基于蒙特卡罗采样的光线追踪算法的C++实现

**一、程序概述**

本程序是浙江大学研究生课程《计算机图形学》的课程作业，实现了基于蒙特卡罗采样的光线追踪算法。

程序使用Visual studio 2017平台进行编写，使用了DirectX11 API进行图像的计算和渲染，使用DXUT以及对应SDK的工具进行数据的定义、交互和访问，使用DirectX11 中的compute shader来进行GPU对蒙特卡罗采样的加速。

**二、运行程序以及依赖关系**

附加包含目录：

dxSDK\Include

dxSDK\Samples\C++\DXUT11\Optional

dxSDK\Samples\C++\DXUT11\Core

附加库目录：

dxSDK\Lib\x64

dxSDK请从<https://www.microsoft.com/en-us/download/details.aspx?id=6812>下载。

程序运行：关闭Nsight中的TDR → 然后重新生成解决方案 → 开始执行(不调试)。

**三、渲染流程**

1.首先使用TinyObjLoader从obj读取数据到内存里面，之后将数据保存成我们想要的格式，在这个程序中是保存成了一个三角形的数组。

2.之后调用dx11 的API编译hlsl文件，将编译出来的文件保存到computeShaderBuffer里面。

3.申请const buffer， texture buffer， shader\_resource\_buffer， unorderAccessBuffer的空间并且绑定到一张texturebuffer上面

4.将数据传输到const buffer和 shader\_resource\_buffer。

5.开启computer shader的多线程模式，将计算交给computer shader

6.从计算出结果的compute shader中拷贝数据到bmp文件里面。

**四、关键算法**

1.所有的rayTracing操作都在computeShader.hlsl里面进行。

2.根据GPU里面的每一个线程生成线程中的第一个随机数种子。

3.根据摄像机和像素点的位置计算出光线传输的方向和起始点，得到第一根出射光线。

4.将第一根出射光线交给doMCRayTracing，在doMCRayTracing中计算这个出射光线是否和所有的三角形有交点，如果没有交点则返回黑色，如果有交点则根据交点的材质类型计算交点的出射光线（折射、全反射、漫反射）。

5.将这个出射光线迭代入doMCRayTracing，直到超过迭代次数或者交到光源。返回该像素点这次sample 的颜色。

6.多次sample会得到多个颜色，将所有的颜色取平均，得到最后该像素点的结果。

**五、程序缺陷**

1.程序最大的缺陷在于没有编写AABB/K-d tree这些加速结构，在进行光线求交的时候耗费大量的时间，10000个sample需要在显卡上跑一个小时，如果做了加速结构的话，理论上可能会提升10倍左右的效率。

2.注释不完全，没有留有合适的接口，且在GPU里面动用了大量的结构体，不利于程序的维护和效率。

**六、踩坑**

1.DX11 关于structBuffer, unorderAccessView, ShaderResourceView, Texture2D等结构体用法的学习、熟悉和应用。

2.DX11在传入数据的时候constbuffer必须进行16字节对齐，否则GPU获取到的数据会发生错误，有可能是因为在向GPU传输数据的时候是以16字节为块进行传输的，如果使用float3 A, float3 B, float3 C,float3 D这样强行凑成16字节的话，会导致B的前半部分数据在A里面，B数据不完全，C、D以此类推。这一个大坑耗费了大量的debug时间。

3.模型错误，<http://10.76.1.181/courses/graphics/2017/>里面的模型缺少材质，<http://10.76.1.181/courses/graphics/2016/>里面的模型法向量出错，需要手动修改（由右手坐标系改成左手坐标系），因此耗费了大量的debug时间。

4.因为GPU运行由TDR限制，GPU和CPU的响应时间不能超过2秒（或15秒），需要在Nsight中将TDR关掉，从而让GPU可以一直独立运行。

5.compute shader的shader debugger无法在1070显卡上运行，所以compute shader的debug需要输出一个float3的图像，然后根据图像的颜色来判断各个值具体是多少，这一点非常的麻烦。

6.在计算折射的时候，因为光线的方向和ppt里面视角的方向是相反的，所以里面有一些值需要取负值，Ni需要根据实际情况取倒数等。

7.随机数的坑：compute shader里面不支持随机数操作，所以需要自己写随机数，网上找的随机数算法有各种各样的问题。例如比较有名的算法：

float noiseX = (frac(sin(dot(uv, float2(12.9898, 78.233) \* 2.0)) \* 43758.5453));

float noiseY = sqrt(1 - noiseX \* noiseX);

return float2(noiseX, noiseY);

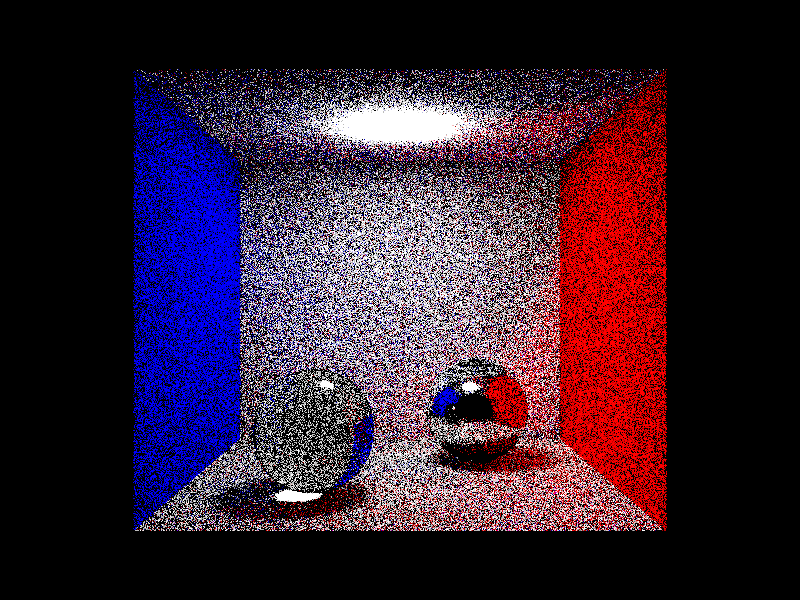
会因为noiseX和noiseY相关，而我们在计算方向的时候需要进行（1-random\*random）这样的计算，所以这种随机数无法使用。类似的算法还有很多，最后我认为效果最好的是C++ stdlib.h里面的rand\_r方法，但是因为我们不想在GPU里面保存全局变量，所以我需要在每次需要生成随机数的时候自己传进去一个种子，而且不能让这些种子之间有相关性。说实话经过实践随机数这方面造成的问题十分严重。而且在GPU里面很难通过调试确定这些种子之间是否相关，所以我在这方面也花了大量的时间研究。

8.绕任意轴旋转的旋转矩阵的生成。这一部分我研究了相对比较长的一段时间，因为我们平时sample出来的方向是在单位半圆上面的，所以需要将他们的y轴旋转到我们需要平面的法线上面。

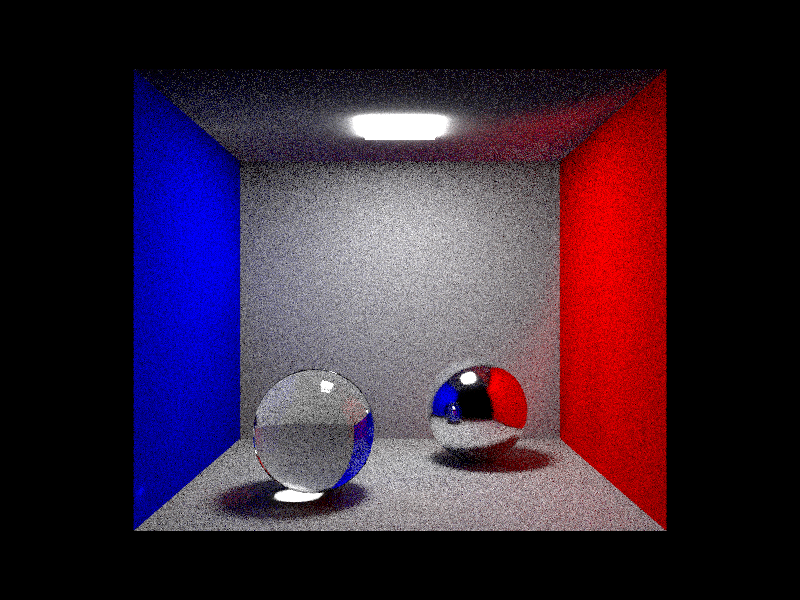
9.GPU中uint到float之间的转换，希望日常使用的时候不要将float转换成uint，如果确实需要的话，一定要进行精确的调试，这个地方很容易出问题。

**七、效果**

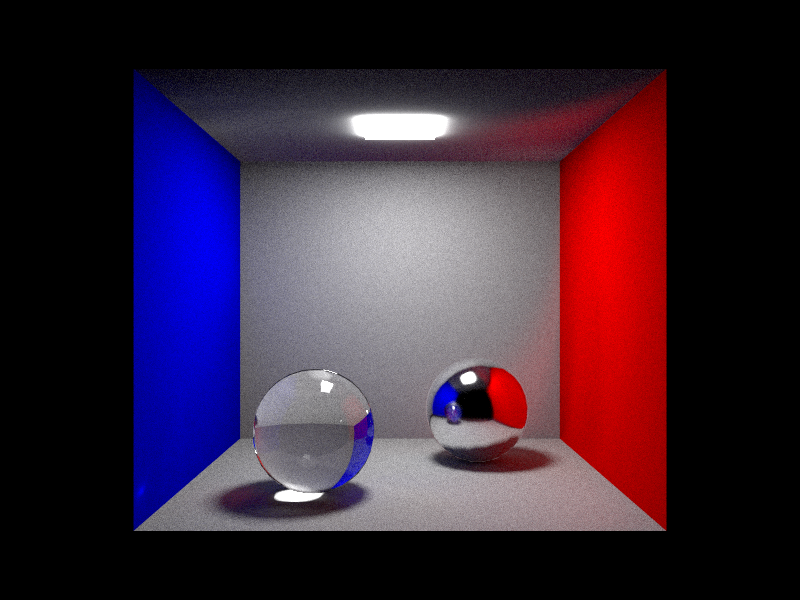
首先是对于第一个场景做100个蒙特卡洛采样：



对于第一个场景做1000个蒙特卡洛采样



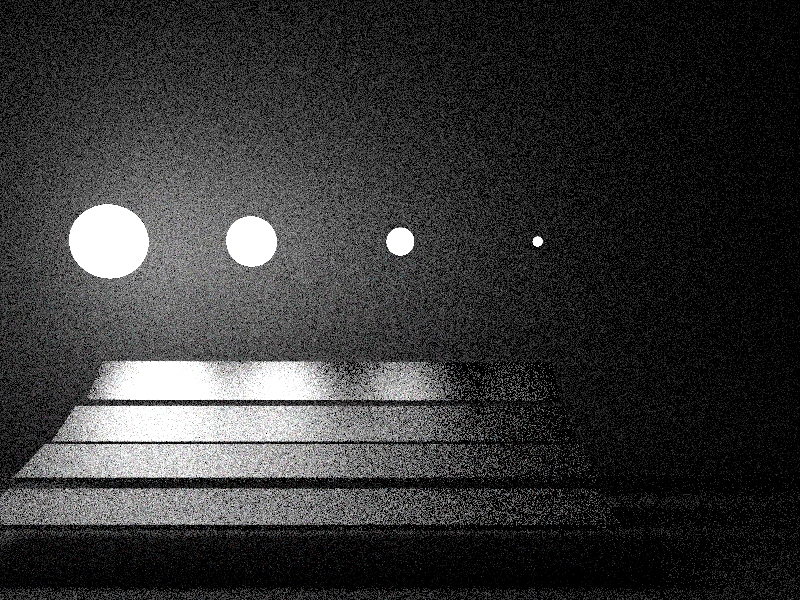
之后是对第一个场景做10000次蒙特卡洛采样，同时迭代次数设置为10次，超过10次就跳出迭代



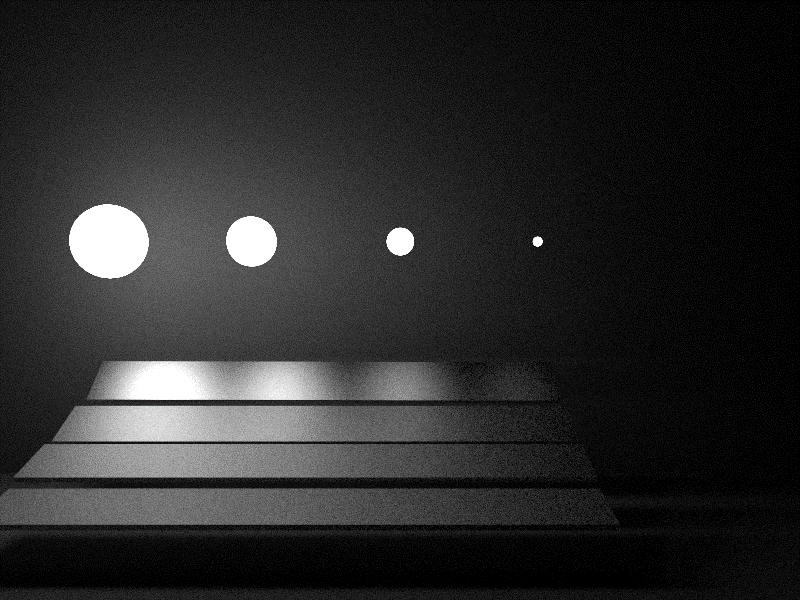
对于第二个场景做100个蒙特卡洛采样，光强度设为4



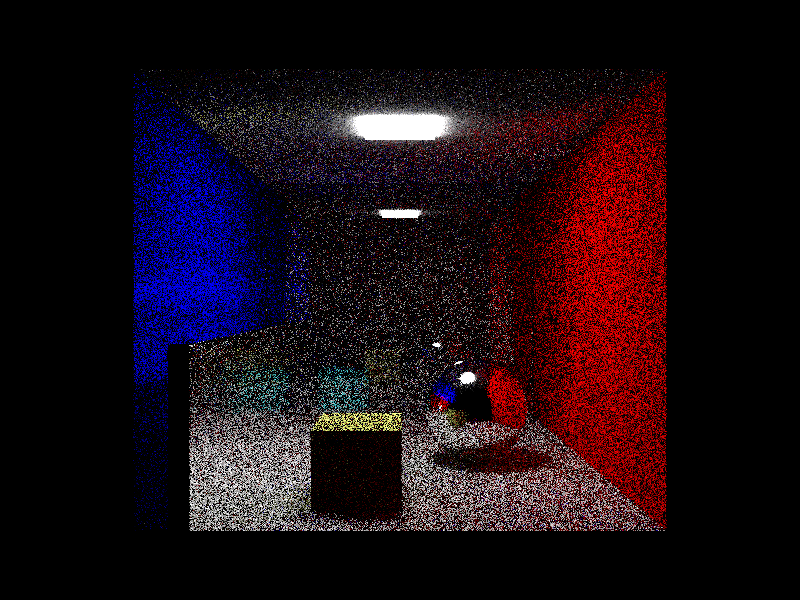
对第二个场景做100次蒙特卡洛采样，光强度设置为10：



对第二个场景做1000次蒙特卡洛采样，光强度设为5



对自己的场景做100次蒙特卡洛采样：



对自己的场景做10000次蒙特卡洛采样：其中自己的场景左边依次是 蓝色漫反射墙 → 全反射墙 → 黄色立方体 → 全反射球 → 红色漫反射墙

从后往前依次是：灰色漫反射墙 → 青色立方体 → 90%的透射墙 →光源

