数）
4 ProgramObj::declareVariable
```

再次提醒一句，Context 是 Handle<ContextObj> 的别名，Program 是 Handle<ProgramObj> 的别名。

1 6 缓冲区 Buffers

我们之前已经创建过输出 Buffer 了，本节我们了解一下细节。

buffers（缓冲区）是 optix 在主机和 GPU 上传输数据用的，缓冲区由主机在调用 rtContextLaunch 之前使用函数创建：

```
1 rtBufferCreate
2 // 使用optixu：
3 ContextObj::createBuffer
```

此函数还设置缓冲区类型（枚举 RTbuffertype）以及可选的标志（枚举 RTbufferflag），类型和标志可以按位“或”来组合。标志有：

```
1 // 输入Buffer，只能主机端写入，传输给GPU
2 RT_BUFFER_INPUT
3 // 输出Buffer，只能GPU写入，我们已经用过该类型多次
4 RT_BUFFER_OUTPUT
5 // 主机和GPU都能够写入
6 RT_BUFFER_INPUT_OUTPUT
7 // progressive launch（不是生成单个帧，而是生成多个子帧，每个子帧比如是一个
      像素一条光线采样的结果）的自动更新输出。可以通过网络连接进行高效流式传
      输。
8 RT_BUFFER_PROGRESSIVE_STREAM
```

关于 progressive launch 的内容我们以后再介绍。

Buffer flags 指定某些缓冲区特性（可选的，不一定需要设置标志），并由 RTbufferflag 的字段枚举：

```
1 RT_BUFFER_GPU_LOCAL
2 RT_BUFFER_COPY_ON_DIRTY
3 RT_BUFFER_DISCARD_HOST_MEMORY
4 RT_BUFFER_LAYERED
5 RT_BUFFER_CUBEMAP
```

关于 flag 的具体含义和功能，我们等以后用到再讲（我们本文会用到 RT_BUFFER_GPU_LOCAL），这里只是先增加一下印象。不过提一句，有些 flag 我从没用到过，也没有见到别人用到过，甚至给出的 Optix 官方例子里都没有用到。

在使用缓冲区之前，必须指定缓冲区的大小、维度和元素格式。可以使用下面的函数设置和查询格式：

```
1 rtBufferSetFormat
2 rtBufferGetFormat
```

格式选项由 RTformat 类型枚举。我们可以选择 C 和 CUDA C 数据类型的格式，如无符号 int 和 float3；可以通过选择格式 RT_format_USER 并使用下面的函数指定元素大小，就可以创建任意元素的缓冲区：

```
rtBufferSetElementSize
```

比如我们想把一个光源的信息存储到一个结构里，我们就可以用该方法直接把光源结构作为一个 Buffer，传递到 GPU 里。

缓冲区的大小由如下函数设置，它们也隐式指定维度：

```
rtBufferSetSize1D
rtBufferSetSize2D
rtBufferSetSize3D
```

使用 optixu 时，那么会根据 ContextObj::createBuffer 的参数列表来选择是生成几维的 Buffer，比如参数列表里提供了 width、height 和 depth，那么就调用：

```
rtBufferSetSize3D
```

下面的函数可用于创建或获取 mipmap 中级别图像的大小（给定其级别号）：

```
// 设定
rtBufferSetMipLevelCount
// 获取
rtBufferGetMipLevelSize1D
rtBufferGetmipLevelSize2D
rtBufferGetmipLevelSize3D
```

二　我们的工程和场景的创建

我们的工程可以大致分为下面几个模块，我们按照 Optix 的 optixWhitted 样例来实现：

```
// 创建context的函数
createContext();
// 创建场景几何的函数
createGeometry();
// 设置相机的函数
setupCamera();
// 设置光源的函数
setupLights();
// 验证context是否合规
context->validate();
// 执行渲染循环
glutRun();
```

此外，如果要实现交互功能，比如当鼠标点击界面并移动时，相机也要转动，但为了避免代码较多影响阅读，我们不做这些交互。本文实现的代码见 2-1 目录。

2 1　场景的创建

场景一共有三种 program：

```
// 光线生成并采样的程序
Ray generation program
// 某像素计算出现异常时的处理
Exception program
// 采样光线没有与任何物体相交
Miss program
```

这几个程序定义在 camera.cu 和 constantbg.cu 两个文件里。

我们需要理一下应该实现的代码结构，这种结构以前我也是花了很长时间才弄懂，因为当时网上也没有什么可以参考的资料，现在我将把这整个代码结构进行梳理。这里的描述都是使用 optixu 里的描述，实际执行的 Optix 底层代码用户可以自行查询，函数名之间的对应都很规范。

在我们实现的每个物体基元中（比如 sphere.cu），都要实现下面的函数（注意函数名无所谓，但是参数列表一定要符合如下规定，绑定时才能正确绑定）：

```
RT_PROGRAM void intersect(int primIdx);
RT_PROGRAM void bounds (int, float result[6]);
```

这样，在 cpp 函数里就可以进行绑定：

```
// 生成基元
Geometry sphere = context->createGeometry();
// 设置此基元数量
sphere->setPrimitiveCount( 1u );
ptx = sutil::getPtxString( SAMPLE_NAME, "sphere.cu" );
sphere->setBoundingBoxProgram( context->createProgramFromPTXString( ptx, "
    bounds" ) );
sphere->setIntersectionProgram( context->createProgramFromPTXString( ptx, "
    robust_intersect" ) );
```

以及相应的材质，每个材质都应该实现下面的两个函数：

```
RT_PROGRAM void closest_hit_program();
RT_PROGRAM void any_hit_program();
```

同理，名字无所谓，但是参数必须都是空。这样在 cpp 程序里绑定：

```
Program matl_ch = context->createProgramFromPTXString( ptx, "
    closest_hit_program" );
Program matl_ah = context->createProgramFromPTXString( ptx, "any_hit_program
    " );
Material matl = context->createMaterial();
matl->setClosestHitProgram( 0, glass_ch );
matl->setAnyHitProgram( 1, glass_ah );
```

之后，需要把材质跟基元绑定：

```
std::vector<GeometryInstance> gis;
gis.push_back( context->createGeometryInstance( sphere, &matl, &matl+1 ) );
```

绑定基元和材质以后，就可以设置几何组，并构建加速结构了：

```
GeometryGroup geometrygroup = context->createGeometryGroup();
geometrygroup->setChildCount( static_cast<unsigned int>(gis.size()) );
geometrygroup->setChild( 0, gis[0] );
geometrygroup->setAcceleration( context->createAcceleration("NoAccel") );
context["top_object"]->set( geometrygroup );
```

几何组是用于任意数量的几何实例的容器。使用如何如下函数设置包含的几何体实例的数量和分配实例（这里是 Optix 底层代码，也就是上面的 optixu 函数调用的）：

```
rtGeometryGroupSetChildCount
rtGeometrityGroupSetChild
```

并且还必须使用如下函数为每个几何组分配一个加速结构：

```
// 对应于上面的optixu函数setAcceleration()
rtGeometryGroupSetAcceleration
```

几何图形组的最小示例用例是值为其指定一个几何基元实例（比如上面的程序，我们只设置了一个几何）。

我们暂时不涉及纹理和纹理采样的功能，留在后面的系列文章里介绍，这样会使本文相对比较简单。本小节的内容只要理顺，写 optix 程序就会很轻松啦。

2 2 相机

在 optixWhitted 官方样例下，使用了 Arcball 来进行相机交互，Arcball 是一种 OpenGL 里常用的鼠标交互时的球体转动方式。我们这里不进行交互，只是构建一个简单的相机（相比较 Arcball，我还是比较喜欢 FPS 游戏里的那种交互方式）。

我们把相机的定义和更新放在了下面两个函数里：

```
// 设置相机视点和朝向
void setupCamera();
// 更新相机相关参数（当屏幕长宽变化时，需要再次调用）
void updateCamera();
```

在相机初始化时，这两个函数都需要调用。当屏幕长宽变化时，会再次调用更新相机相关参数的函数。

三　定义物体

本节我们来定义物体，我们有三种物体，一是实心球体，以及一个空心球壳，以及一个地板。本节代码仍见 2-1 目录。

3 1　球体

定义基元需要一些变量声明，比如当前光线变量：

```
1 rtDeclareVariable(optix::Ray, ray, rtCurrentRay, );
```

以及几何法向量和着色法向量（着色法向量，比如使用法向量贴图使得原有的几何法向量进行一些偏移扰动，增加细节感）：

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

在材质程序中，法向量会被用来进行着色。顺便说一句，这里的 attribute 名都是我们自己定义的，并不是 Optix 内置名称（比如 rtCurrentRay）。Optix 一共有五种内置语义 (semantics)：

Semantic name	*Access*	*Description*	*Ray generation*	*Exception*	*Closest hit*	*Any hit*	*Miss*	*Intersection*	*Bounding box*
`rtLaunchIndex`	read only	The unique index identifying each thread launched by `rtContextLaunchnD`.	✓	✓	✓	✓	✓	✓	
`rtCurrentRay`	read only	The state of the current ray.			✓	✓	✓	✓	
`rtPayload`	read/write	The state of the current ray's payload of user-defined data.			✓	✓	✓		
`rtIntersectionDistance`	read only	The parametric distance from the current ray's origin to the closest intersection point yet discovered.			✓	✓		✓	
`rtSubframeIndex`	read only	The unique index identifying each subframe in a progressive launch. Zero for non-progressive launches.	✓	✓	✓	✓	✓	✓	

Table 5 – Semantic variables

其他的语义，比如法向量，要在基元和绑定的材质上都进行声明，这样基元有交点时计算出法向量，然后材质才能使用该法向量进行计算和着色。

定义基元的两大代码要素是包围盒与求交：

```
1 RT_PROGRAM void intersect(int primIdx);
2 RT_PROGRAM void bounds (int, float result[6]);
```

不过，sphere 里定义了两种求交方式，一种是一般的求交，另一种是鲁棒的求交（包含了对浮点数精度的处理，我也不清楚实现细节或者可参考文献，但是不妨碍理解 optix 的代码）。

从 geometry program 报告 intersection 是分两个阶段的过程。如果 geometry program 计算出光线与几何体相交，它将首先调用 rtPotentialIntersection，将确定报告的命中距离是否在与光线关联的有效间隔内，如果相交有效，则返回 true：

随后，几何程序将在调用 rtReportIntersection 之前计算与交叉点关联的属性（法线、纹理坐标等）。rtReportIntersection 的参数是材质序号：

```
1 rtReportIntersection(0);
```

因为一般一个物体只有一种材质，所以材质序号都是 0，如上所示。

如果 rtPotentialIntersection 返回 true，则必须在计算得到属性变量（法向量、纹理坐标等）后调用 rtReportIntersection。此外，属性变量只能在 rtPotentialIntersection 成功返回后写入。

3 2 球壳

球壳的实现见 sphere_shell.cu 文件。

球壳的处理稍微麻烦一点，因为这是一个空心透明的球，在 [9] 中有详细的描述，注意外球的法向量应该朝着球外，内球的法向量应该是朝球内的。这里的代码并不是将球壳作为两个物体来定义，而是作为单独的一个物体，那么它的法向量朝向就应该根据球面是内球还是外球来定义了。

这里会求出两个变量：front_hit_point 和 back_hit_point，这是因为对于玻璃材质，它可能会发生反射，也可能会折射，而表面交点毕竟位置精度有限，所以为了保证交点发出的光线不会再次跟表面相交（比如如果表面交点偏在物体内部，那么发生反射时，可能会再次跟这个球面相交），所以需要对求得的交点加一个微小的偏移量：

```
1 float3 offset = normalize( shading_normal )*scene_epsilon;
2 front_hit_point = hit_p + offset;
3 back_hit_point = hit_p  - offset;
```

3 3 平行四边形

平行四边形的实现见 sphere_shell.cu 文件。

anchor 是平行四边形的一个顶点的位置，这个平行四边形由顶点位置和这个顶点所在的两条边的两个向量 v1 和 v2 来确定。plane 变量是一个 float4 类型的变量，前三个分量表示法向量，最后一个分量是为了计算求交而提前计算的。

这里面的两个变量 lgt_idx 和 lgt_instance 可能是 light_index 的意思，但是又完全没有用到（Optix 样例里也没有使用过这两个值），而且就算注释掉也没有任何问题，我认为可能是平行四边形光源会用到，但是在 PathTracer 里也没有看到使用了该变量。

3 4 关于坐标系的问题

了解过大型渲染器的人应该会知道，一般在渲染中有很多坐标系统，比如全局世界坐标系、物体坐标系、着色空间坐标系（着色时，会把世界坐标下的向量变换到以法向量为 y 轴的坐标系下进行着色计算）等。我曾经写的一个实时医学影像渲染器为了增强交互功能，甚至实现了更多的坐标系。了解坐标系对理解渲染是必不可少的。

目前我们所有的图形都是直接定义在了世界坐标系下。比如球体的球心，就是世界坐标系下的点。而 PBRT 中的方式是球心永远在局部坐标系的 $(0,0,0)$ 处，计算求交时，先讲光线变换到物体局部坐标系下，然后再进行求交。

但是有些时候，我们可能会定义一些变换节点（比如实例化，会用一堆变换节点引用同一个物体）TransformNode。光线在遍历过程中遇到 TransformNode 时会应用投影变换，变换后的光线被认为在对象空间中，而原始光线被认为是在世界空间中。我们暂时不会涉及变换顶点，但要注意的是 Optix 程序中的实现会涉及这方面内容，比如在计算着色时，会将物体局部坐标系下的法向量变换到世界坐标系下：

```
1 const float3 n = normalize(rtTransformNormal(RT_OBJECT_TO_WORLD,
    shading_normal));
```

将局部坐标系下的法向量变换到世界空间，这样就能跟世界空间的光线计算夹角了（着色中的余弦值）。如果没有变换节点，那么其实这一项相当于不起任何作用。

rtCurrentRay 语义在不同的代码里相当于在不同的空间中：

Program type	*Space*
Closest hit	world
Any hit	object
Miss	world
Intersection	object
Visit	object

Table 7 – Ray space

所以说在 closest-hit program 里，需要将法向量变换到世界空间中。而在 intersection program 里，只需要计算光线和物体在局部坐标系的交点即可，因为它们都在局部坐标系里。

我们也可以说，将球体的球心不放在 $(0,0,0)$ 而是放在某个 (x,y,z) 处就是其局部坐标系下的坐标，只不过从局部坐标系转到世界坐标系的变换是单位矩阵变换，因此可以省略这种变换关系。

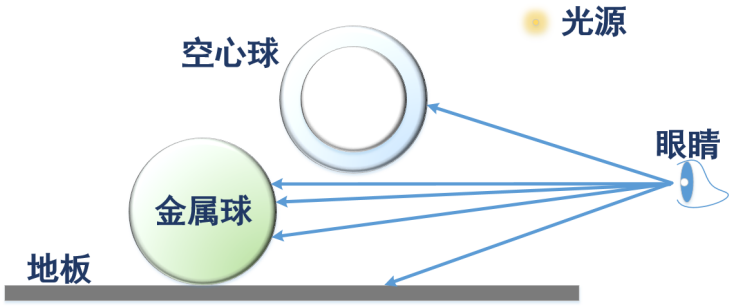
四　定义材质

材质需要符合一套逻辑，理解了这套逻辑，我们才能掌握材质的书写。

什么是材质逻辑呢，这个词是我自己定义的。当我在学习 PBRT 时，发现一个混合了玻璃、金属、次表面散射材质的场景很难去理解渲染流程，尤其是对于双向路径追踪，混合了 delta 材质以后，场景变得非常复杂。采样光线遇到不同的材质会有不同的散射效果，在求阴影时也会变得复杂。而对于一些非物理场景，情况更是如此，因为多了很多近似（从物理的角度来说是错的），所以逻辑上很难理解。比如我们要讲的 Whitted 光线追踪，里面的材质都会进行一些近似。

4 1　材质逻辑简单介绍

我们的场景中一共会包含三种材质，一种是玻璃，一种是金属，还有一种是“地板”。



先考虑最简单的情况，地板是不反光的，因此光线与地板求交以后，直接根据光源位置来计算光照着色即可。

然后稍微复杂一点，光线与金属球有交点，那么就需要先计算 Phong 着色的结果，然后再继续跟踪反射（非物理材质）。计算 Phong 着色时，需要考虑阴影，被不透明物体遮挡的阴影比较简单，我们后面再介绍被透明物体遮挡所产生的阴影的逻辑。

最复杂的情况是玻璃。玻璃既可能发生折射，也可能发生反射，所以要分别把散射部分和折射部分得到的结果相加（所以有可能光线会分叉，一条去追踪折射，另一条去追踪反射）。

4 2　Phong 材质和金属材质

因为本文重点是如何使用 Optix，而不是介绍材质原理，所以材质方面我不会过多去解释。只是说一些逻辑上的东西。

金属反射的效果也是根据 phong 来实现的。phong 材质定义在了 phong.h 和 phong.cu 里，注意 phong.h 只能被.cu 类型的文件包含，因为它全是与 optix 的 GPU 端有关的实现，phong.cu 就是比较基础的 phong 的实现。

光源类定义在了 common.h 头文件里（点光源 BasicLight），并且 phong.h 中有一个 BasicLight 数组，也就是说可以使用多个点光源。我们这里只是提一句光源的定义去哪里找，免得读者不知道光源在哪里。后面的一个小节会专门介绍光源。注意，由于点光源不会被采样到，且点光源不能被用于计算镜面物体的着色，所以玻璃材质的实现中没有包含点光源的位置颜色等信息。

phong 有一个反射参数 Kr，是 float3 类型的。该值只要有一个分量大于 0，就会递归跟踪光线，作为镜面反射。

phong 的遮挡很简单，设置衰减到 0，并且结束光线（见 phongShadowed() 函数）。

4 3　玻璃材质

玻璃材质见 glass.cu。

玻璃既可能发生折射，也可能发生反射，所以要分别把散射部分和折射部分得到的结果相加

对于基于物理的路径追踪来说，被玻璃遮挡和被不透明物体遮挡一样，都被视为直接光照估计为 0，这是因为路径追踪中，被遮挡以后的光确实不算是直接光照了（一两句话也解释不清楚，需要用户了解路

径追踪技术）。

但是对于 Optix 的 Whitted 光线追踪来说，这里还是设置了一个衰减项，使得被玻璃遮挡的阴影比完全遮挡的阴影要亮一点儿。每次跟玻璃表面求交，都会根据计算的值使得光衰减一些：

```
prd_shadow.attenuation *= 1-fresnel_schlick(nDi, 5, 1-shadow_attenuation,
    make_float3(1));
```

4.4 点光源

首先明确一点，Optix 并没有专门识别光源的标志或者语义属性，所有的光源都需要自己去实现。

Optix 官方例程的实现中，面光源和点光源的实现都有，我们这里介绍 Phong 中使用的点光源。点光源以 Buffer 的形式被赋值到 GPU 中（用户实现的 setupLights()）：

```
Buffer light_buffer = context->createBuffer(RT_BUFFER_INPUT);
light_buffer->setFormat(RT_FORMAT_USER);
light_buffer->setElementSize(sizeof(BasicLight));
light_buffer->setSize(sizeof(lights) / sizeof(lights[0]));
memcpy(light_buffer->map(), lights, sizeof(lights));
light_buffer->unmap();
```

而在 phong.h 中的 phongShade() 函数就会根据光源信息来计算着色，所以说，必须将点光源初始化，否则 Optix 程序就找不到 lights 这个参数。

我们在计算阴影遮挡的时候，也得需要知道光源的位置。不过玻璃材质因为不需要计算光源阴影遮挡（因为是 delta 函数材质，从玻璃材质发射出阴影光线采样光源一定采样不到），所以 glass.cu 文件中没有光源的声明。

4.5 材质构建和基元绑定

内容见 main.cpp 的 createGeometry() 函数。

其他过程都比较简单，前面都介绍过了，这里只说一下最后建组的时候：

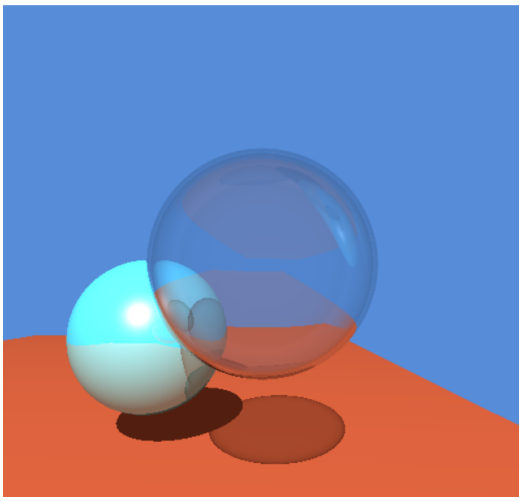
```
GeometryGroup geometrygroup = context->createGeometryGroup();
geometrygroup->setChildCount(static_cast<unsigned int>(gis.size()));
geometrygroup->setChild(0, gis[0]);
geometrygroup->setChild(1, gis[1]);
geometrygroup->setChild(2, gis[2]);
geometrygroup->setAcceleration(context->createAcceleration("NoAccel"));

context["top_object"]->set(geometrygroup);
context["top_shadower"]->set(geometrygroup);
```

top_object 就是相当于构建的加速结构的最顶端，而 top_shadower 是除了光源以外的加速结构的最顶端。因为点光源没有形状结构，所以两个顶端对象都是一样的。如果是存在面光源，那么 top_shadower 就是 top_object 刨除面光源的部分构建的加速结构。

由于场景内容少，所以这里没有构建加速结构，参数为"NoAccel"。

运行，渲染结果是：



我们可以看到边缘有些小锯齿，解决方案有很多，比如超分辨率采样（每个像素采样多个样本然后累加）。我们在下一本小书中会介绍一些改进和优化，渲染出更漂亮的图像。

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