Project Report for CG100433 Course

Team member

学号	姓名
1753309	毕晓栋
1750761	秦政睿
1751918	温鑫
1752585	齐勖劢
1753070	刘佳伟
1753341	刘啸威

Project Title

基于 GL Pipeline 与光线追踪技术的融合实现的台球模拟器

Abstract

本项目实现了一个台球模拟器。首先我们使用 Box2D 建立了 2 维的物理引擎，算出了当前所有球体的位置，角度，并通过全局变量与其他模块通讯。之后基于这些角度和位置，我们首先用光线追踪的方法还原了场景，又用图形流水线的方法还原了场景。最后，我们将光线追踪计算出的阴影像素值和色彩像素值，作为纹理输入，通过 fragment shader，与原来仅仅只通过了 Phong Shading 和纹理贴图的 fragment 进行像素融合，得到融合后的图片。本项目充分利用到了本学期所学的所有知识：三维运动、光照渲染、光线追踪等技术，同时也尝试对不同图形学模拟方式进行了实现与融合。

（同样，我们的工作开源在了 GitHub 上：https://github.com/ganler/bimulator）

Motivation

台球运动十分受欢迎，台球类的模拟器并不不常见，但基本都是基于二维平面贴图的游戏或者模拟器，本组希望实现对台球系统在特定场景下的光线追踪，并使用 shader 实现一系列特效且与光线追踪的结果相结合，同时编写对应的物理引擎和框架，实现对台球运动的高还原度的实现。这意味着什么呢，**这意味着我们不**

仅要从光追的角度去实现这个模拟器，还要从传统流水线的方式进行实现，最后还需要用到融合算法进行融合。这样做的好处就是能够使用到两者的优点，光线追踪效果惊艳、真实，而传统图形学流水线已有功能丰富，可以利用 shader 完成贴图、几何变换、各种特效等功能。此外，我们搭建的是一个完整的模拟系统，这意味着我们需要完成一个物理引擎和人机交互的部分。

总的来说，本项目难度较大，但也乐趣无穷。但我们分工明确，互相监督，希望借助图形学课程的锻炼，完成这次 hard core 的项目。

The Goal of the project

1. 实现台球系统的物理引擎；
2. 实现人机交互的规则；
3. 实现友好的人机交互界面；
4. 实现球类，吊灯，台球桌和场景的建模；
5. 从 Ray Tracer 的角度实现模拟器渲染，并研究实时算法与代码优化实现；
6. 从传统 GL 流水线的角度实现模拟器渲染，并通过 shader，实现贴图，天空盒，光晕、景深特效等特效；
7. 将 Ray Tracer 与传统 GL 流水线的结果通过融合，打造更加完美的视觉效果，同时也是一次两种技术融合的伟大尝试。

图 1: 系统架构

The Scope of the project

- 我们的光线追踪和特效仅针对台球，吊灯和台球桌面；
- 我们的台球物理运动场景是 2D 的，但渲染效果是 3D 的；

Involved CG techniques

1.光线追踪

- 渲染方程
- Monte Carlo 积分与重要性采样
- 光线求交
- 基于 Microfacet 理论的 BRDF 模型

2.流水线渲染

- 纹理贴图
- 填空盒
- 纹理混合
- 景深特效
  - 小孔成像
  - Accumulation Buffer

3.物理引擎（box2D 实现）

- 动量定理
- 能量定理
- 正碰策略
- 斜碰策略
- 摩擦力策略

Project contents

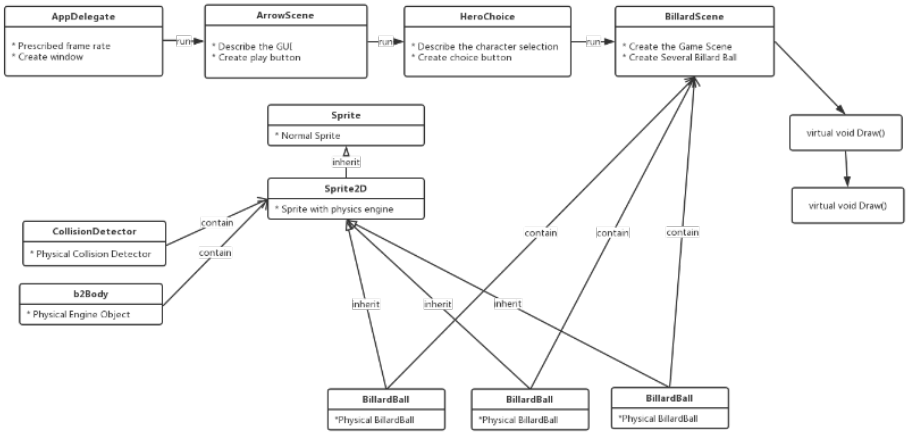
项目是一个台球的模拟器，将实现以下功能：

- 实现台球模拟器的可操作性
- 对台球实现光追特效
- 对台球的运动实现实时渲染

Implementation

1. 总体框架

总体框架如下图所示。



我们在 `BillbardScene` 提供了游戏场景，声明了 10 个带有物理碰撞检测的 `BillbardBall`，实时模拟了台球的位置。

游戏采用第一人称实现，并且监听鼠标和键盘来控制了视点和视角，通过判断视线和台球是否相交来选中台球。关闭了游戏引擎自带的 2D 画面，重载了 `BillbardScene` 的画面绘制函数 `Draw()`， 采用 OpenGL 和 ray tracing 结合的方法，绘制了 3D 画面。

2. 物理引擎

我们使用 cocos2dx-3.17 内置的 Box2D 引擎作为我们的物理引擎。下面将描述我们对物理引擎的使用。

（1）在初始化时，创建 `ContactListener`，之后每一帧，此容器内将保存着正在碰撞的对象；

（2）首先使用_world->CreateBody(&bodyDef) 创建了 Box2D（物理引擎）的场景，然后采用 b2FixtureDef 定义了具有物理特性的 fixtureDef（台球对象），其中 fixtureDef（台球对象）的形状由 b2CircleShape（台球的初始位置，台球的初始半径）所描述；

（3）每一帧，遍历碰撞容器_contactListener，得到所有的碰撞对象 SpriteA 和 SpriteB，分别判断 A 和 B 都是球，A 和 B 一桌一球的情况，然后用 ChangeVehicle 函数，利用动量守恒和能量守恒改写 SpriteA 和 SpriteB 的速度以及下一帧两者的位置；

（4）每一帧更新台球的位置数据后，将所有台球的位置数据传送给 GPU，这样 GPU 可以根据更新后台球的位置重新绘制台球，这样就实现了台球的移动；台球的旋转是用一定的“技巧”实现的，GPU 会记录上一帧和此帧台球的位置，并根据这两个位置信息决定旋转的角度，进行旋转。

3. 光线追踪

这里，我们不会对光线追踪理论的技术细节做过多展开，也不会再对光线追踪理论的流程及方法做过于宏观的介绍，而是针对于我们实时光线追踪场景的技术难点，结合实际情景与软硬件条件，对设计的关键技术与概念加以介绍。

3.1 渲染方程

对于模型表面上的每个点，我们必须在垂直于该点法线方向的半空间上做积分运算以便求出该点的颜色值：

$$L_O = L_e + \int L_i(l) f(l, v) \cos\theta_l dl$$

一个表面的亮度是由两个部分组成的，一部分是表面的自发光 L_e，另一部分是其他表面的射向该表面的光线 $L_i(l)$ 经过 BRDF 作用后的结果。

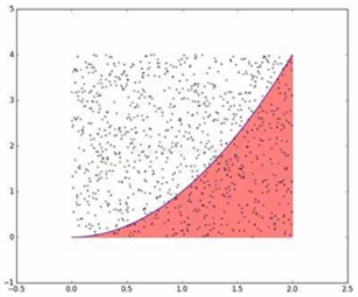
BRDF（Bidirectional Reflectance Distribution Function）即**双向反射分布函数**。它的作用是对于我们给定传入方向 l，计算有多少光在任意给定的方向（观察方向）v 上散射。

在这一方程的基础上，辐射度方法和蒙特卡罗光线跟踪的方法就可以看成是对方程中积分的不同的数值求解方法。

注：真正的渲染方程实际上应该包含折射部分的计算（BRDF + BTDF），而我们这里只考虑反射部分。为了简便计算，我们在实际编程中使用的均为 Blinn-Phong 光照模型。

3.2 Monte Carlo 积分与重要性采样

为了估计某个函数在一个定义域内的积分而在这个定义域内随机地找一些采样点，计算采样点所在位置的函数值，把所有采样点的函数值平均即可得到该积分的估计值，这样的方法叫做**蒙特卡洛**（Monte Carlo）积分。



对于样本的生成最简单的方法就是直接在积分的定义域内生成均匀分布的随机样本，但是对于函数值变化平缓的位置这样生成的样本就会过密，而对于函数值变化剧烈的位置样本数又不足以显示其真实情况。

如果我们先验地知道函数的大致形状，那我们就可以针对性地生成非均匀分布的随机样本，这样能够在相同样本数量的情况下对目标积分得到一个更准确的估计，这就是我们常说的**重要性采样**。

让我们再次观察渲染方程的结构：

$$L_O = L_e + \int L_i(l) f(l, v) \cos\theta_l dl$$

由于对于渲染方程的计算是个递归过程，右侧积分收敛很慢。一个有效的方案就是使用重要性采样，提高采样效率。

注意到右边的积分中有一个因子 $\cos\ x$，这意味着可以在半球上进行带有弦函数权重的概率 $p(w) = \cos\ \theta/\pi$ 来采样。另外还可以对双向反射分布函数 BRDF 进行重要性采样，或在跟踪足够长的路径后，基于光源位置和 BRDF 同时进行采样，使用**多重重要性采样**（MIS）进行混合。

3.3.1 减少采样深度

光线追踪的经典应用场景之一便是多个镜面小球之间的多重镜面反射效果，然而为实现该效果所付出的代价是巨大的，多重的迭代将迅速提升算法复杂度。在我们的场景中，由于是台球游戏，大多数时候球之间都距离较远，因此不需要过多次迭代，或过深的采样便可以得到理想的反射效果。基于第三层采样的贡献值很小的假设，我们将采样深度简化为两层，以提高渲染速度。具体而言：

1. 从视点出发向物体求交
2. 从交点出发采样其他物体的光线

3. 与其他物体相交之后直接对光源采样

从而，我们的渲染效果可以表示为：

$$我们的效果 = 直接对光源采样 + 对其他物体采样 \times BRDF$$

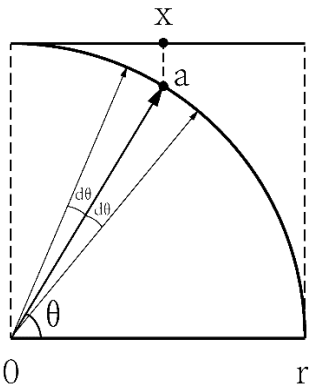
3.3.2 减少采样数

采用重要性采样加快收敛。这里，我们使用余弦函数进行重要性采样。

具体而言，对函数

$$Approx(x) = \frac{1}{N}\sum_{n=1}^{N}\frac{x_n}{pdf(x_n)}$$

我们需要在向量 N 上半球面生成随机的向量 L，使得 $pdf = cos(L,N)$，即 L 与法线夹角的余弦值与随机向量在该方向上的概率密度相等。经观察可知：这是一个在 L 与法向量组成的平面上生成二维向量的问题。



对于在水平区间 $[0,r]$ 上均匀分布的 x，$p(x < x_0) = \frac{x_0}{r}$。$x$ 的落点 a 在四分之一圆上的概率密度：

$$\begin{aligned} p(a) &= \lim_{d\theta\to 0}\frac{pc-pb}{2d\theta} \\ &= \lim_{d\theta\to 0}\frac{r\sin(\theta+d\theta)-r\sin(\theta-d\theta)}{2rd\theta} \\ &= \lim_{d\theta\to 0}\frac{2\cos\theta\sin d\theta}{2d\theta} \\ &= \cos\theta \end{aligned}$$

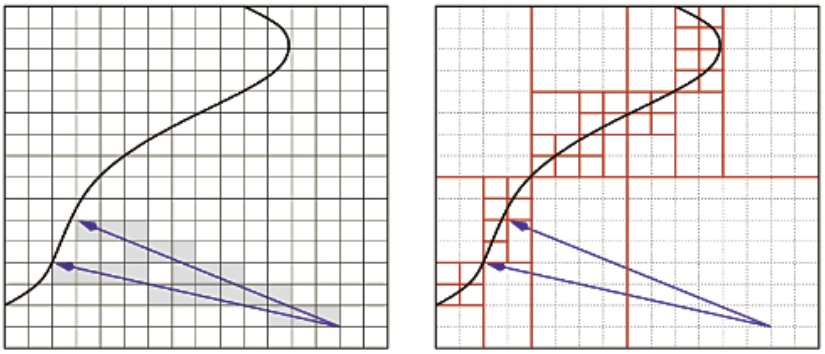
所以，我们只需要：

1. 按照均匀分布生成 r_0
2. 生成一堆随机数 k_1,k_2，满足$k_1,k_2 \in (-1,1], k_1^2+k_2^2=1$
3. $L = k_1w_1 + k_2w_2 + N\cdot h_0$

其中 w_1, w_2 为任意两个与 N 互为两两垂直的单位向量（考虑到 N 的任意性，如果先生成随机向量再旋转到 N 的方向，显然旋转的成本是很大的）。考虑到我们通常确定一个法向量后要生成大量随机向量进行采样，如果重用 w_1, w_2，则只需要计算向量的数乘与加法。

3.3 光线求交

在光线追踪过程中，从眼睛发出的光线与 3D 模型的三角面求交是一个复杂问题。通常精致的 3D 模型可能由几十万至上百万三角面构成，如果采用穷举法求交点，其时间复杂度将是 O(n), 过于复杂。为了减少不必要的求交检测，应采用空间划分技术，最常用的是平衡 k-d 树算法，提高求交检测的效率。



在计算机科学里，k-d 树（ k-维树的缩写）是在 k 维欧几里德空间组织点的数据结构。

k-d 树是每个节点都为 k 维点的二叉树。所有非叶子节点可以视作用一个超平面把空间分割成两个半空间。节点左边的子树代表在超平面左边的点，节点右边的子树代表在超平面右边的点。

我们尝试了很多用于排除模型的数据结构（如 k-d 树、AABB 包围盒法等），当模型很少时（例如只有少数台球物体及单点光源的场景），使用多余的数据结构会带来很大的常数级复杂度，因此我们必须采用常数很小的优化方法，针对具体场景采取相应的优化方法：

1. 对于水平面场景：直接按照 z 轴判断相交即可
2. 球体场景：此时，可能相交的最近距离为 $l - r$。当 $l - r <$ 已有交点的距离，则省去求交步骤：

$$\begin{aligned} L &= Length(O - P) \\ l &= Dot(O - P, dir) \\ h &= \sqrt{L^2 - l^2} \end{aligned}$$

此时，若 $h - r$ 大于目前已经求得交点的距离，则即便相交也不会得到更近的交点。这个优化可以有效减少大部分距离较远的球体求交中的根号运算和除法运算。

3.4 软阴影

理想点光源产生的阴影会有锐利而对比分明的阴影，这与实际场景中柔和的阴影效果相违背。这是由于现实中的光源作为物理实体都具有一定的体积，因此不存在完美的点光源。

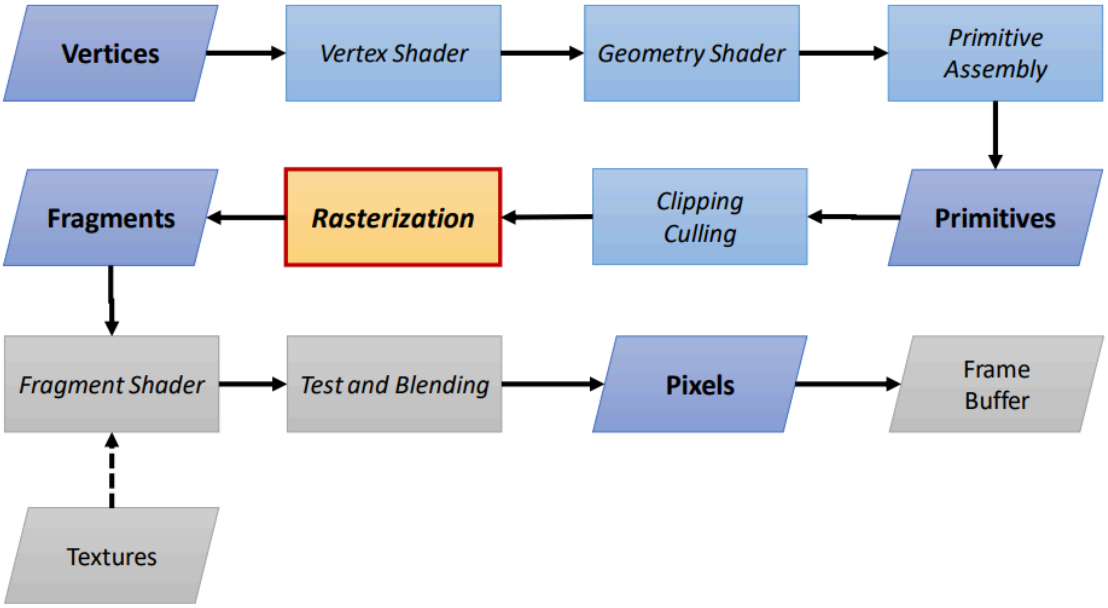
为了模拟软阴影的效果，我们使用 PCS 方法对理想点光源加以随即抖动。具体而言，我们首先向光源发出射线，与其他物体求交。然后向光源采样 N 次，假设其中 n 次相交，$N-n$ 次不相交，则：

$$Res = \frac{n}{N} \times phong\_shading$$

其中 Res 代表采样得到的阴影系数。在实验中，产生的阴影符合一般规律，即距离物体越近越瑞丽，越远越模糊。采样数越多，得到的软阴影越自然、平滑。但使用这个方法会带来每个像素额外的求交运算，我们还未尝试其他通过减少采样数提高速度的经验模型和排除软阴影区域的方法。

4. 图形学流水线

（1）整体的心路历程是这样的，我们首先用 SimpleGL（自研 wrapper）写了一份代码，然后用 clang 作为编译器跑得风生水起，然后准备移植到 MSVC 上时，发现 MSVC 对于模板的支持实在是弱鸡的不行，所以无奈之下，我们首先转为了原始的 OpenGL 代码，但后面由于要和 Cocos 融合，无奈之下又照着 Cocos 对应 OpenGL 的接口写了一遍，最终才跑通。



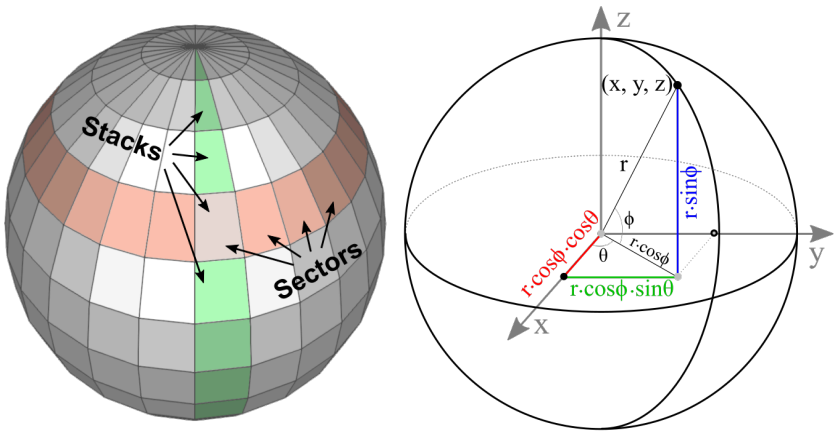
（2）首先，我们在 geometry 这一块的建模，被建模的物体有：

1　球体；

2　围栏；

3　桌面；

其中对于球体的建模，我们将其分割成矩形，然后矩形内部又可以将其划分为三角形。沿着纬度我们分多个 stacks，沿着经度我们划分了多个 sectors。如下图（图源为 http://www.songho.ca/opengl/gl_sphere.html）所示：



其中对于球类建模的一个小优化就是将顶部和底部的矩形，只画一个三角形。因为顶部和底部的矩形都有 2 个重合点，故实际上就只需画一个三角形就好。

```cpp
template<typename Iter, typename ... Args> /* continuous assignment */
    void seq_add(Iter &&it, Args ... args) { ((*it++ = args), ...); }

constexpr sphere(
        float cx, float cy, float cz, /* (cx, cy, cz): 球心位置 */
        float r,                      /* r           : 半径     */
        float factor_w = 1.0,
        float factor_h = 1.0) noexcept
{
    auto data_it = data.begin();
    auto elem_it = index.begin();

    constexpr float stack_step = pi / StackCnt;        // 0 ~ pi
    constexpr float sector_step = 2 * pi / SectorCnt;  // 0 ~ 2pi

    for (std::size_t i = 0; i <= StackCnt; ++i)
    {
        float phi = pi / 2 - i * stack_step, rcosphi = r * std::cos(phi), z = r * std::sin(phi);
        int k1 = i * (SectorCnt + 1), k2 = k1 + (SectorCnt + 1);
        for (std::size_t j = 0; j <= SectorCnt; ++j, ++k1, ++k2)
        {
            float theta = j * sector_step;
            float x = rcosphi * std::cos(theta), y = rcosphi * std::sin(theta);
            seq_add(data_it, cx + x, cy + y, cz + z,
                    factor_w * static_cast<GLfloat>(j) / SectorCnt,
                    factor_h * static_cast<GLfloat >(i) / StackCnt);
            if (i < StackCnt && j < SectorCnt)
            {
                if (i != 0)
                    seq_add(elem_it, k1, k2, k1 + 1);
                if (i != StackCnt - 1)
                    seq_add(elem_it, k1 + 1, k2, k2 + 1);
            }
        }
    }
}
```

具体的话我们建模了 10 个球。

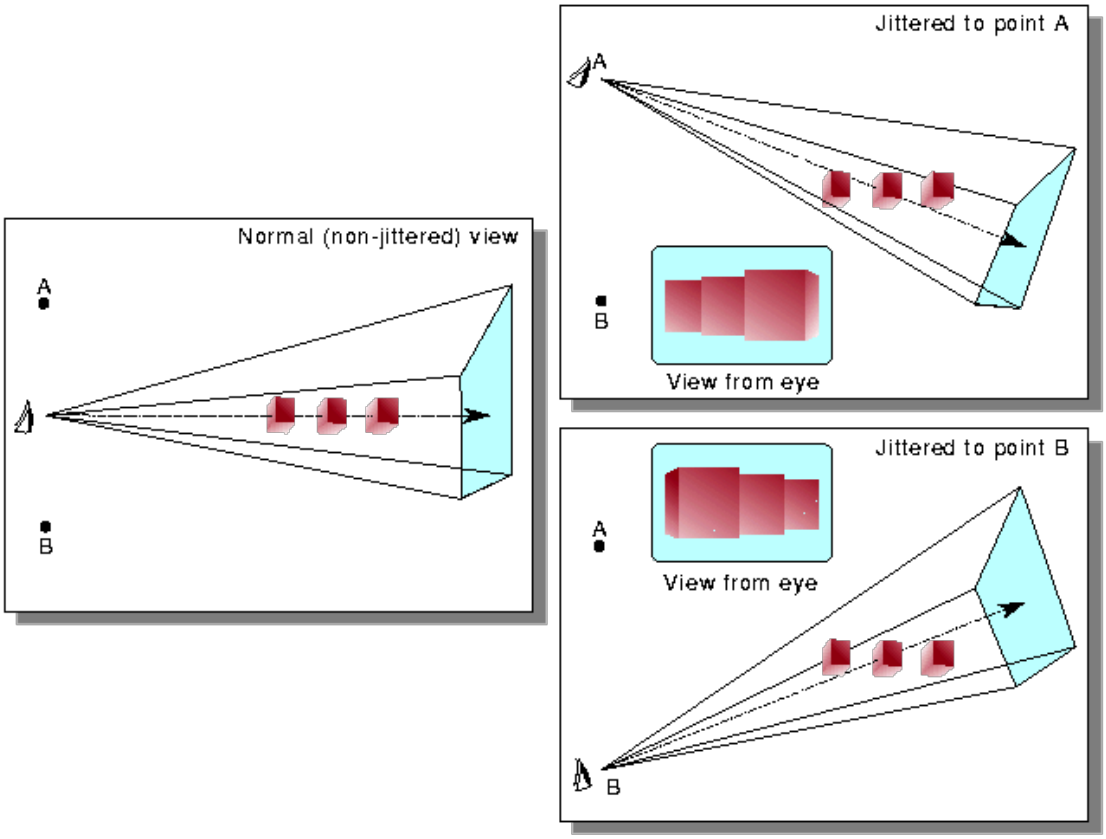
（3）关于球类的运动，直接通过物理引擎计算出当时的旋转矩阵、位移矩阵、投影矩阵等等，计算得到旋转后球体群。其中瞄准之后球会变纹理，变成显示一个「AIM」的纹理。

（4）纹理贴图，参数如下：

```cpp
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR_MIPMAP_LINEAR);
```

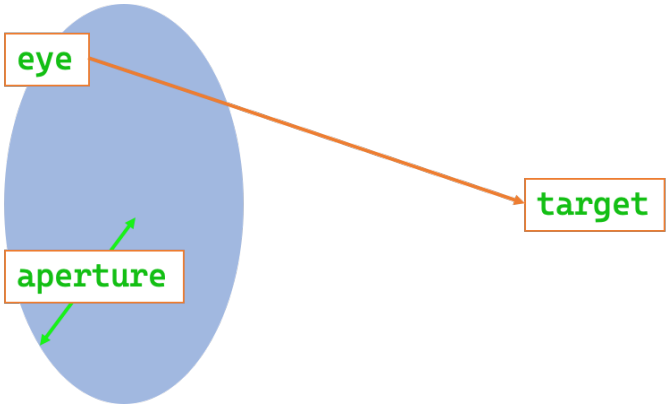
（5）景深效应（implemented but not used for performance issue）：

通过摄像头延 viewport 平行面做圆周运动，来模拟景深效果。

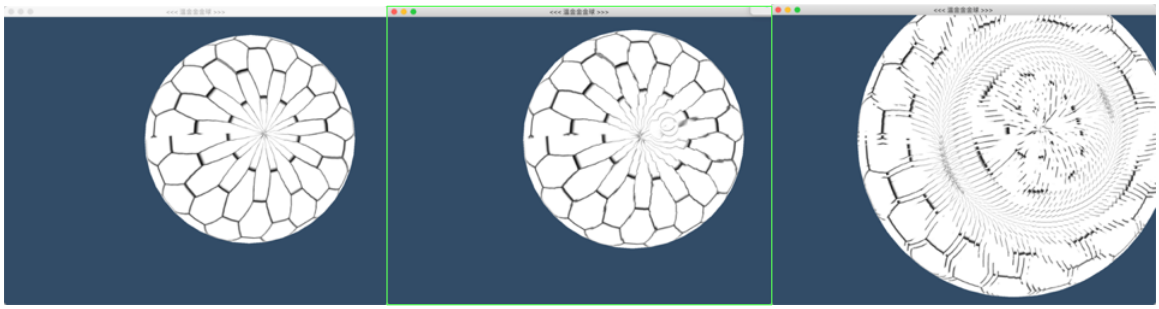


使用 accumulate buffer 可实现这个效果：

- 原始的 LookAt：glm::LookAt(eye, target, up)
- 做圆周运动的 LookAt：glm::LookAt(eye + *shuffle*, target, up)
- *shuffle* = aperture * bokeh
- glAccum({LOAD/ACCUM}, fraction)



效果如下：

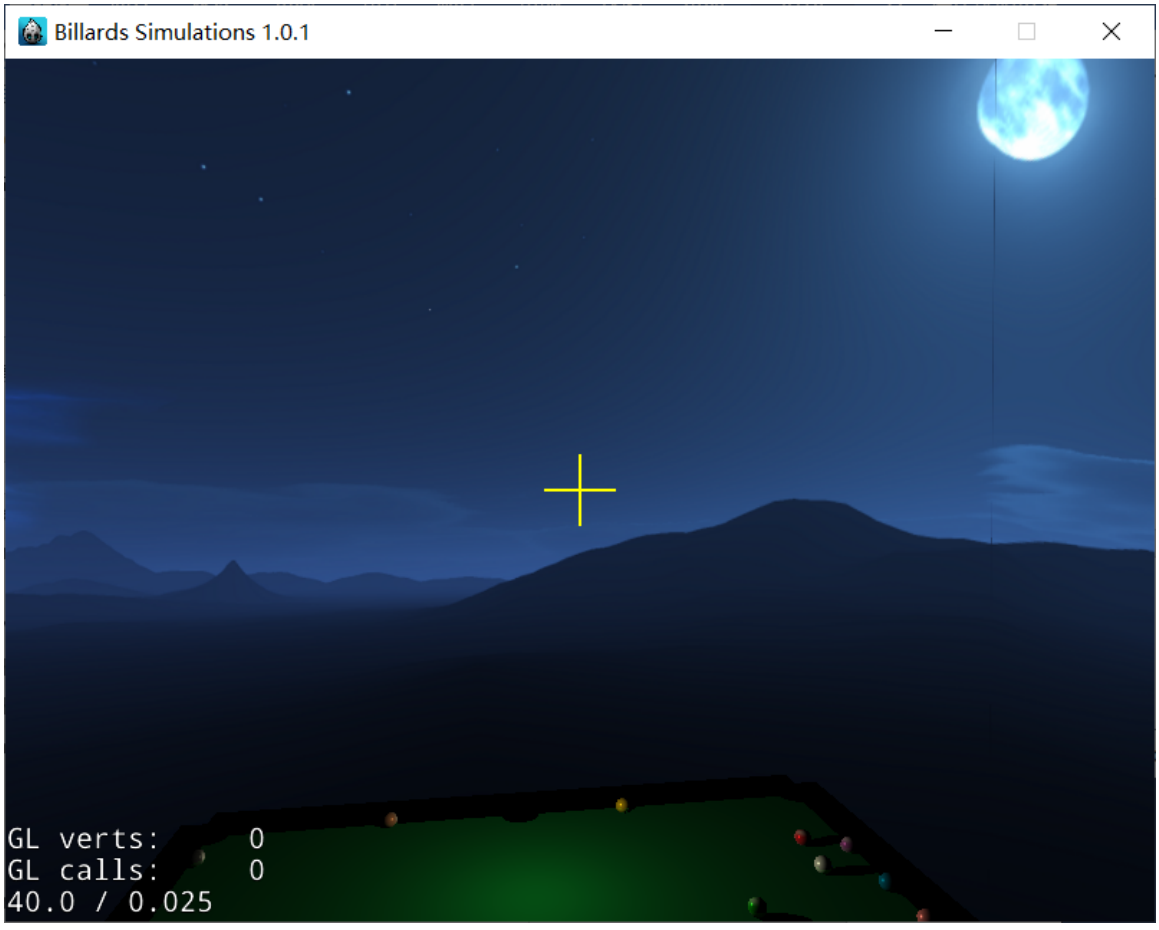


(100, 0.02)　　(100, 0.2)　　(100, 2)

（6）天空盒（SkyBox）

为了在场景中呈现更好的气氛，加入了一个天空盒特效。在 OpenGL 里有现成的 CubeMap texture 用于实现天空盒。然而悲剧的是，这里用的 cocos2dx 引擎所使用的 GLSL 版本过于老旧，不支持 CubeMap texture，仅支持 2D texture，所以我们只能通过手动贴 6 张图的方式实现天空盒了。

实现效果：



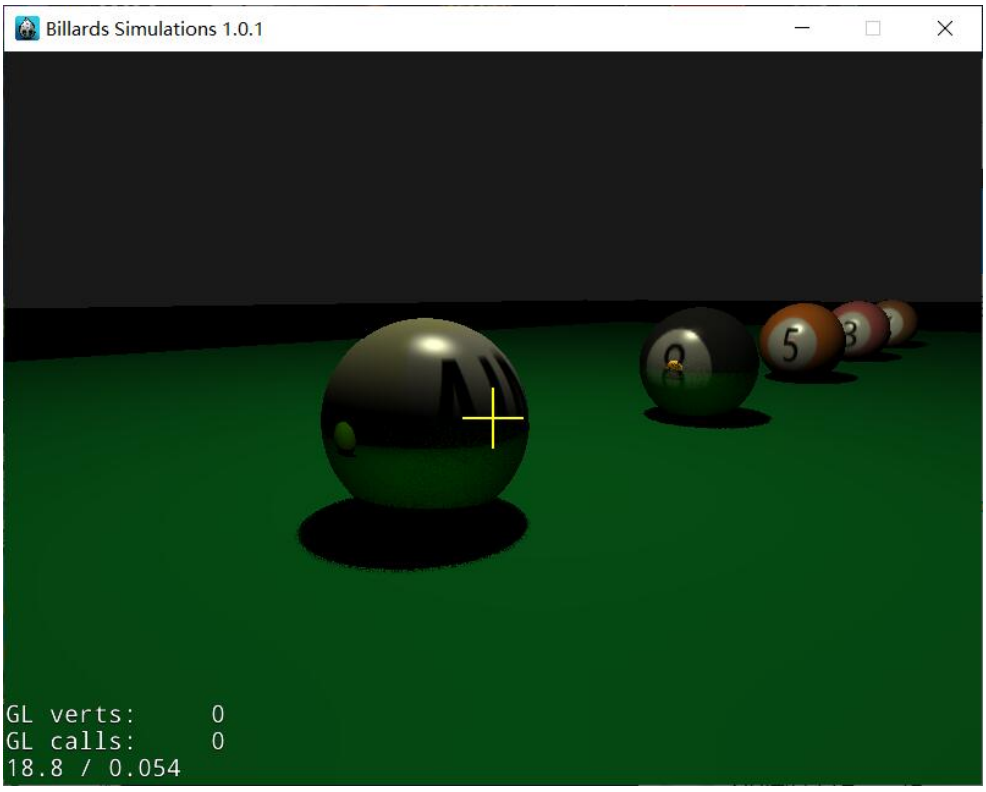
5. 流水线与光线追踪的融合

目前来说，我们用流水线做出来的图是在 OpenGL 的框架里，而用光线追踪做出来的图是在 OpenCV 里。融合的话有 2 种思路：

1. 将 OpenGL 的渲染结果导出成 bitmap，然后在 OpenCV 中与之融合产生融合后的 cv::Mat，然后通过 cv::imshow 来显示。
2. 通过 pixel shader 将光追结果 cv::Mat 融合。OpenGL 里不能直接的使用 pixel shader，我们只能将 cv::Mat 当做纹理（Sample2D）导入后，将其采样到 gl_FragCoord.xy 上（当然，要 x 和 y 要除以窗户大小）。

我们最终选择了第二种方案，融合的算法也比较简单，但效果我觉得还行，如下图。对于光线追踪部分需要传给流水线的参数：

1. 阴影系数矩阵；
2. 反光矩阵；

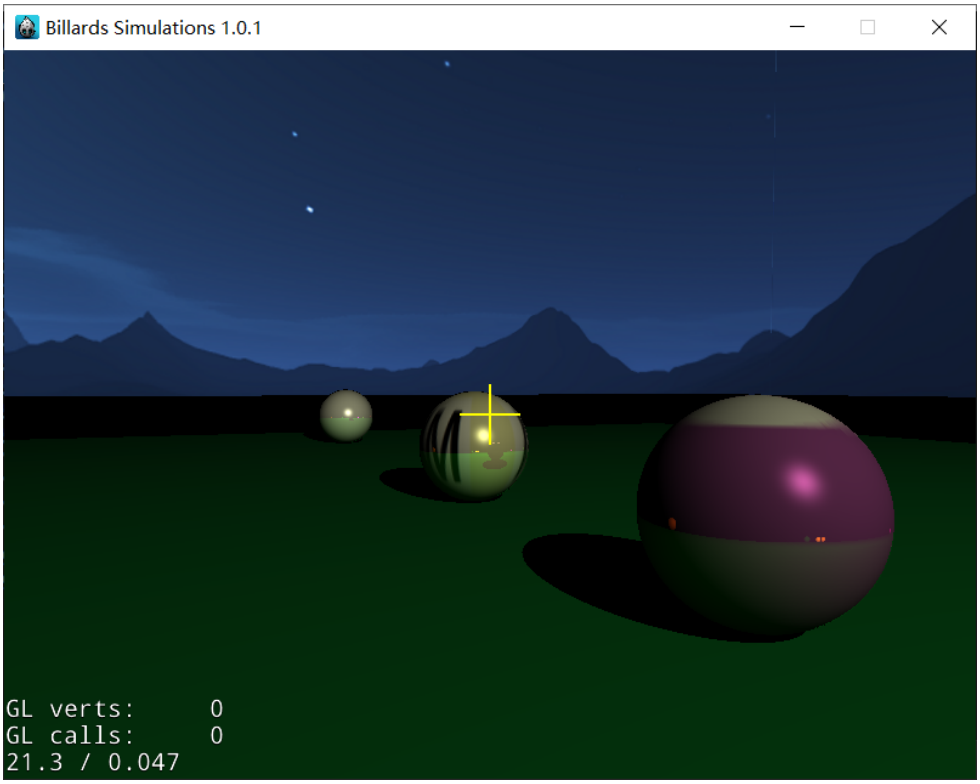


$$C_f = \alpha C_p + C_r$$

即阴影系数 * phong 得到的光照 + 反光。在 shader 中我们是这么写的：

```
gl_FragColor =
    shadowfactor/* 阴影系数 */ * vec4(result,1.0)/* 阴影系数 */ +
    texture2D(   /* 光追反光 */
        CC_Texture1, vec2(
            gl_FragCoord.x/800.0,
            -gl_FragCoord.y/600.0
        )
    )
);
```

Results



Roles in group

人员	内容
1753309 毕晓栋+1753341 刘啸威	代码整体框架与 GUI
1750761 秦政睿+1751918 温鑫	光线追踪算法实现与优化
1753070 刘佳伟+1752585 齐勖劢	OpenGL 流水线渲染、流水线与光追的融合

References

- https://zhuanlan.zhihu.com/p/30944509
- Learn OpenGL-CN
- http://www.songho.ca/opengl/gl_sphere.html
- https://raw.githubusercontent.com/quietshu/-paper-cg-RayTracing/master/RayTracing.pdf
- https://zhuanlan.zhihu.com/p/51387524
- https://zhuanlan.zhihu.com/p/41269520
- https://zhuanlan.zhihu.com/p/21376124
- https://www.cnblogs.com/graphics/archive/2010/08/09/1795348.html