计算机图形学期末大作业报告

——照明模型与纹理

南开大学软件学院软件工程专业 2019 级 2 班 韦诗睿 1913184

I Introduction

本学期的课程从一维的函数变换开始，到二维的点、线、面操作，最后到三维的矩阵变化和场景漫游，顶点经过一系列变换最后成为呈现在屏幕窗口的图形，一整个 pipeline 讲述了计算机中的图形世界是如何从零开始建立起来。而线框图仅仅只是开始，着色与纹理作为场景真实感渲染的重要环节，也值得一探究竟。

本次大作业实验基于以下两个学习案例，主要探究照明模型和纹理的原理与应用：

Study Case 8.1：利用 OpenGL 制作着色物体

设计一个程序：一个照相机飞越空间观看各种多边形网格物体。然后在场景中设置一个点光源，给网格设置各种材质属性。包括环境光、漫射光和镜面反射光分量。

Study Case 8.2：纹理绘制

编写一个程序：能够读取图像文件，并把它贴到一个 OpenGL 纹理物体上，试验放不同的图像纹理到一个立方体的各个面，然后设置一个动画旋转立方体。

II Light

2.1 Problem Formulation

光可以使物体的网络建模看上去更加真实，如何在计算机中模拟真实世界的光照则需要应用不同的着色模型。着色模型试图描述光源发出的光线和场景中的物体是如何相互作用，主要可以分为三个部分——反射、光源和物体。

光照射物体表面可能会发生吸收、反射和折射，这里主要关注反射。反射可以分为漫反射（diffuse scattering）和镜面反射（specular reflection）两种；为了更加接近现实，还可以引入第三个光分量，环境光(ambient light)。环境光发出后经过周围不同物体的多次反射，使得物体背向光源的阴影部分可以更加柔和。全部的反射光为三种反射分量之和，不同的光源发出的三种光的光强不同，这里用

光强I表示；照射到的物体材质不同，三种光的反射程度也有所不同，这里用反射系数ρ表示；而光源、物体、眼睛三者的相对位置也决定了最后反射光进入眼睛的多少，这里分别用$Lambert$和$Phong^f$表示漫反射和镜面反射的位置关系，其中f为衰减系数。

综合考虑三种反射分量，最后从光源到达眼睛的总光亮I为：

$$I = \text{环境光} + \text{漫反射光} + \text{镜面反射光}$$

$$I = I_a\rho_a + I_d\rho_d \times lambert + I_{sp}\rho_s \times phong^f$$

而彩色光由 RGB 三种颜色通过不同比例混合而成，因此在处理彩色光的时候需要单独计算每个颜色分量的光强I_r, I_g, I_b，最后再简单相加。

以上即为简单的着色模型。在光源部分，我们需要设定的是三种光分量的光强大小（分 RGB 分量）以及光源的位置；在物体部分，我们需要设定的是物体的材质，即物体对于三种光分量的反射系数（分 RGB 分量）。

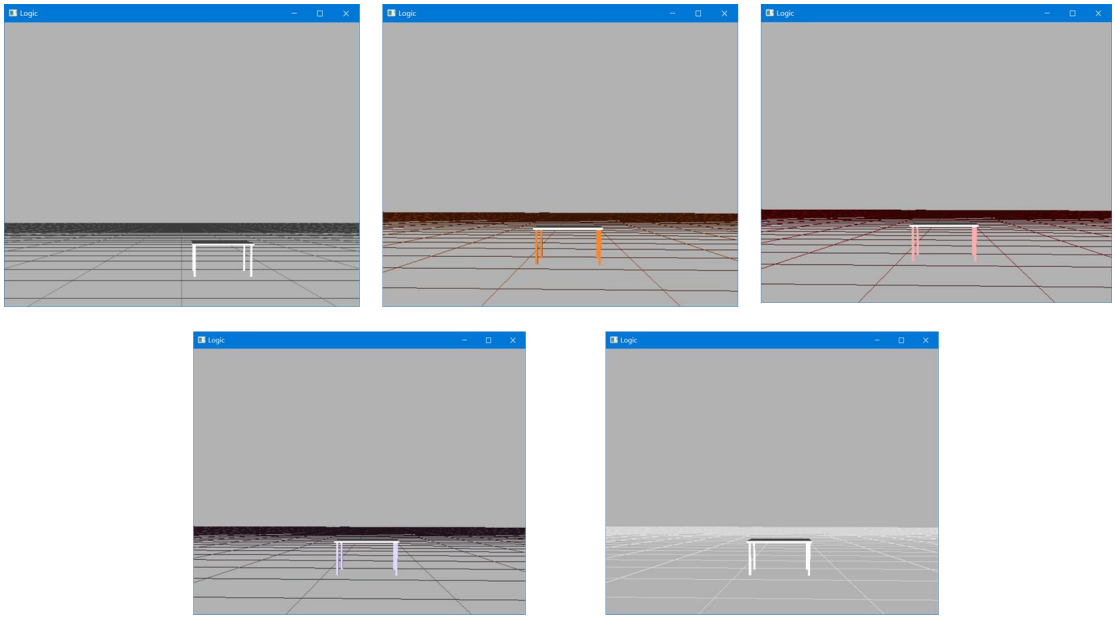
2.2 Experiment

2.2.1 Experiment Setup

为了设置光与物体发生交互，绘制了一张桌子作为反射物体，并给桌子选择了不同的材质置于表面。为了探究光的位置和眼睛视角对反射效果的影响，将整个场景放置于照相机漫游框架中，通过键盘控制照相机移动来观察不同角度下桌子对光的反射效果。照相机的移动不仅可以观察固定点光源下不同视角的情况，还可以将光源固定于照相机上以模仿探照灯的效果。

2.2.1 Material

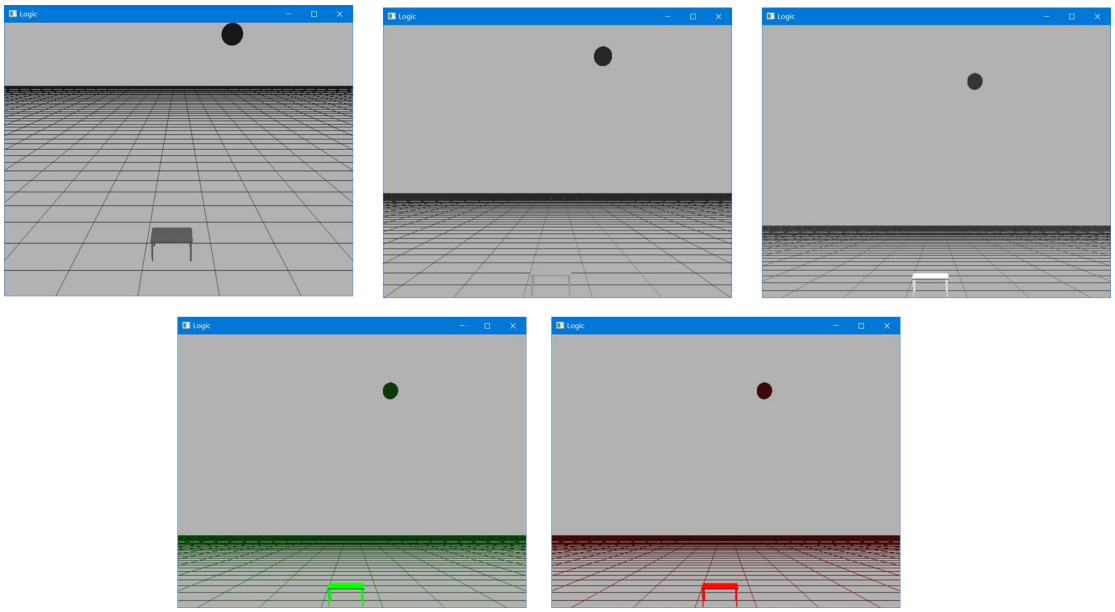
首先探究了不同物体材质在相同光照下的表现。根据【*McReynolds97*】提供的反射系数，为桌腿设置了银、铜、红宝石、紫罗兰、灰色五种材质/颜色，从正面视角观察反射效果如下：



可以看到不同材质/颜色存在明显差别，通过反射系数的搭配，可以使物体呈现多种特定材质。

2.2.2 Intensity

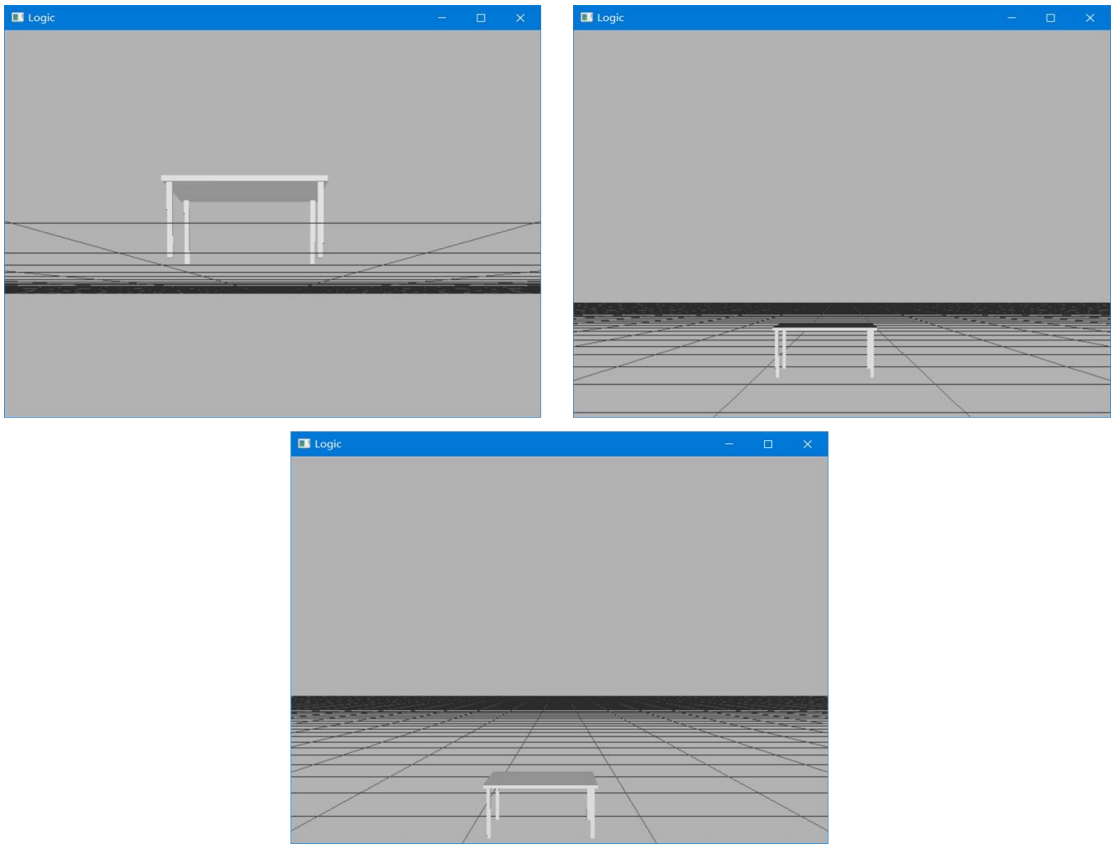
其次探究相同材质在不同光强下的反射效果。使用银材质，使用静止点光源分别在光强为 0.3、0.6、0.9 的白光下以及光强为 1.0 的绿、红单色光下进行反射，反射效果如下：



可以看到银材质在不同光强下的反射效果存在显著差异。通过光强和材质的搭配，便可以模拟现实生活中的多种光照情况。

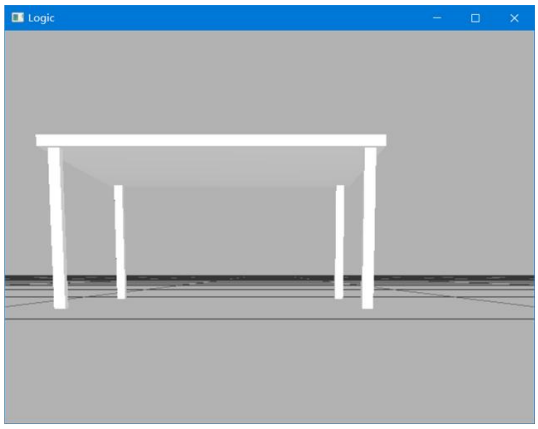
2.2.3 Position

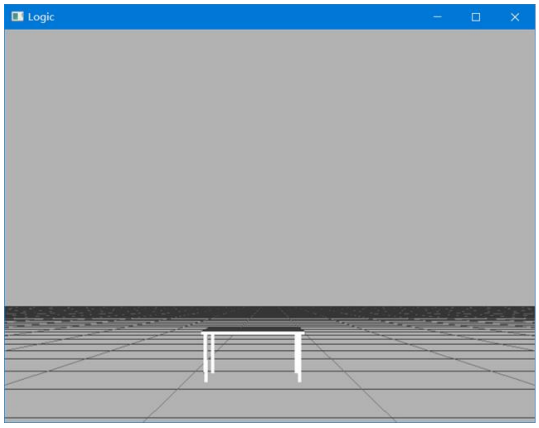
结合照相机的场景漫游，首先将光源固定于照相机处，在银材质以及 0.7 的光强下，探究不同位置下物体的反射效果：

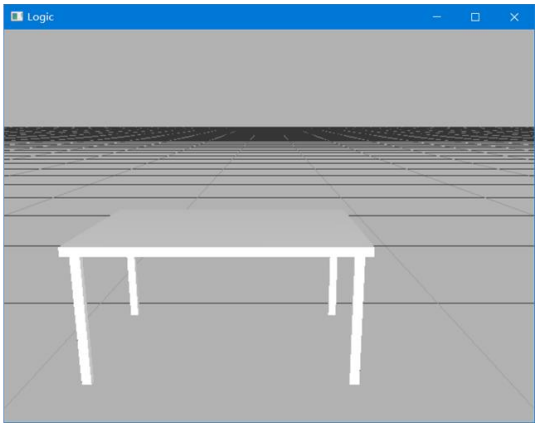


可以看到，当光源固定在照相机处时，随着位置和视角的变化，桌子顶部逐渐由黑转为灰，观察到明显的反射光增强。

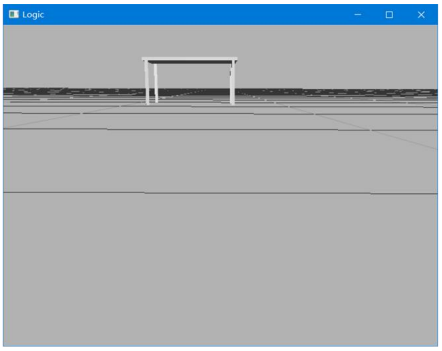
在银材质以及 0.9 的光强下反射效果，可以看到，由于光强的增加，桌子顶部的反射效果更加明显：

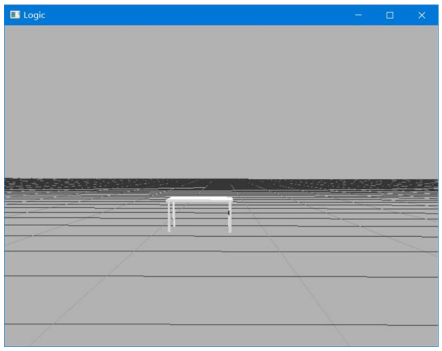


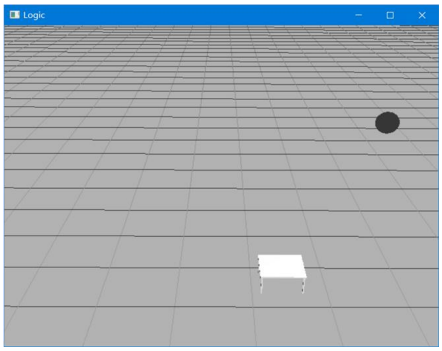




保持材料和光强 0.9 不变，将光源固定，反射效果如下：



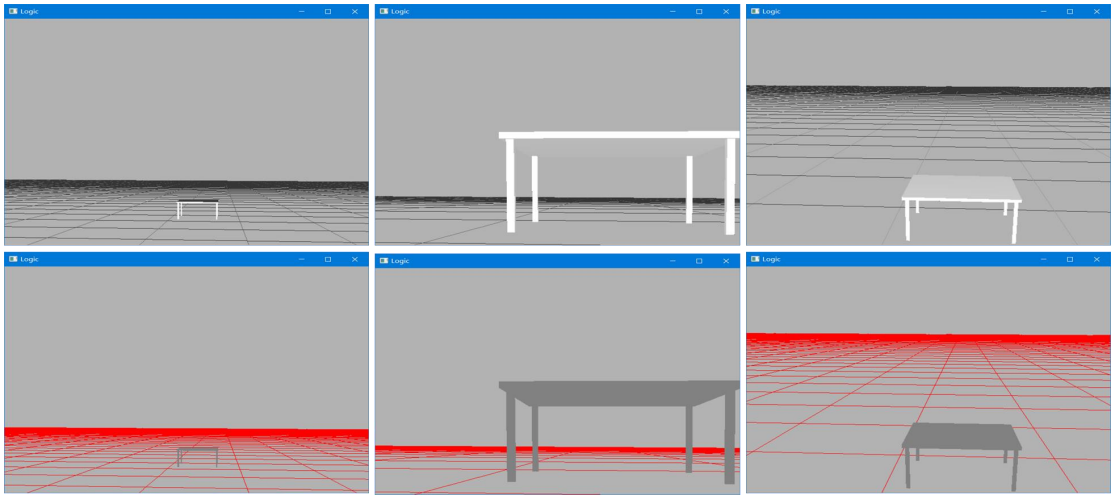




可以看到，由于光源固定于桌子上方，因此不管视角如何改变，桌子顶部一直处于被照亮的状态，而桌子底部一直处于阴影之中。

2.2.4 On/OFF

通过关灯和开灯的设置，探究灯光效果下场景的真实性。保持桌子的材质为银和开灯后 0.9 光强不变，在不同视角下关灯和开灯的效果如下：



可以看到，无论视角如何改变，关灯后桌子的颜色没有任何改变；而在开灯后，随着视角和位置的变化，桌子的颜色也随之变化，更加接近现实生活场景。

III Texture

3.1 Problem Formulation

图像的真实感也可以通过为网格物体的各个表面贴纹理而被显著增强，纹理最普遍的来源是生成函数和位图。通过数学函数可以定义纹理，任何函数 texture(s,t)，对于 s 和 t 在 0 到 1 之间的取值产生一个颜色或者亮度值，都可以认为定义了纹理。位图可以认为是矩阵一样的像素数组，每一个数组元素都存储着颜色值，将颜色值转为 0 到 1 之间的 s, t 值便成为了纹理。

有了纹理之后，下一步是如何将其适当地映射到物体表面，并从相机中看到它。这一过程可以分为从纹理坐标到世界坐标T_{tw}和从世界坐标到屏幕坐标T_{ws}两步，纹理上的值（s*,t*）映射到物体表面的点（x,y,z），最后到达屏幕上的像素$(s_x, s_y) = T_{ws}(T_{tw}(s^*, t^*))$。在这一过程中主要解决的问题即为，哪个纹理坐标（s,t）是和点（x,y,z）相对应。在本次实验中，主要关注的是在平面上贴纹理。使用 OpenGL 可以直接指定纹理空间中的点和面片上的顶点的对应关系，当每一个顶点都有了一个纹理坐标后，纹理便通过映射贴到了面片上，面片内部的点的

纹理坐标值通过插值进行计算。插值不能使用简单的线性插值，因为立体视觉中近大远小的现象会使得线性插值得到的纹理扭曲，正确的做法是使用双线性插值。

3.2 Experiment

3.2.1 Experiment Setup

本次实验主要是为一个旋转的立方体的六个面贴图。为了探究不同纹理效果，设置了三种纹理样式，分别为棋盘纹理、无规则纹理和图片纹理。纹理的创建首先需要一个图片大小的 RGB 三元组，每一个三元组都代表一个像素的三通道亮度值。得到像素图像后通过 OpenGL 的调用来创建纹理，包括绑定纹理名字、选择纹理放缩时的像素处理方法（这里选择使用相邻 NEAREST 的元素来填充）以及将像素数组传递给当前纹理。

3.2.2 Texture Function

首先是人工编写的函数生成纹理。在 64*64 大小的棋盘下，每 8 个元素的亮度值在[0，255]两者中跳变一次，便得到了黑白相间且和坐标参数有关的棋盘纹理。相关算法如下：

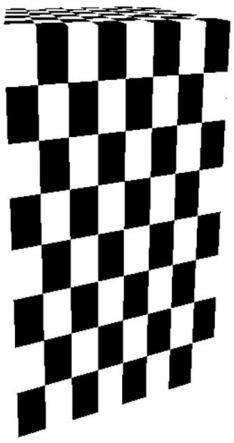
```
1.    //生成棋盘纹理数组
2.     void makeCheckerboard() {
3.      //棋盘大小：64*64
4.      nrows = 64;
5.      ncols = 64;
6.
7.      //位图数据生成
8.      pixel = new RGB[nrows * ncols];
9.
10.     int count = 0;
11.     for (int i = 0; i < nrows; i++) {
12.      for (int j = 0; j < ncols; j++) {
13.       //棋盘纹理：每 8 个像素在 0，255 之间跳变一次
14.       int c=(((i & 0x8) == 0) ^((j & 0x8) == 0)) *255;
15.       pixel[count].r = c;
16.       pixel[count].g = c;
17.       pixel[count].b = c;
18.       count += 1;
19.      }
20.     }
21.    }
```

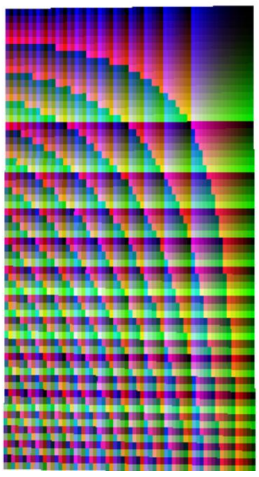
而无规则纹理便是将 RGB 三通道的亮度值赋为和坐标相关的 0-255 之间的值，

具体算法如下：

```
1.    //生成无规则纹理
2.     void makeRandom() {
3.      nrows = 64;
4.      ncols = 64;
5.      pixel = new RGB[nrows * ncols];
6.
7.      int count = 0;
8.      for (int i = 0; i < nrows; i++) {
9.       for (int j = 0; j < ncols; j++) {
10.       pixel[count].r = (i * j) % 255;
11.       pixel[count].g = (i * i) % 255;
12.       pixel[count].b = (j * j) % 255;;
13.       count++;
14.      }
15.     }
16.   }
```

两种纹理的效果如下：





3.2.3 Image

图像纹理通过读取图像文件并且在内存中创建像素图像完成。在这里我并没有选择使用读取 BMP 图像的方法，而是采用了比较常用的图像解码库 stb_image[1]，可以解析包含 png\jpg 在内的多种图像格式。该图像库通过将头文件转为可执行

[1] *https://github.com/nothings/stb*

的源码进行使用，支持包括图像加载、图像写入、尺寸更改等多种图像操作。图片纹理载入效果如下：



Ⅳ Conclusion

通过对照明模型和纹理贴图相关方法的学习和实践，我了解到如何在线框图的基础上更进一步，使计算机内的图像世界塑造得越来越接近真实场景。

在调研本次大作业选题的过程中，我曾学习到一个模型使用 OpenGL 来模拟黑洞的吸积盘[2]，主要方法为通过计算光线和吸积盘测地线的交点，以确定屏幕上每一个像素来自吸积盘上哪条原始光线以及其走向。这样的光线模型比起本次实验探究的简易光照模型复杂了很多，但万变不离其宗的还是那一整个 pipeline，光线的世界坐标如何转为相机坐标再转为屏幕坐标。在电影《星际穿越》的制作过程中，科学顾问基普·索恩以及特效制作团队也曾就电影里的黑洞画面如何模拟进行讨论，而讨论的问题无非可以简单归结为——如何确定相机“拍摄黑洞”的走向，以及光线在黑洞周围的运动轨迹。[3]最终在看到近期艺画开天的《三体》动画中宣布面壁者时的画面移动以及光线效果后，我确定了尝试照明模型结合照相机漫游的本次大作业的前半部分内容。

虽然最后做出来的效果还存在诸多可完善之处，如可将光源分离照相机进行单独移动、多绘制几个物体查看光照效果，以及纹理问题上探究如何使纹理画面呈现得更好、将纹理贴到曲面上等，但我想只要掌握了最基本的一个顶点如何呈

---

[2] *Müller, Thomas, and Jörg Frauendiener. "Interactive visualization of a thin disc around a Schwarzschild black hole." European journal of physics 33, no. 4 (2012): 955.*

[3] *James, Oliver, Eugénie von Tunzelmann, Paul Franklin, and Kip S. Thorne. "Gravitational lensing by spinning black holes in astrophysics, and in the movie Interstellar." Classical and Quantum Gravity 32, no. 6 (2015): 065001.*

现到屏幕上这一整个处理原理，那么如何添加真实感的渲染都是锦上添花。

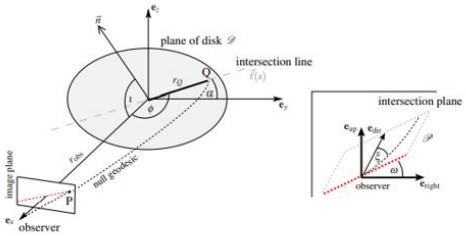
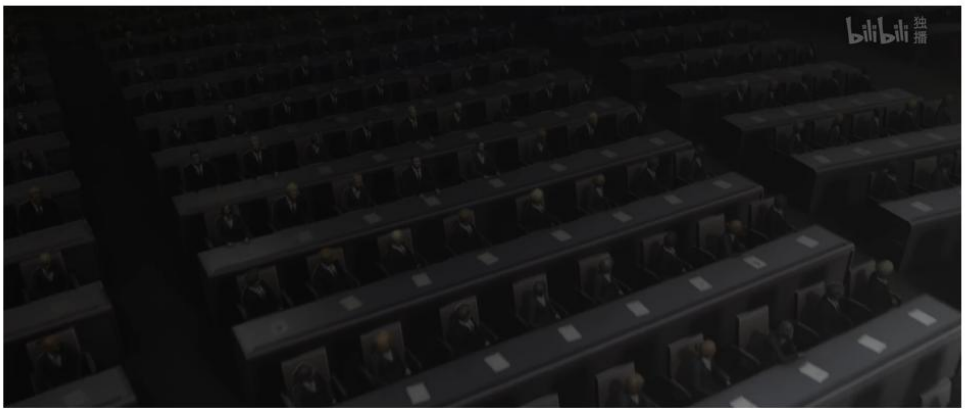


Figure 1. For each pixel P of the virtual image plane of an observer located at $r = r_{\rm obs}$, the intersection Q of the corresponding null geodesic and the thin disc with inclination ι has to be determined. With respect to the observer's camera system, the intersection plane is tilted by the angle ω and the geodesic has initial angle ξ.



通过本门课程的学习，我对于计算机图形学和计算机视觉的知识都有了更深的兴趣，希望自己能够保持这份兴趣在计算机中能够创造一个更真实的世界。本次实验代码可见于 *https://github.com/RuiNov1st/NKU-SE-Computer-Graphics.*

Ⅴ Reference

[1] Hill.F.S. and Kelley.S.M., Computer Graphics: Using OpenGL, Third Edition, 2009.