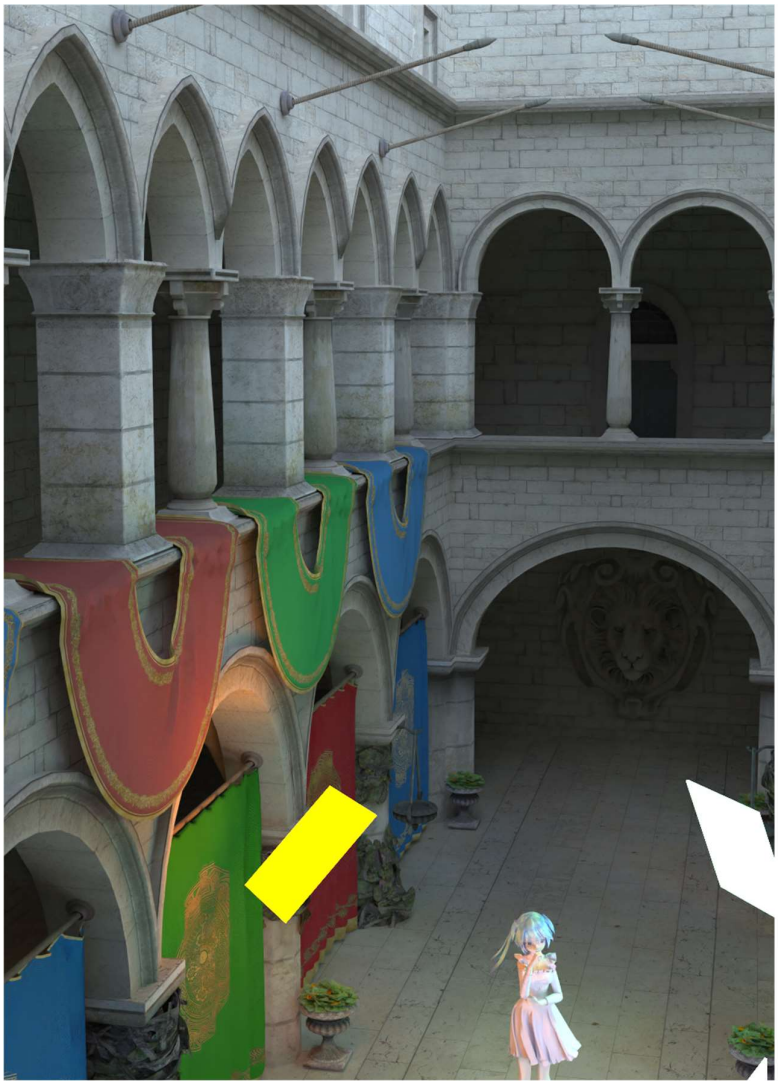
使用 OPTIX（CUDA）构建的路径追踪渲染器

SimplePathTracer_OptixVer

（根据 Ray Tracing in One Weekend Series[1]
使用 CUDA/C++，在 Optix7Course[2] 的基础上修改实现）



开发者： Dofingert （*）

项目简介

本项目（SimplePathTracer）是一个按照 Ray Tracing in One Weekend Series[1] 中的介绍实现的结构简单的高性能路径追踪渲染器。光线追踪是一个并行度非常高的程序，每一条同时发出的光线之间不会有任何交互，故每一条光线都可以进行并行。借助 CUDA，可以充分利用 GPU 的高度并行化对此任务进行加速，利用 Optix 框架编写，可以减少重新构建光线追踪框架的重复工作，并充分利用现代 GPU 上的一些专门为光线追踪设计的硬件单元去加速路径追踪的任务。

设计目标

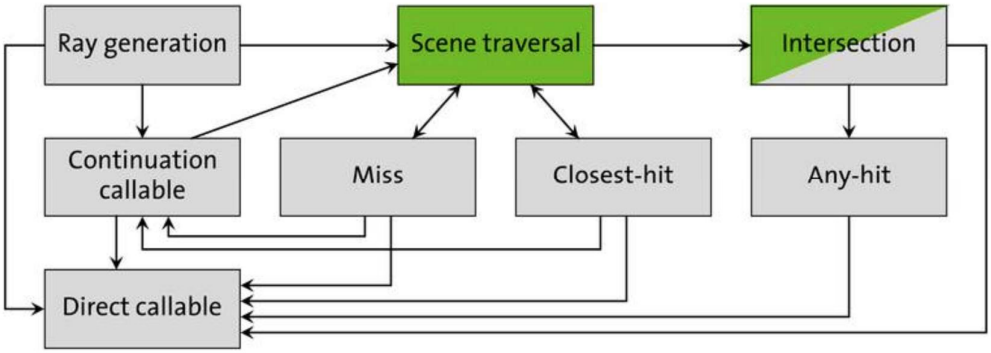
在使用 CUDA, Optix 实现本程序之前, 已经有使用 C++以及 JAVA 语言实现此渲染器。其中，JAVA 版本的渲染器功能更为完善，且支持了三维文件（非标准 Obj 文件及 Mtl 材质描述文件）的 IO，故本程序将以移植 JAVA 版的渲染器中的所有功能，并得到一致结果为基本目标，尝试拓展 GUI 交互功能，尝试提高文件 IO 兼容性，尝试加入骨骼动画的支持，尝试使用 OPTIX 中提供的时域降噪获得更稳定的结果。

使用 Nvidia 的 Optix Api 比较复杂, 有较多的初始化步骤, 以及渲染管线配置上的要求。只看官方的文档，依然觉得这部分实现过于复杂，于是转而寻找相关的实例，之后就找到了在 Siggraph 上的 Optix7Course 讲座以及配套实现的 Optix 框架代码，其中包含了对 Optix 的初始化，管线的创建，一些基础的 Cuda 向量运算的元模板库，文件的 io，以及对于窗口事件的管理。附带一套实现了软阴影的渲染 Shader。这套框架中的这部分代码实际上是所有 Optix 程序基本上都会用到的，没有意义再对其进行重新的编写，故本项目将会以 Optix7Course 中的 ex11 项目作为基础进行后续路径追踪程序的开发。

本程序使用 CUDA 11.4 版本，以及 OPTIX 7.3 版本，在 windows 上使用 VS2019 以及 MSVC 进行构建。

OPTIX 框架理解

OPTIX API 是一款专门为射线追踪任务而制定的一款硬件加速 API, 其中主要使用 CUDA 作为其 Shader 程序的编程语言。这套框架类比于经典的光栅化图形框架中的光栅化 Pipeline, 提供了一套专门应用于射线追踪的 Pipeline。使用这套 API，就类似使用传统的 DirectX API 或者 OpenGL API。这套 Api 需要初始化设备，确定 Pipeline 中的 Shader 程序，给定 Pipeline 中的场景数据描述。对于 Shader 程序，它们是使用 CUDA 编写的，用于描述光线在不同情况下的行为，整体 Pipeline 如下图所示。



其中。灰色部分即为我们可编程的 Shader，分别是 Ray Generation（RG），Miss（MS），Closest Hit（CH），Any Hit（AH），Intersection（IS）这些。对于 Continuation callable，并不在程序的考虑中，对于 Direct Callable，实际上类似于传统 Cuda 程序中的 Device 程序，不需要特别考虑。

Optix 中，支持实现多种类型的光线，每一条光线，可以通过实现其的 AH，CH 这些 Shader 进行表达。每一条光线的这两个 Shader（AH、CH）打包在一起被称作 HitGroup，描述了一个类型的光线在物体表面时的表现。

对于 Optix 这个框架，还有 Shader Binding Table（SBT）的概念，SBT 将场景中的每一个几何体与各种类型的 HitGroup 进行绑定。每个几何体的 SBT 记录中，首先记录了其对于每一种光线所应该使用的 HitGroup Shader 组，以及一些其它的可选信息。在进行渲染时，每条光线与场景几何体中相交时，

利用 SBT 中的自定义信息，可以实现记录实体表面信息的功能，进一步可以实现渲染器的材质系统等等。在本项目中，也将借助 SBT 完成材质部分的存储功能。

具体到 Optix 是如何进行光追渲染的，首先，Optix 会以用户决定的维数（三个维度，一般是渲染的宽度，高度，以及采样数）并行的启动用户所写的 RG Shader，类似于 Cuda 中从主机侧多维度启动核函数那样。在 RG 程序中，将会根据输入的参数表（实际上是一片 Cuda 的 const 数组），以及 RG 程序启动的维度信息，生成光线，并发射到场景中计算每个像素的颜色。光线会在 Optix 内部实现的一套 BVH 加速结构中进行遍历，寻找光线与场景的每一个交点。每次相交，都会运行 AH Shader 并根据 AH Shader 中的设置，决定是否更新光线的 Tmax 信息（Tmax 在遍历过程中，用于记录光线的最近碰撞点信息）。在确定没有与场景更多的交点之后（遍历结束），若没有找到任何交点，则会调用 Miss Shader 程序决定逃逸出场景的光线所应有的颜色，若找到了交点，则会调用 CH Shader，并将最近交点所处的实体信息进行记录，交给 CH Shader 使用。在 CH Shader 中，需要描述光线与场景中离其最近的物体相交之后，发生的过程。对于我们的路径追踪渲染器，此时光线需先依据物体表面的材质信息，决定发生金属反射（镜面反射）或者漫反射，之后，依据每种反射类型，以及概率（通过随机数实现对概率分布的近似）计算出射方向，将出射方向，及出射起点，记录在光线专有的存储数据结构中，供下次 RG Shader 调用使用。

关于颜色的渲染方面，经典的路径追踪使用一个递归的方式，对于每一条追踪光线，本条光线返回的颜色，等于本条光线射中的物体表面的对应某种出射的反照率 RGB 值（即传统概念上的物体本来的颜色）乘以本条光线产生的下一条光线返回的颜色值再加本条光线射中的物体表面的“自发光”材质的色值（RGB）。在 Optix 的框架中，由于 Gpu 栈空间有限等等问题，如果使用递归的方式实现，在光线追踪深度高的情况下，很有可能会出现爆栈的问题。但仔细观察上述表达式，可以发现，实际上完全可以使用一种完全等效的循环替代上述的递归。只需要为每条光线引入两个变量，分别是光强度变量以及光强度乘数变量。每次射

中物体表面时，将物体的自发光色值乘以光线的光强度乘数变量的结果，加到光强度变量上。之后将光线的光强度乘数变量乘以物体表面反照度的值作为新的光强度乘数变量保存。若光线未命中，则在光强度变量中加上未命中时的光线颜色乘以光强度乘数。

使用上述方法，便可以在 RG 程序中，避免 CH 程序内的递归，仅使用一个循环完成光线光照强度的求解。

具体关于程序实现的细节，将在后部分介绍。

实现过程

Step 1

在仔细通读了 Optix 7 Course，并对课程提供的示范代码及库代码进行了一定的熟悉认识之后，我进行了第一次对源代码进行修改的尝试。

本次尝试，在 Optix7Couse/EX11 的基础上进行，不改变原先代码的所有结构，只改变原代码的 Raygen 程序 ClosesetHit 程序，Miss 程序以及部分用于传递参数的结构体。

本次尝试的目标是完成一个仅支持漫反射，以及单色天光的简单路径追踪器。由于目前仅需要支持漫反射以及单色天光，暂时不需要修改材质系统。只需要将原代码中实现的，仅有 DL（Direct Light）直接光照的软阴影算法变成最常见的路径追踪算法即可。

原代码中实现的软阴影，原理比较的暴力。首先从视点向屏幕发射光线（以实现透视视角），光线与场景中最近点相交之后，在交点处向每一个面光源上随机的采样，发出一系列的采样光线。依据采样光线是否能够交到面光源，产生一个光线的“到达率”。此到达率，就表达了原光线交点处位置的光源光照强度。根据光照强度，以及材质的反照率，生成一个合理的色彩值放回到屏幕中。

上述的算法，实际上是仅考虑了一次光照，而没有间接光照的光追算法。对于路径追踪，相比上述方法更为简单暴力，但由于最终会考虑光线的多次反射过程，会以效能为代价，换了真实非常多的结果。

对于路径追踪，本程序重写了部分的 CH 程序，重点节段如下：

```
prd.pixelMuler *= diffuseColor; //Record Albedo Value of the Hit Surface
prd.ori = optixGetRayTmax() * rayDir + prd.ori;
//Hit Point is Next Ray's Original Point
float rdx = prd.random() - 0.5f;
float rdy = prd.random() - 0.5f;
float rdz = prd.random() - 0.5f;
//Generate Random Value from -0.5 to 0.5 for 3DimVector
vec3f rd = normalize(vec3f(rdx, rdy, rdz));
//Generate Normalized Vector(Length == 1.f)
if (dot(rd, Ns) < 0.f) rd = -rd;
//Invert Direction if RandomVector Point Into The Surface
prd.dst = rd;
```

此段程序，将碰撞表面的材质漫反射（Diffuse）色彩信息乘到了此条光线的 PixelMuler 中，并将光线与实体的交点（Hitpoint）作为下一条遍历光线的起点记录。

对于路径追踪程序中的漫反射实现，本质上是求漫反射出射方向的问题。我们知道漫反射光线的出射分别比较随机，但是按找什么分布随机的呢？实际上有两种较为常见的分布拟

合。第一种拟合，认为漫反射出射方向更集中在法线方向上。这种认识与我们日常生活经验比较贴近，即大多数物体从正面看比较亮，而从接近水平的方向看过去却是比较黑的。在 Ray Tracing In One Weekend 中有介绍到这种分布的拟合，拟合的方法非常简单，只需要在物体表面上找到朝外的法向量，在法向量上加上一个球体内的随机单位向量，并将结果标准化即为所需的出射方向。按照此种拟合，得到图 1。

图 1

对于第二种拟合，第二种拟合基于对于漫反射材质的理想模型描述：在均匀的光照下，从各个方向观察，漫反射材质的颜色应保持一致。根据这种模型的描述，无论从哪个方向射入的光线，出射的方向应该随机的分布在一个朝外的半球上。故要实现这种分布，仅需将材质表面向外指的法线与产生的随机向量进行点积，若结果不为正，则将随机向量指向进行反转，即可保证随机向量分布在半球上，满足要求。对于这种模型实现的效果，如下图 2。

图 2

实际上两种模型得到的渲染结果，并没有显著差别，都比较美观，真实。其中，方法一由于漫反射采样方向更侧重法线顶部方向，使得其捕获更多的光线能量，整体场景更为明亮。而方法二则略为暗淡一些。但是，方法一的地面上所展示出的 Gi 效果（两侧彩布漫

反射天光到地板上的光照效果），较为弱，而方法二中则较为明显。这里由于个人偏好方案二得到的画面结果，故选用方案二中作为最终使用的漫反射实现。

Step 2

先前的程序，是以 Optix 7Course/ex11 为基础进行修改的。其中，原教程在 ex11 中介绍的是使用 Nvidia Optix 内部预置的 Ai Denoiser 对图像进行降噪处理。额外的，ex12 中，启用了 AOV 多图层利用多个图层丰富 Ai Denoiser 能够获得的信息，进一步提高降噪效果。在我的尝试二中，将之前能在 Ex11 中运行的代码，进一步的修改到了 ex12 中，以获得更好的降噪效果。改动并没有什么复杂之处，但是收获了意外的性能提高。此外，在 ex12 中，还启用了 HDR 的支持，第一次加入 Gamma 进行色彩矫正以得到正确亮度的图像。使用 ex11 修改的程序得到的降噪前，降噪后对比如图 3，4，使用 ex12 得到的则如图 5，6，两者之间的差异非常令人震惊。

图 3 降噪前画面（1spp 无伽马矫正）

图 4 降噪后画面（1spp 无伽马矫正）

图 5 降噪前画面（1spp）

图 6 降噪后画面（1spp）

可见额外提供给降噪器的反照度信息，法线信息，极大的帮助了降噪器分析光照，得到了非常清晰的画面结果，甚至可用信息远超降噪之前的画面，令人大受震撼。

Step 3

在之前的步骤中，实际上已经得到了一个可用的，有简单漫反射支持的路径追踪渲染器，在步骤三种，将尝试对原有程序的材质系统进行拓展，以支持更多类型的表面。

对于材质的描述，传统的方式是使用 diffuse，reflect（specular），shinning，这些参数，去描述材质表面的漫反射强度，镜面反射（高光）强度，以及高光聚集度。而更现代一些的渲染中则会使用 PBR 材质系统，使用 Albedo 替代 diffuse（实际上可以认为是同一个描述对象）描述物体本色，使用 metallic（金属度）以及 roughness 参数描述材质反射光线的表现。对于 metallic 为 1 的金属，不发生漫反射，仅发射镜面反射，此时粗糙度反应在镜面反射出射光线的集中度上，roughness 参数越大，出射光线越分散。对于 metallic 为 0 的绝缘体，大概率发射漫反射，小概率出现镜面反射（由 roughness 参数控制镜面反射概率）。

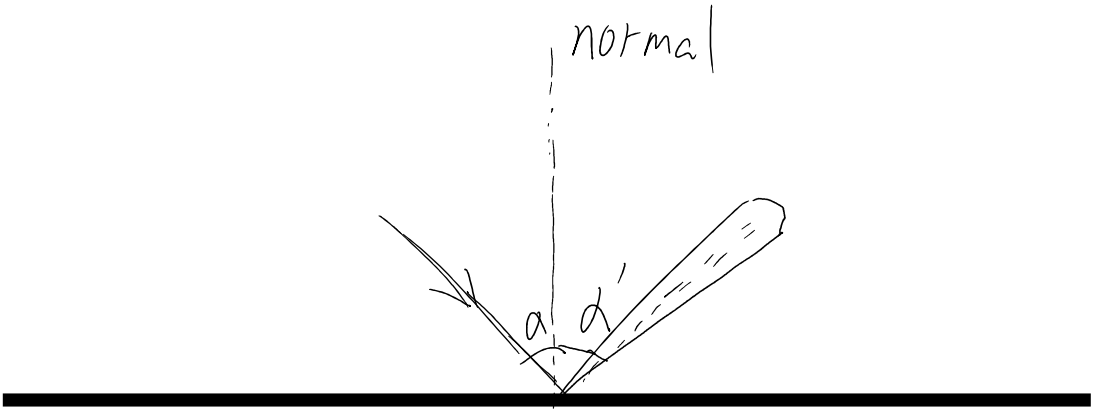
本程序中，使用类 PBR 描述的一套材质系统。在原有材质系统中（仅有 diffuse 一种材质），引入 metallic 与 roughness 两个新贴图，保留 diffuse 贴图，但作为 albedo 使用，控制

材质表面属性。额外加入 emissioin，作为材质自发光的控制项，以实现场景中的光源。若材质 emission 值不为 0，则会对外发出强度为 emission*albedo 大小的光照。如上就实现了本程序需要的所有材质控制系统。

Step4

在先前的准备中，首先实现了能运行的路径追踪程序，之后加入了对于较复杂，较完备的材质系统的支持。在本步中，将以之前为基础，实现镜面反射的新光线产生方法，并引入随机算法，由材质表面定义的概率，确定光线路径，实现最终可用的较为复杂的路径追踪渲染器。

关于镜面反射，实际上对于镜面反射的定义，在初中物理中就有所介绍。入射光与出射光向量在入射点处关于法线镜像对称。



如上述示意图所画，出射光理论上应该完全与入射光对称，但实际上由于表面不平整，法线方向并不是完全的能够确定，故出射光线的分布大致会落在一个束状区域内，只能近似与入射光线对称。在渲染器，为了模拟这种现象，会产生一个单位元内的随机向量，乘以 fuzz（模糊参数），加到理论计算的完美出射光向量（事先已经单位化）上，作为真实猜测的出射光向量，就可以得到非常不错的效果。

具体到实现上，实际上比较简单，仅需要计算好出射理论方向，加上随机值即可。对于理论出射方向的计算，根据对镜面反射的几何约束，很容易得到反射光线方向的计算公式

$$\mathrm{V}_2 = \mathrm{V}_1 - 2(N * \mathrm{V}_1)N$$

式中V_1为入射光线方向向量，V_2为出射方向向量，N为表面法线方向向量，*表示点积。将上述方程用程序实现，即完成了对理论出射光线方向的求解。

对于如何决定下一条的光线是应该发生镜面反射，或是漫反射，首先可以根据材质算出发生镜面反射的概率。在 PBR 的常用实现中，对于绝缘体，有 0.08*(1-roughness)的概率发生镜面反射，而对于金属，则有 1 的概率发生反射。对于 metallic 值处于 0-1 之间的物体，可以使用线性插值算出其发生镜面反射的概率。若不发生镜面反射，则发生漫反射，如此就完成了镜面反射概率的计算。发生镜面反射的概率公式如下：

$$P = 0.08 * (1 - metallic) * (1 - roughness) + metallic$$

产生一个 0-1 内均匀分布的随机数，若大于 P，则发生镜面反射，若小于 P 则发生漫反射，如此便完成了需求。

在程序内实现的核心代码如下：

```
const float posibilityToSpecular = 0.08 * (1 - metalness) *
    (1 - roughness) + metalness;
const bool isDiffuse = prd.random() > posibilityToSpecular;

float rdx = prd.random() - 0.5f;
float rdy = prd.random() - 0.5f;
float rdz = prd.random() - 0.5f;
vec3f rd = normalize(vec3f(rdx, rdy, rdz));
float dotNR = -dot(Ns, rayDir);
vec3f reflectDir = rayDir + 2 * dotNR * Ns;
prd.dst = (isDiffuse) ?
   ((dot(rd, Ns) >= 0.f) ? rd : -rd) :
   normalize(reflectDir + rd * roughness * dotNR);
```

此时，实际上已经得到了一个比较可用的路径追踪渲染器，得到的第一张样图如下。图中测试了低粗糙度的非金属材质，以及自发光材质，高粗糙度非金属，以及金属材质。由于没有对材质进行精细调整，样图并不够美观。

图 7

Step5

发现之前代码中计算反射的部分有一些错误，进行了修正。对于镜面反射，有模糊度项的存在。当入射线角度较大时，出射方向会比较贴近出射平面。如果此时模糊度项较大，则有可能会出现出射线穿过平面的情况，造成错误的反射现象出现。为了纠正这个错误，在前述反射光线计算式中，向随机项中乘以了入射光线与出射光线的点积值，保证出射光线始终在平面上方。

为了展示效果更美观，在三维软件中对之前的 sponza 场景进行了编辑修改，对材质进行了调整，并加入了一个高面数的人物模型，使整体场景面数上升至约 160 万面，对渲

染器能够产生一定的压力。同时，对场景进行了一些比较简单的布光，效果如下图 8，下图 9。

在上述改动之外，还重写了 Miss Shader，将天空的颜色根据光线的角度进行了一次线性映射。

图 8

图 9

Todo List

1. 实现更完整的材质系统，支持法线贴图，bump 等。
2. 实现更完整的摄像机支持，如 DOF（实际比较简单）。
3. 支持重要性采样，或者多重重要性采样，等较为高级的采样技术以提高效能。
4. 优化现有渲染模型，引入菲涅尔项的影响，以帮助金属渲染更真实。加入对折射的支持，利用折射实现更真实的物理效果。
5. 支持光源，而非只能利用物体自发光属性。如面光源，点光源，聚光灯，基于物理的太阳光大气等。
6. 支持 IBL，环境贴图，实现更真实的 SkyBox。
7. 支持骨骼动画或帧动画，以实现动画支持。
8. 换用较新的时域降噪器，以充分利用时域上的信息，避免之前采样结果被浪费造成的大量重复运算。计算 Albedo 通道时，换用新方法以对镜面保持好的兼容性。
9. 优化并行性能，优化数据结构存储方式。
10. 优化 GUI，加入 IMGUI 等 GUI 控件，减少硬编码，以及修改贴图参数时的硬重启。
11. 优化 GUI 性能，提高操作手感。
12. 实现高级效果，如 SSS（次表面散射）。

演示视频

附件中的视频是一段此渲染器的演示视频。视频中使用 1 采样/像素的采样率，2560*1440 的分辨率，6 次最高的迭代深度，分别在开启降噪，不开启降噪，开启采样点积累，不开启积累等多个情况下，进行了性能展示。在测试机环境（RTX3070@95W）下完全可以以交互式的速度进行渲染工作。

编译环境及运行环境

本程序在 VS2019 环境下编译，CUDA 环境为 11.4，Optix 版本为 7.2。根据 Nvidia 官方的说明文档，此程序预先编译好的版本，需要在 R455 及以上版本的驱动程序下运行。