**浅学图形学(实现一个简单的光追渲染器)**

**前言**

由于没什么好写的(菜鸡一枚)，**最近对图形学感兴趣，学了一点图形学**。我这些浅显的知识显然比不上你自己感兴趣然后去自主寻找资源进行学习来得好，但是显然把一些知识写出来对我是有好处的，如果你在这篇文档中体会到了图形学的乐趣那也不错(我也尽量希望达到这一点)。**浅浅地学习图形学(如实现一个简单的光线追踪渲染器)可以只涉及到高中的知识，因此我在这里写出我学习图形学的一点点知识，希望可以让不知道图形学的人对图形学有个极其大致的认识，并实现一个简易的光线追踪器(在屏幕中渲染出图像的那一刻会让人很兴奋)。**

**第一章 基础概念**

**计算机图形学(Computer Graphics,简称CG)是一种使用数学算法将二维或三维图形转化为计算机显示器的栅格形式的科学**。**简单地说，计算机图形学的主要研究内容就是研究如何在计算机中表示图形、以及利用计算机进行图形的计算、处理和显示的相关原理与算法**。

浅显地说，将3D的现实世界体现在一个画布上，需要考虑3D物体在画布中的位置，需要考虑每个部分的颜色亮度从而体现立体效果。因此为了模拟这个过程，需要有相机(小角度视域的眼睛),画布，视口，视口外的物体(可以在视线中，也可以在外面)

图形渲染通常有两种方式：一种是**光线追踪(Ray Tracing)**：主要考虑的是一个像素是如何被场景中的物体影响的。原理比较符合直觉，相对容易实现；另一种是**光栅化(Rasterization)**:主要考虑的是场景中的物体是如何影响图片中的每一个像素的。渲染时间很快，一般游戏使用的是这种方式。

|  |  |
| --- | --- |
| 图一.光线追踪简易原理示意图(类似于模拟光线,但是是从相机发出的,返回颜色而已) | 图二.光栅化简易原理示意图(类似于投影) |

**接下来我们只关注光线追踪渲染的方式。**

**第二章 实现光线追踪渲染的准备**

根据上面提到的原理**我们需要有几个对象：相机，画布，视口，需要渲染的物体**。同时正如现实世界要看见东西需要有光照，**光照可以分为全局光照(主要是为了能看见物体)，方向光照(固定方向类似于太阳光)，点光照**。**这里我们主要考虑简单地进行光线追踪渲染的尝试，因此可以直接定义相机位于(0,0,0)处，朝向Z轴，方向“向上”方向为Y+。**(电脑软件常见的坐标定义[起源])。定义一个大小为Cw和Ch的矩形画布(Canvas)，定义一个大小为Vw和Vh的矩形视口(ViewPort)，两者的边分别平行于X轴和Y轴，显然这两者对应的边是成比例的，为了简化我们可以直接从画布通过一个比例来定义矩形视口。这里定义画布到相机的距离为d(你也可以定义视口到相机的距离为d，但是光线追踪是从相机到画布再到视口，这样定义更符合直觉)。需要显示的物体坐标定义于视域内(类似于四棱锥)视口后即可。光照随意。[**在这里画布坐标系单位和显示窗口需要有一个比例，保证画布坐标值小(避免运算数据过大)**，这里涉及的比例大概对应的就是Unity中定义的PixelPerUnit(每画布世界单位对应像素值)和Unit(世界单位)]

定义完成后，我们首先根据由画布到世界的原理，取画布上的一个像素点坐标(**这里涉及界面到画布的转换,请自行理解[浮点数单从小数位数来看有较大优势,具体情况可以自行研究]**)对应到视口上的坐标(注意是三维的，由于画布和视口都与XY平面平行，考虑(X,Y)坐标即可)，至于对应到视口上是为了方便保证要显示的物体在视口后(同时更符合原理)，由坐标系定义，X和Y值范围分别是[-Cw/2,Cw/2],[-Ch/2,Ch/2]。即**需要一个CanvasToViewPort的函数**(1)

有了ViewPort上的点坐标D光线自然就表示出来了，**光线方程使用向量的形式(在3维世界中更简洁) P = O + t\*D**

**O是原点，D是ViewPort上的点，P是直线上的点**。

在数据的世界有了光线的表示自然需要有物体的表示，由于球体的数学表达简单，这里先只涉及**球体的方程 (P - C)(P - C) = r^2 ; P是球体上的点，C是球体原点， r是半径。 [其他的可以自行研究]**

球体和光线方程**联立得t^2 \* D^2 + 2 \* t \* D \* CO + CO ^ 2 - r ^ 2 = 0这个方程， 令a = D^2 , b = 2 \* D \* CO , c = CO ^ 2 - r ^ 2. 显然当 dealt = b^2 - 4\*a\*c > 0时相交, t = [-b +- dealt^(1/2) ]/2\*a**

由上述方程光线击中球体能够有**一个IntersectRaySphere函数(2)来获取击中的物体并返回得到最小的t(后面的被遮住了)**；**如果击中则可以获取物体当前被击中点的颜色**。先抽象**一个ComputeLighting(计算颜色强度)的函数(3)**

**颜色和强度按照颜色通道(R,G,B)进行乘法运算(即为数乘以向量)从而**

从以上分析看主要是物体方程和计算颜色的函数需要被拓展和实现(CanvasToViewPort简单通过相似即可实现)

物体方程我们这里不拓展，**接下来主要实现ComputeLighting**

**第三章 光线运行计算及简易光学现象的建模(以下都是简单地在球体场景下的建模)**

**首先规定光线向量P(OP)，O是原点(即相机的坐标)，P是击中物体上的点，同时规定光学计算都需要的单位法向量N(在球体上简单地为CP，C为球心坐标，定义为单位法向量便于之后的计算)**

**3.1 该文档需要实现的光学现象**

**球体的漫反射，镜面反射，阴影，反射。**

**3.2 漫反射的建模**

|  |  |
| --- | --- |
|  | 在这里考虑是极小光线(宽度为I)照到的极小的区域A(宽度为A)，L向量即为指向光源的向量(这样定义便于与之后的统一计算)，**N向量为单位法向量**，那么通过几何类比，将光的宽度I表示光强，宽度A均摊宽度I的光强，I/A = cosA = N\*L/|N|\*|L| , I/A即为每单位面积光强度比，同时规定场景中有n个能照射到P点的光源对应的光强为In，因此场景中P点漫反射光度可计算为: |

**3.3 镜面反射建模**

|  |  |
| --- | --- |
|  | N和L和之前意义一致，R向量为反射向量，V为视线向量指向相机，实际获得当前亮度的为视线向量V，**镜面反射不能单纯地进行向量分解，且不同的物体会有不同的镜面反射效果**。  因此**先对镜面反射的亮度关系进行以下建模**：1.先近似寻找一个符合角度A小值大(始终小于1)，当角度A大于等于90度时都为0值，值最好曲线变化的简单的便于描述的函数 2.由cosA可以很简单地表示并且cos函数在-90度到90度符合以上的描述，来选择cos函数作为描述的函数。3.再引入一个参数来调控函数的差异来体现不同物体镜面反射的效果。4.这里**引入S(通常被称为镜面反射指数)**，并定义S的作用为作为函数的幂次来近似模拟指数高的物体A角度小亮度高且衰减快的效果。**[这就是通常光泽度的简单定义]**  其中的**R可以表示为R = 2\*N(N\*L) - L**  **视线向量** **V = O - P**  因此**镜面反射的光照方程**见左图: |

**3.4 阴影的实现**

在本文档的场景中阴影比较容易实现，只需要在计算光强(ComputeLighting)时在计算漫反射和镜面反射之前先判断当前待计算的点的待计算的光源的光路上是否有球体遮挡即可，需要抽象一个函数，这个函数本质上也是求向量和球体方程的交点，可以封装前面提到的**IntersectRaySphere函数为一个功能更强大的(4)函数ClosestInter返回t值并且返回求交完成的球体。如果有球体相交就跳过下面计算的过程(和上述函数使用差别不大，适当调整传入的参数即可)**。

**3.5 反射的实现**

|  |  |
| --- | --- |
|  | 其中V为视线向量，N为P点对应得单位法向量，R为反射向量，**V和R的方向都是为了便于计算，不必过于纠结**。  再次使用到了**求反射向量**过程和之前差不多，只是参数意义不太一样，因此封装一个(5)**函数Reflect**来求反射向量R，**公式同求镜面反射**  由于反射，**P点的颜色**显然会受到从反射向量R获取的**颜色**的影响。这里的反射效果同样会受到物体性质的影响。因此继续对反射进行简单的建模：  1.**P点的颜色**由两部分构成：a.未反射部分的颜色(记为local\_color) b.反射部分的颜色(记为reflected\_color)，需要有一个关系关联这两个部分 2. 分析生活中的物体如(理想的)镜子，完全反射相当于只包含b部分；理想的黑体(无反射和透射)相当于只包含a部分，介于两者中值的物体显然这两部分是均分的，因此可以简易地用加权函数来关联这两个部分。3.直接引入参数r(0<=r<=1)称为反射率来模拟上面的现象。  **方程见左**  **从方程也能看出需要两个颜色，并且反射并不是单次的(得到颜色需要先得到两个小颜色)，因此需要用到递归** |

**第四章 代码实现以及相应的效果(这里用C#实现[好像通常是C++])**

**综合以上的函数以及层次结构,用一个TraceRay函数返回颜色**





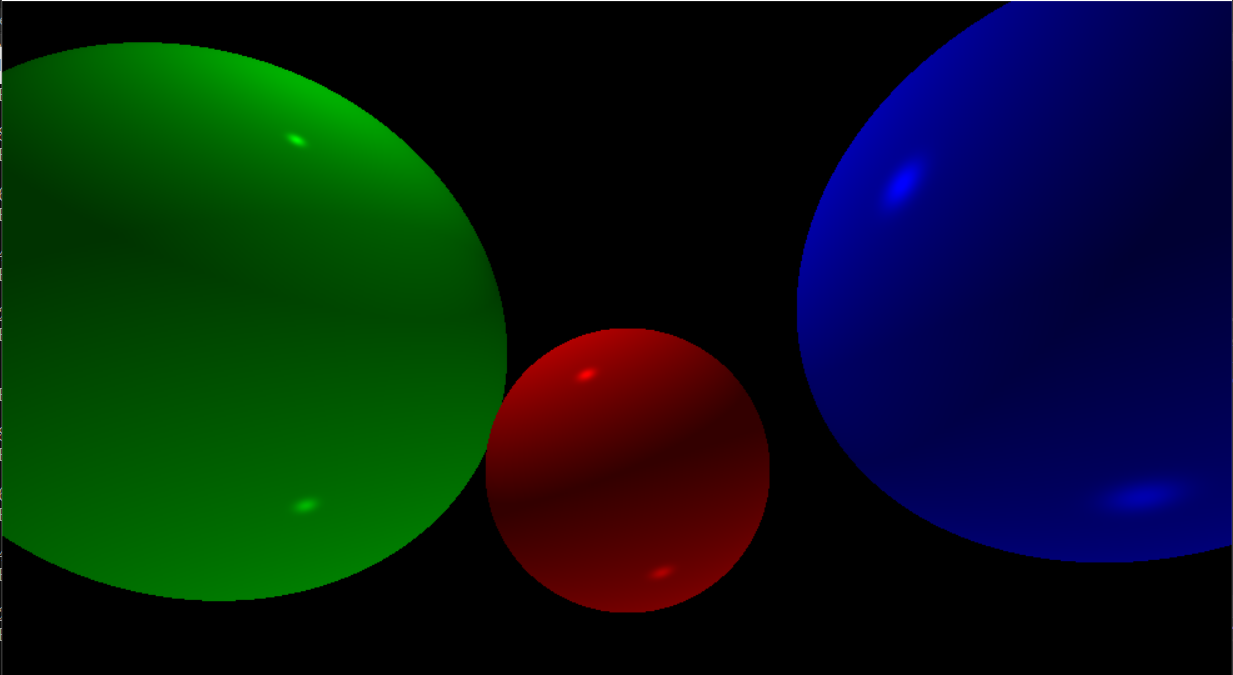
**ComputeLighting**



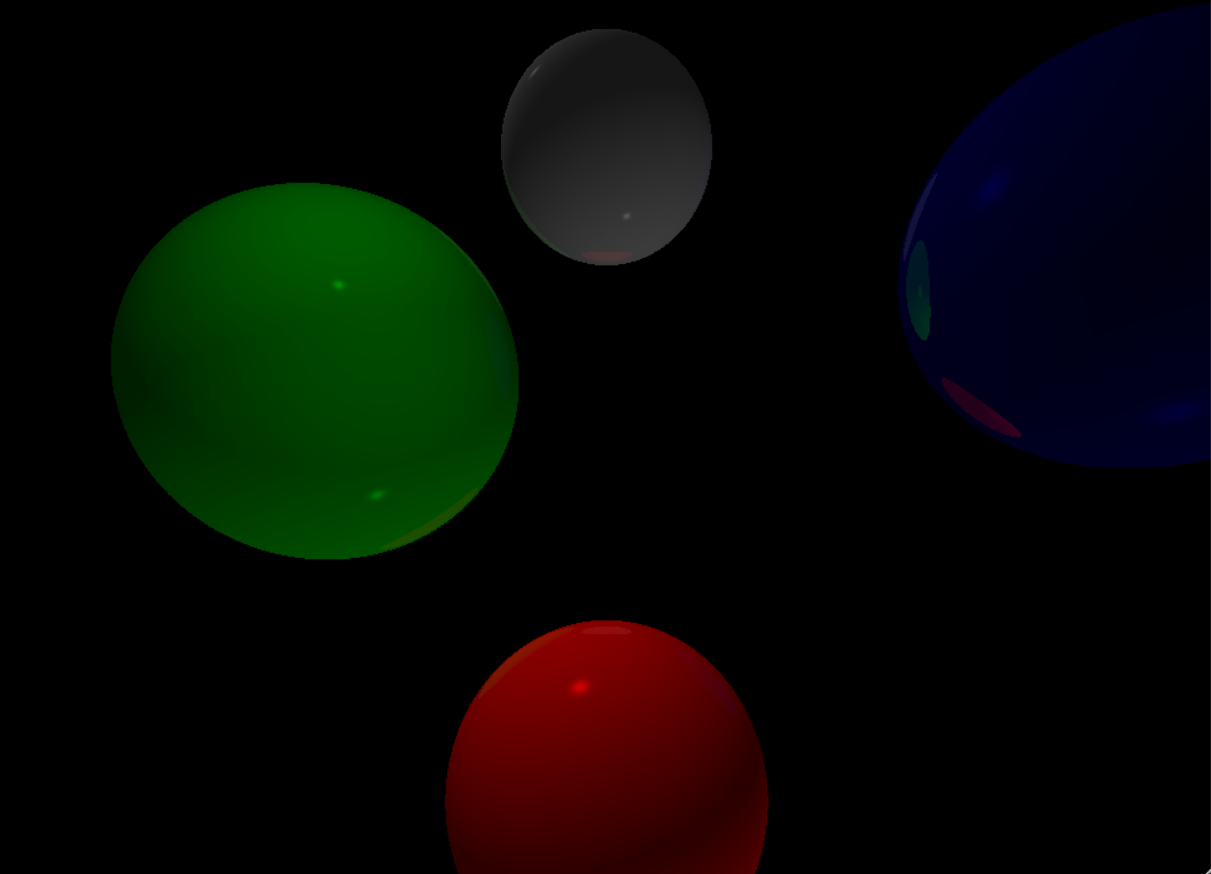
还有一个最主要的部分是反射的递归，我简要地描述**递归出口：没有交到球体，递归次数达到规定极限**   
**算了，不粘贴了，毕竟这篇文档主要的目的是分析原理实现一个简易的光线追踪渲染器，如果感兴趣的话可以看看**[**我的代码**](https://github.com/Unknownky/Simple-RayTracing-Renderer)**(先叠几个甲(比较简陋)(不会使用多线程)(没有进行运算优化))**

**效果如下：**

**带漫反射、镜面反射和阴影的**



**带漫反射、镜面反射、阴影和反射的**



**两者亮度设置不一样，并且反射会影响亮度**

**第五章 优化**

优化的部分我也没有做，但是大概有如下几种能够理解实现的优化，如果你对这些部分感兴趣的话，可以自行研究

I/O操作首当其冲

1.并行优化 (线程并行)

2.缓存不变的值 (一些不变的值不用重复计算，如r^2)

3.阴影计算的优化 (可以从阴影的点出发蔓延判断周围的点是否在阴影中从而避免阴影前的计算)

4.空间结构使用边界体结构 (大致为广大的空背景没必要计算，只需要计算可能存在物体的边界体部分的光线追踪)

5.子采样 (间隔采颜色样本对跳过的像素点颜色进行假设来提高运算速度, x和y步长为2的话可以获得四倍加速!)

…………还有很多…………

**结语(如果之后我不想再写了的话)**

以下是我单纯地讨论一些可以被看做事实的观点：本篇文档以上所述甚至称不上图形学的冰山一角，距离图形学的门路也相差很远，如果以上内容引起了你对于图形学的兴趣那也不错。**我的兴趣比较广泛，本质是个菜鸡，写这篇文档也只是为了证明我在小目标上没有半途而费，并总结一点学到的知识(试试看是不是更有效果)**。图形学听说是编程方向一个很难的部分，可能你付出同等(甚至更多)的努力从利益的角度比不上从普通编程方向获得的利益，这也应该是图形学从业人数少的原因，但是如果你有追求兴趣胜过利益的勇气和格局，并真切地认为你对图形学很感兴趣，那我会认为你很厉害。**(无聊BB)**