interactive computer graphics notes

Jerry chaos

2025 年 5 月 14 日

前言

这是我复习Edward Angel与Dave Shreiner的合著《交互式计算机图形学 – 基于OpenGL着色器的自顶向下方法（第六版）》的笔记。《交互式计算机图形学第六版》计算机图形学考虑的是用计算机生成图片或者图像的方方面面，包括但不限于图形图像的生成、处理、显示。主要活跃的有至少两个分支：a.图形人机接口；b.可视化技术。1995年第一款3dfx Voodoo图形加速卡首次发布，GPU的绘制结构发生了明显变化，经历了从有限的固化算法（Fixed-Function GPU）到统一渲染架构下的完全可编程GPU（Programmable GPU），其后再经过OpenGL ES和WebGL扩展到移动互联网领域。

Chapter 1

图形系统和模型

图形系统主要包括6个主要的物理部件：

1. 输入设备
2. 输出设备
3. CPU
4. GPU
5. 存储器
6. 帧缓存

呈像模型有：

- 虚拟照相机模型：从小孔成像说起；
- 笔试绘图仪模型：其方法类似于，moveto，lineto

如果设计的图像系统是基于虚拟照相机模型的话，其在API的设计上需要有相应的函数来确定以下四中对象：

1. 对象
2. 观察者
3. 光源
4. 材质属性

1.1 图形绘制流水线

图元的类型与顶点集定义了场景的几何数据。一个普通场景可能需要几千甚至几万个顶点，复杂场景就更多了。如此多的数据就需要流水线的方式去处理这些数据，各个模块只负责处理好自己的任务。图形绘制流水线主要由两部分组成：

1. 流水线前端处理
2. 流水线后端处理

1.1.1 流水线前端处理

这个阶段的流水线主要针对顶点与图元在几何定义阶段的计算与处理，包括顶点处理与片元裁剪重组两个模块。

顶点处理

顶点处理模块的功能是批量的对所有顶点进行矩阵变化与颜色计算，矩阵变换是指将模型在制作阶段的物体坐标系变换到3D场景的世界坐标系下，然后继续变换到观察者（照相机）坐标系下。这个过程至少需要2次矩阵乘法，我们可以先将模型变化矩阵与视角变化矩阵相乘，从而得到被称作模视矩阵(mv)的新矩阵，然后再与各个顶点相乘。这个模块还可以进行顶点颜色的计算，可以使用常见的光照模型，比如Phong模型；亦可使用立即数直接指定统一的顶点颜色。此前我有疑问为什么要这么早计算顶点的颜色或者计算的意义在哪里。首先顶点的光照计算肯定要放在绘制流水线的前端处理，因为无论什么场景在一帧中的显示画面是一定的，比如对象/光源位置、光源强度、观察角度等，而需要计算的是一大批符合物理光照规律的顶点，用流水线肯定没错。而计算的意义在于为后端光栅化处理向帧缓存写入片元颜色奠定的数据基础，不然一个线段只定义两个顶点，要绘制在显示器上，倘若没有顶点处理阶段时对颜色的计算处理，线段中间的顶点选什么颜色？倘若端点颜色全部是黑色，中间顶点是黑色无可厚非；一个端点白色，一个黑色，中间顶点就需要插值计算，显示结果便是个渐变线段；指定固定颜色就更好说了，经常在Demo中看到的屏幕上一个纯色三角形，那就是指定了三个顶点全是红色的绘制结果，也可以通过flat插值方式，让整个三角面使用最后一个顶点的颜色，这样渲染下来也是纯色。

图元的裁剪与重组

注意这里词汇的变化，将顶点改成了图元。这就意味着这个模块处理的最小单元是由顶点构成的图元，所谓图元就是具体的图形图像系统指定的最基本的图形，比如三角形、线段、圆等。图元的裁剪与重组是将视角（摄像机、观察者的眼睛）不可见的物体部分彻底剔除，流水线会将几何体不可见的部分按照一定的裁剪算法进行处理，然后将可见部分重组成一个或多个新的图元。至于摄像机后边的独立几何体我们是全部看不见的，也就没有必要让它的数据完整的走过流水线，这个过滤过程一般是在引擎的BSP阶段直接排除，而不是让流水线来处理。裁剪掉的不作处理，处理了也没有意义。但能看见的图元尚需要进行投影变换，其目的是将这些图元映射到投影平面上，从而防止光栅化后进行更多投影的矩阵计算。此期间会涉及到四维矩阵，其中第四维度一般会用来存储原始图元深度，从而在更后期的片元处理阶段比较大小决定谁应该压着谁。

1.1.2 流水线后端处理

所有现代图形系统都是基于光栅的。
《计算机图形学》第三版

这个阶段更多的操作是位级别的操作，比如对像素颜色的填充和对片元颜色的混合。

光栅化

光栅化就是将几何图元像素化。我们在给图元定义的时候，一个线段就是两个顶点，4个浮点数，一个三角形就是三个顶点。拿三个不共线的顶点来说吧，这么简单的数据想要在显示器上绘制出来一个完整的面，请问其他数据从何而来？比如三角形中间某一个像素颜色从何而来？位置从何而来？这就是光栅化模块要做的事情，它是利用高效的光栅化算法比如Bresenham（布兰森汉姆）算法、DDA（数字微分分析器）算法等，计算像素位置与差值计算颜色。每一个在视锥体（frustum）内的图元都要进行光栅化，其

结果是将图元细分为片元[1]（fragment），输出会直接进入帧缓存中。在整个过程中，不可避免的会将在视椎体内但被更靠前的物体遮挡的图元也进行光栅化后将片元尝试载入帧缓存，此时会判断片元深度，从而丢掉被压着的片元。

片元处理

片元处理是绘制流水线的最后一道工序，这个阶段才是我们经常说的纹理映射阶段。我们可以采用传统的纹理映射，也可以使用凹凸映射这种比较新的技术。映射结束后颜色缓存中的每个像素的颜色就是纹理必要位置的颜色，再与此前顶点处理阶段对顶点早期指定（或计算）的颜色进行某种混合，就成为最终屏幕像素的颜色了。此处‘某种混合’就OpenGL来说一共有16中，基本上概全了对颜色中任何一位的与或非异或等操作。

1.1.3 性能特征

第一章还说到了性能特征，就是业界对于一款图像显示卡的判断，主要有两条：

1. 能以多块的速度处理几何实体；
2. 每秒能修改多少个帧缓存中像素的颜色；

1.2 总结

交计图中第一章重点我觉得就这些东西，它是一个总述的章节，希望能从全盘先给读者一个轮廓，然后带着问题、好奇与目前的初步理解从第二章开始按照上面的顺序往下走，这也是交计图这本书作者Edward Angel多年教学颇为成功的方式。

[1] 片元可以认为是像素的一个超集，它所承载的数据不仅仅是像素的颜色，还有深度信息等等。

Chapter 2

绘制模式

第二章围绕谢尔并斯基镂垫程序做了图形学宏观上的介绍。传统的绘制模式有典型的两种：

1. 立即绘制模式：就是边计算边提交给OpenGL试图让其绘制；
2. 延迟绘制模式：先计算出全部的需要绘制的数据，比如坐标、颜色等，保存在一定的数据结构中，完事统一提交给OpenGL；

简单分析下这两种方式就会发现：如果需要再次绘制，第一种便需要重新再计算一遍数据；如果数据提交频繁（比如一个物体动来动去），第二种方式就会因内存数据频繁的向GPU提交而成为性能瓶颈。于是OpenGL出现了第三种方案，应该说是前两种方案的折中：采用延迟绘制模式提交，只要模型有变化，重新计算，重新一个批次提。

2.1 点与顶点

《交互式计算机图形学》第六版在OpenGL中一定区分“点”与“顶点”的含义。“顶点”是空间中的一个位置，我们在计算机图形学中使用顶点的空间包括二维的、三维的和四维的。我们利用顶点来定义图形系统中可识别的基本几何图元。比如我们一个点，那它在空间中就需要一个顶点来定义；再比如一条线段，就需要2个空间中的顶点来定义等等。既然OpenGL、D3x都是虚拟照相机模型下的图形系统，那么其API的设计就免不了下面7种类型：

1. **图元函数**：定义了系统可以显示的低级对象或者最基本的实体。视API的不同，图元函数可以包括点、线段、多边形、像素文本和各种曲线；
2. **属性函数**：控制图元在显示器上显示的方式；
3. **观察函数**：可以指定各种视角；
4. **变换函数**：用户可以对对象进行诸如旋转、平移、缩放之类的变换；
5. **输入函数**：为了开发交互式应用程序，图形系统必须提供输入函数；
6. **性能特征**：使得用户可以与窗口系统通信，初始化我们的程序，以及处理在程序执行期间发生的任何错误；
7. **查询函数**：在应用程序中我们可能经常使用API中的其他信息，包括照相机的参数或者帧缓存中的值。

OS	library
X window System	GLX
windows system	wgl
mac system	agl

表 2.1: 与操作系统匹配的图形接口库

图元	枚举值	说明
点	GL_POINTS	每个顶点被显示的大小至少是一个像素
线段	GL_LINES	相继的顶点配对成线段的两个端点
折线	GL_LINE_STRIP	中间顶点也相连，但顶点数组中的首尾顶点不相连
环折线	GL_LINE_LOOP	首尾也相连的折线
三角形	GL_TRIANGLES	相继的每三个顶点被解释为一个三角形
三角带	GL_TRIANGLE_STRIP	每个顶点与前两个顶点组成一个新的三角形
三角形扇	GL_TRIANGLE_FAN	第一个顶点共享， 后续依次减去第2、第3...顶点构成新的三角形

表 2.2: openGL中的图元种类

OpenGL中的参数大多数是持续性的，他们的值保持不变，除非我们调用改变状态的函数显式地改变他们。OpenGL与各个不同的系统之间有一个接口函数库，查看2.1。还有一些与OpenGL没有直接关系，主要聚焦在多平台统一上的库：OpenGL扩展库(GLEW)是个与操作系统无关的库，主要是一些函数、变量等的初始化；OpenGL实用工具库glut（建议使用GLFW库）提供了任何现代窗口系统所需的最小功能集。

2.2 图元的类型

- 几何图元(geometric primitive)
- 图像图元/光栅图元(raster primitive)：光栅图元的一个例子是像素阵列，比如点阵字体，它不具有几何属性，我们不能像对几何图元那样对光栅图元进行几何操作。光栅图元需要经过另一条流水线的处理，这条流水线与几何流水线并行，终点也是帧缓存。

2.2.1 OpenGL中的几何图元

OpenGL中一共有7种图元，查看2.2。其中三角形是支持的唯一多边形，其绘制有三种不同的模式：点模式（GL_POINT）、线模式（GL_LINE）、填充模式（GL_FILL），通过glPolygonMode函数进行指定（点与线图元无效），具体参考官方API指南。

我们把用来描述对象如何被绘制的方式称为属性(attribute)。
《交互式计算机图形学》第六版中文

2.3 颜色

目前常用的颜色是RGBA颜色，四个分量，每个8位，共32位。历史上出于帧缓存深度太小（比如只有8位），而出现过索引颜色模式，这里简单说下索引颜色模式。假如帧缓存深度只有8位，那么只能表

示256种不同的数字（2的8次方），倘若每个数字表示一种颜色的话，这远远不能说是色彩丰富。但是当我们不用这些数字表示颜色，而是用他们表示一个索引，从一个被称为查色表的地方去映射一种颜色的话，那么整个颜色的选择就会增大自由度。换句说话如果用256中数字仅仅表示颜色，那么这些颜色是固定死了的，不同的应用只能使用这些颜色，爱用不用。假如是索引的话，应用A建立一个查找表，里面是256种自己中意的颜色，应用B也建立一个自己的查找表，里面又是自己青睐的色彩，这样增大了颜色可定制可选择的自由度，是一种技术上的进步。OpenGL设置清屏颜色的函数是glClearColor，参数依次是RGBA的四个分量。

2.4 与OS窗口系统的交互

前面的内容说了下GLUT库函数提供了任何现代窗口系统所需要的最小功能集，这就意味着当你需要更多更全面的与系统窗口交互的功能，需要自己写功能。GLUT对于一般的应用是不二的选择，主要提供了以下几个重要函数。

```
glutInit(int argc,char* argv);
glutCreateWindow(char *title);
glutInitDisplayMode(GLUT\_RGB | GLUT\_DEPTH | GLUT\_DOUBLE);
glutInitWindowSize(GLuint w,GLuint h);
glutInitWindowPosition(x,y);
glutMainLoop();
glutMouseFunc();
glutKeyboardFunc();
glutDisplayFunc();
glutIdleFunc();
```

重点注意glutInitDisplayMode函数的参数之间是通过逻辑或结合起来的。

2.5 传入顶点数据

书中是通过OpenGL显卡缓存的方式提交顶点的，这就是之前说的OpenGL提供的第三个方案，这部分内容我看在书后期基本没有再出现，不打算将其作为重点复习，而会在全新的链接中参考第五版蓝宝书第八章缓冲区对象的内容记录。先给出按照传统的glBegin()，glEnd()的方式传递顶点数据给OpenGL的代码，程序如下：

```
void render() {
    glClear(...);
    glBegin(...);
    // upload your data.
    glEnd();
    glFlush();
    glutSwapBuffers();
};
```

2.6 双缓存

现代显卡帧缓存中的数据提交到显示器与应用程序刷新帧缓存是两个不同步的过程（部分设备有这样强制同步的机制），于是就产生一个明显的问题，比如帧缓存正在提交数据，而此时应用程序又

刷新了帧缓存，那么最终呈现到监视器上的画面就是两部分，一般来说是上面一个画面，下面一个画面。双缓存的意义就在于帧缓存的两边各用一个缓存，应用程序所有的提交经过OpenGL后达到后端缓存(back buffer)，而监视器上显示的都是前端缓存(front buffer)的内容。程序员只需要在合适的时候调用glSwapBuffer()函数即可告知显卡将后端缓存中的内容提交到前端。

2.7 着色器

说起着色器，最基本的两个便是顶点着色器、片元着色器，其他的着色器比如细分控制着色器、细分计算着色器、还有几何着色器。拿GLSL着色器语言来说，它也有自己的一套规范，扩展了C语言的数据类型（添加了矩阵和向量）。Edward在第二章首次介绍了着色器的基本用法，并且在镂垫程序中派上用场。

2.7.1 顶点着色器

前面说过应用程序的数据在通过渲染管线的时候，几何图元中的顶点肯定要经过顶点处理模块，该模块会对经过的顶点经行矩阵变换从而将其坐标从物体坐标系下转换到照相机坐标系下。这是一般的情况，程序员完全可以不让其进行坐标变换或者随性乱变换，这都是可以的，而这些“想怎么着就怎么着”的做法就是在顶点着色器中进行的。

```
#version 430
in vec4 vPosition;
void main(){
    gl_Position=vPosition;
}
```

上面是一个很简单的顶点着色器程序，其中我们申明了一个vec4类型的变量vPosition，同时用in修饰表示他是从流水线上游（并不一定是直接上游，见下图）提交过来的输入数据，gl_Position是顶点着色器内置的一个vec4实例的类型，不需要我们再次申明。顶点着色器可以附带处理颜色数据，但是其最基本的作用就是要给流水线下游的模块输出顶点的新坐标，也就是内置的gl_Position必须在输出后至少有数据。当然gl_Position不给他赋值的话一样可以编译通过，并且正常运行程序，此时OpenGL会保证你看不见任何东西。

现在新的问题出现了，如何映射用in修饰的变量与主程序中的数据？比如我有2个in修饰的上游变量，主程序也有2个数据提交，现在谁对应谁？什么类型？（因为OpenGL与GLSL是不同的语言，虽然同属C家族，但实现上不同）；还好OpenGL提供了获取GLSL代码中部分变量的函数glGetAttribLocation，见下程序，我们获取了在顶点着色器中的变量vPosition，将它映射到了loc下，紧接着我们开启对这个变量内容的写入，第三条glVertexAttribPointer()函数是指明了我们传给着色器vec4变量的内容的格式。

```
GLuint loc = glGetAttribLocation(program, "vPosition");
glEnableVertexAttribArray(loc);
glVertexAttribPointer(loc, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
```

2.7.2 片元着色器（片段着色器）

[!quote] 《交互式计算机图形学》第六版光栅化模块输出位于视见体内部的每个图元的片元。每个片元都会被片元着色器处理。每次执行片元着色器时必须至少输出每个片元的颜色，除非这个片元在片元着色器中被标记为丢弃。

```
#version 430
void main(){
    gl_FragColor=vec4(1.0,0.0,0.0,1.0);
}
```

上面就是个简单的片元着色器程序，可以看出它有一个与gl_Position很相近的变量gl_FragColor。联想可知其做表达的应该是最终的片元颜色，没错！从OpenGL3.0开始片元颜色的输出也可以使用另一种形式，如下：

```
#version 430
out vec4 fColor;
void main(){
    fColor=vec4(1.0,0.0,0.0,1.0);
}
```

其中out修饰的便是输出变量，每个片元都经过上面程序处理后都输出了红色。片元着色从之前固定的模式到现在自定义起名可以看出，OpenGl的发展是适应现代图像系统完全可编程的理念的。

2.7.3 编译、连接着色器程序

当程序员写完了各种着色器程序后，不能光在外部保存个文件或者在程序里面写完放着。我们需要编译每个着色器代码，然后连接到被称为程序对象(Program Object)的对象上才能使用。把着色器连接到程序对象的过程中会产生一个表，着色器中的变量名会与表中的索引相关联。首先利用glCreateShader方法创建一个着色器对象(Shader Object)，然后将着色器代码传入对象中，紧接着编译它（编译可能会失败，可以通过glGetShaderiv()获取一些失败的信息，方便调试）。编译成功后将着色器对象附加到程序对象上，当全部你想要的着色器对象附加完成后，就可以尝试将程序对象连接到主程序中。

```
//创建程序对象
GLuint program = glCreateProgram();
//创建顶点着色器对象
GLuint vShader=glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vShader, 1, (const GLchar**)&s.source, NULL);
glCompileShader(vShader);
glAttachShader(program, shader);
//创建片元着色器对象
GLuint fShader=glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fShader, 1, (const GLchar**)&s.source, NULL);
glCompileShader(fShader);
glAttachShader(program, fhader);
//链接;
glLinkProgram(program);
```

注意

交计图第五版书后面的着色器程序写好后，请务必用LF结尾保存，CR或者CRLF都不会顺利被着色器对象编译，这个问题暂时没有查到相关文章，期待解答；

2.8 总结

本章 [1]用一个镂垫程序为引子，重点介绍了一个普通的OpenGL程序基本的结构、程序与OpenGL交互的基本方式方法，扩展说了下颜色，着色器。这里是一些相关的资料：
https://www.youtube.com/watch?v=8p76pJsUP44
https://github.com/opentk/opentk/issues/18

[1]为了更清楚的说明镂垫程序，第二章还简单说了下观察与坐标范围，我会在第四章的时候说明。

Chapter 3

三维图元

二维图元比较简单，比如点、线段、三角形等，但毕竟我们生活在三维空间中，不可避免要涉及到三维图元的思考。在三维图元中，会遇到至少两个问题：

1. 三维图元的数学定义可能会很复杂；
2. 三维对象非常多，我们只能对图形系统能有效实现的对象感兴趣，现有的图形系统不能支持所有的三维对象，除非通过近似的方法；

图形学发展至今，前辈们选择了使用网格的形式描述三维对象。这些三维对象具有以下特征：

1. 可以通过其表面来描述外观，并且认为对象是中空的；
2. 可以由三维空间中的一组顶点来确定，表面要么可以用二维空间中的平面凸多边形来组成，要么可以通过它们来近似；

3.1 标架

我们在中学开始接触的解析几何就是在三维空间中进行的，其中三维空间是建立在坐标系中。想想那个时候的坐标系有什么东西吧：一个被称为原点的点，外加3个互相垂直的向量，向量虽然没有位置的概念，但这里的三个向量必须要以原点为起始点去建立。标架的描述性定义于此类同但不限于刚才的描述，具体而言就是去掉“相互垂直”与“三维”这两个约束，即：一个指定的点（被称为原点）外加空间中的一个基（基的构建要以原点为起点）。至于标架与坐标系有什么关系，我觉得坐标系范围更广比如还有极坐标系，球面坐标系等等而且坐标系这个概念更通俗，就是哪个学科都能说“来，我们先建立个坐标系”，而标架的概念应该更加专业，更加与具体的科学相关。现在说一下格式中“原点”的含义，原点无非空间中的一个特殊的点，而点是离不开具体标架的，从这层含义来说当指定一个具体标架后，其中原点的定义是与该标架的基相关的，只不过各个分量为0。但千万不要与其他标架下的原点混淆，认为他们是同一个标架，更不要认为是同一个点，所以不同标架下原点之间的关系依然是建立在具体标架下的。由于三维标架是定义在空间中的三个方向外加一个原点组成，那么我们一般将其表示为：

$$\mathbf{frame} ::= \begin{Bmatrix} \mathbf{v1} \\ \mathbf{v2} \\ \mathbf{v3} \\ \mathbf{O} \end{Bmatrix}$$

$v1$，$v2$，$v3$是三个基向量，代表该标架的三个方向，O是原点，也就是标架开始的地方。写一个具体的标架示例：一个原点在$(28.999, 49, 99)$，三个基向量分别为$(2e^{99}, 1, 3)$，$(5, -7.4, 9)$，$(1, 1, -8)$的标架（无所

谓它是什么样子，是否正交，这就是范概念下的标架）：

$$\textbf{frame} ::= \left\{ \begin{array}{cccc} \textbf{v1} & \textbf{v2} & \textbf{v3} & \textbf{O} \\ \hline 2e^{99} & 5 & 1 & 28.999 \\ 1 & -7.4 & 1 & 49 \\ 3 & 9 & -8 & 99 \end{array} \right\}$$

3.1.1 齐次表示

我们在中学就开始接触三维空间，比如当时的解析代数。三维空间中的“点”与“方向”的表示是一样的，都是形如(x, y, z)，如果将这种没有明确区分点与向量表达式的方式用在计图里面，那将是一片混乱，最简单的来说我们就难以使用流水线形式处理几何对象，于是发展出来用四位齐次坐标的形式描述他们（至于为什么叫“齐次”，我始终没有自我感觉理解到位）。我们用$(x, y, z, 1)$表示空间中一个点，用$(x, y, z, 0)$表示空间中一个向量，其中0，1可不是随意安排的，他们是来自数学定义。

我们知道三维空间中方向是无所谓位置的，它只需要基向量的放大倍率，可以表示成这样：

$$\vec{v} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{3.1}$$

$$= (x, y, z) \tag{3.2}$$

而点是存在位置关系的，需要有一个参考点（没有可以认为是原点）为基准，沿着具体的方向移动后得到，可以表示成这样：

$$P = \vec{v} + P' \tag{3.3}$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + P' \tag{3.4}$$

$$= (x + P'_x, y + P'_y, z + P'_z) \tag{3.5}$$

从上面可以看到，要计算一个方向，只需要矩阵点乘运算，而要得到点，还需要额外一次向量加法运算，为了方便硬件做批处理，索性结合在一起，期望能用一次矩阵运算得到结果。通过矩阵点乘的运算规则知道，只需要给标架加上第四行即可。

3.1.2 新的标架表示

用四维矩阵表示标架一个很明显的问题就是矩阵第四行与向量的第四分量具体是什么，尝试用解方程的方法，假设参数全是未知量：

$$P = \begin{bmatrix} 1 & 0 & 0 & P'_x \\ 0 & 1 & 0 & P'_y \\ 0 & 0 & 1 & P'_z \\ i & j & k & l \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ m \end{bmatrix} \tag{3.6}$$

$$= (x + P'_x m, y + P'_y m, z + P'_z m, ix + jy + kz + lm) \tag{3.7}$$

很显然如果是表示点，m必须是1，这样才能保证P的前三个分量是空间正确的，而m必须是0才能保证方向运算的正确性。其他未知量保证结构一致性即可，它们只能是$(0, 0, 0, 1)$。上式从数学角度看没有问题，但是从仿射空间的定义来看有错误的地方，错误出在仿射空间没有定义标量与点的运算，所以我们需要约

定：$1P = P$ ，意思是标量1与任何一个点相乘还是这个点，没有约定其它非零标量与点相乘的含义，例如不能刻意认为标量3乘以点P，是将点P放大3倍或者其他什么意思。

到目前我们至少明确区分了在计算机图形学中位置与方向的表示：点是$(x, y, z, 1)$，方向是$(x, y, z, 0)$，将原本可以用三维表示并计算的方向补齐到了四维，没准这也是为什么叫“齐次表示”的一个原因吧。

3.2 标架变换

标架变换是个很重要的概念，涉及到的问题就是：空间中同一个点在不同的标架下分别是什么表示，怎么互相转换。我们先布置一个环境：有两个标架Fv与Fu，他们之间存在以下关系：

$$Fu_x = 2Fv_x \tag{3.8}$$

$$Fu_y = Fv_x + Fv_y \tag{3.9}$$

$$Fu_z = Fv_x + Fv_y + 3Fv_z \tag{3.10}$$

$$Fu_o = Fv_o + (2, 2, 2, 0) \tag{3.11}$$

其中$(2, 2, 2, 0)$是v标架下的一个方向，表示v下的原点需要移动自己标架下定义的$(2, 2, 2, 0)$这个方向后就会与标架u在空间中指定的原点$u0$重合。现在要求计算点P在u标架下的表示。我们先将标架Fv与Fu的关系写成矩阵的形式：

$$\begin{bmatrix} Fu_x \\ Fu_y \\ Fu_z \\ Fu_o \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 3 & 0 \\ 2 & 2 & 2 & 1 \end{bmatrix} \begin{bmatrix} Fv_x \\ Fv_y \\ Fv_z \\ Fv_o \end{bmatrix}$$

假设点Fu标架下有个点$Pu = (d, e, f, 1)$，与Fv标架下的$Pv(a, b, c, 1)$在空间中位置一致，于是：

$$Fv(a, b, c, 1)^T = Fu(d, e, f, 1)^T = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 3 & 0 \\ 2 & 2 & 2 & 1 \end{bmatrix} Fv(d, e, f, 1)^T$$

如果说Fv标架是单位阵，那么可以得到$(a, b, c, 1)$与$(d, e, f, 1)$之间的关系：

$$(a, b, c, 1)^T = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 3 & 0 \\ 2 & 2 & 2 & 1 \end{bmatrix} (d, e, f, 1)^T$$

上面等式就说明了同一个空间下两个不同的标架之间的变换关系，从等式中可以看到u标架下的任何一个点的坐标右乘矩阵M就可以得到该点在v空间下的坐标。最基本的变换便是**平移**、**缩放**、**旋转**、这三种了，理解一下**错切**也是不错的。

3.2.1 旋转

旋转相对复杂，我们先在二维平面说明。想象平面上有一个点$P(x, y)$，它与x正向夹角ϕ弧度，围绕原点旋转了θ弧度后来到位置$P'(x', y')$，两点的位置用极坐标表示为：

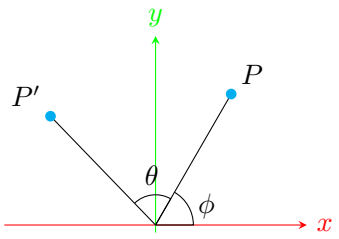

图 3.1: 2D平面中顶点围绕原点旋转θ弧度

$$x = L\cos\phi \tag{3.12}$$

$$y = L\sin\phi \tag{3.13}$$

$$x' = L\cos(\phi+\theta) = L\cos\phi\cos\theta - L\sin\phi\sin\theta = x\cos\theta - y\sin\theta \tag{3.14}$$

$$y' = L\sin(\phi+\theta) = L\sin\phi\cos\theta + L\cos\phi\sin\theta = x\sin\theta + y\cos\theta \tag{3.15}$$

写成矩阵形式就是：

$$P' = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} P \tag{3.16}$$

这是否意味着在3D空间中，这样的旋转就是P在xoy平面上，绕着z轴旋转了θ弧度呢？扩展到3D空间，其旋转矩阵为：

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.17}$$

空间中物体围绕标架的z轴旋转也是这样，不过绕y就需要注意了，这并不是特殊情况，而是在右手标架下，绕y旋转是向着x到z的大夹角方向旋转，但在纯2D平面上显然是小角度方向，故而想要旋转θ角度，取相反数即可：

$$\begin{bmatrix} \cos(-\theta) & 0 & -\sin(-\theta) \\ 0 & 1 & 0 \\ \sin(-\theta) & 0 & \cos(-\theta) \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \tag{3.18}$$

2D绕任意点的旋转

初学矩阵变化可能会犯这样的错误，就是将“旋转”想当然的认为是物体绕自己的旋转，这当然是错误的，一定需要知道正因为用明确了坐标的点定义了多边形，那这个多边形就是定义在了点所在的空间，旋转只是针对空间原点的转动，这个一定要注意。我们处理物体绕定点旋转的思路一般是在旋转位置建立个新的空间（以旋转点为原点），将物体所有的顶点变换到新的空间后再旋转，最后将新的顶点位置从新的空间转化回此前的空间。简单来说就是**平移旋转平移**。图3.2中$P(1,2)$点是旋转中心，先将其移回标架原点，这是最简单的平移矩阵T，之后使用旋转矩阵R，最后平移回原来的位置使用矩阵T^{-1}，这也就好了。

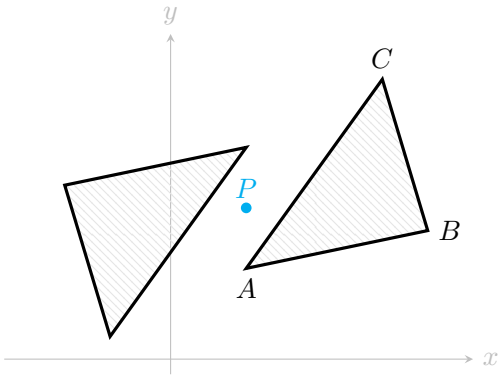

图 3.2: 三角形绕任意点旋转

OpenGL中的物体是靠图元来定义的，而图元是由顶点来组成的，所以物体的变换其实质是组成物体的点的变换，因此变换顶点就是变换物体。我们将刚才的系列变化写成矩阵如下（图中ΔABC旋转180角度）：

$$T^{-1}RT \tag{3.19}$$

$$= \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\pi & -\sin\pi & 0 \\ \sin\pi & \cos\pi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -2 \\ 0 & 1 & -3 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.20}$$

$$= \begin{bmatrix} \cos\pi & -\sin\pi & -2\cos\pi + 3\sin\pi + 2 \\ \sin\pi & \cos\pi & -2\sin\pi - 3\cos\pi + 3 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.21}$$

绕任意给定直线的旋转

这个问题比起上一节明显具体多了，我们不去计较物体到底是围绕着给定的直线旋转还是物体围绕着以给定直线的方向为轴自己的重心旋转，这都不重要，这节的重点就是我给你一个方向，并且指定一个物体，要你沿着这个方向旋转这个物体。我们此前的讨论都是围绕物体如何绕标架的某一个具体的基的分向量旋转，说到这里可能会有灵感爆发：绕任意直线的旋转可以分成两步，第一步想办法通过一个矩阵N将给定直线旋转到与某一条基的分量平行的状态；第二步就是之前的旋转M。计算出来后，将他们级联（注意顺序），就是最后的变换矩阵了。看完交计图第三章，第四章多遍后，结合网络上的其他文章，就会发现Edward有意将级联思想灌输给读者：任何复杂的变换，我们都可以用许多基本的变化通过级联的方式得到最终的变换矩阵。这个思想我觉得很重要，一来方便硬件流水线化计算，二来很清晰的展现了矩阵变化的思想，三来有助于理解复杂变换。我们先将“直线”抽象为“单位向量”，并且将空间中这条向量平移到起始点与标架原点重合的位置。紧接着将向量旋转到xoz平面。除了将直线围绕z轴以外，围绕x轴也可以，不过大家都采用x轴。θ是要旋转的角度。从(3.16)看出不必非要计算出θ的值，只需要知道$\cos\theta$与$\sin\theta$即可。我们只需要将向量l旋转θ_x角度后，就能到xoz平面。而θ_x就是向量d与z轴之间的夹角，而d是l在yoz平面的投影。这样一来$\cos\theta_x$与$\sin\theta_x$都能算出来，分别是：$\cos\theta_x = \frac{\alpha_z}{|d|}, \sin\theta_x = \frac{\alpha_y}{|d|}$ 写成矩阵形式：

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{\alpha_z}{|d|} & -\frac{\alpha_y}{|d|} & 0 \\ 0 & \frac{\alpha_y}{|d|} & \frac{\alpha_z}{|d|} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

就是将l旋转到xoz平面后的图像，我当年这里有个小小的理解错误，就是将图中淡绿色向量（就是l）理解成了图中l在xoz平面的投影，大家注意，图中的l不是原来的投影，而是向量l旋转θ_x后的位置。现在就

是将图中向量l再次旋转到与z轴重合，很明显通过围绕y轴就可以做到，图中所示旋转θ_y度便是，明显能够知道：

$$\cos\theta_y = \frac{|d|}{|l|} = |d| \tag{3.22}$$

$$\sin\theta_y = \frac{\alpha_x}{|l|} = \alpha_x \tag{3.23}$$

$$M = \begin{bmatrix} \|d\| & 0 & -\alpha_x & 0 \\ 0 & 1 & 0 & 0 \\ \alpha_x & 0 & \|d\| & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.24}$$

看完这个矩阵，如果不假思索，可能会觉得两处α_x的符号应该互换。但实际情况是不互换，不要被图中的θ_y角度所欺骗，它只是标注了需要旋转的角度，实际情况是向量l旋转了$-\theta_y$度。最后再沿着z轴旋转，这之前我们讨论过。

四元数旋转*[1]

作为图形开发者，如果你不是学术派的话，最起码应该知道四元数怎么用。四元数是复数的**超集**，就像复数是实数的超集一样，是“数”在广度上的进一步扩展，记作$q = (w, \mathbf{v})$，其中w表示实部系数，$\mathbf{v}$是用向量表示的虚部，其每个分量依次是虚部的系数，q也可以写成$(w+x\mathbf{i}+y\mathbf{j}+z\mathbf{k})$或者$(w, < x, y, z >)$，或者直接$(w, x\mathbf{i}, y\mathbf{j}, z\mathbf{k})$写成向量形式，$\mathbf{i}, \mathbf{j}, \mathbf{k}$是虚数单位，与复数范畴使用的$\mathbf{i}$并无差别：$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = -1$，不过有一些扩展出来的规则需要记住：

$$\mathbf{ij} = \mathbf{k} \quad \mathbf{jk} = \mathbf{i} \quad \mathbf{ki} = \mathbf{j}$$
$$\mathbf{ik} = -\mathbf{j} \quad \mathbf{kj} = -\mathbf{i} \quad \mathbf{ji} = -\mathbf{k}$$

可以把**ijk**重复至少2遍写在纸上，从左往右任意连续的两个相乘就是第三个，而从右往左任意连续的两个相乘是第三个的相反数。

ijkijkijkijkijkijkijkijkijkijkijkijkijkijk

从新的规则中可以知道四元数的乘法**不具有交换性**，注意这里说的是四元数之间的乘法，不包括它与标量的乘除运算，这与矩阵乘法一样。至于基本运算（加减乘除）与多项式运算没有区别，该分配的分配，该合并的合并。

$$q_1 + q_2 = (w_1 + w_2, \mathbf{v_1} + \mathbf{v_2}) \tag{3.25}$$

$$q_1 q_2 = (w_1 + x_1\mathbf{i} + y_1\mathbf{j} + z_1\mathbf{k})(w_2 + x_2\mathbf{i} + y_2\mathbf{j} + z_2\mathbf{k}) \tag{3.26}$$

$$= w_1w_2 + w_1x_2\mathbf{i} + w_1y_2\mathbf{j} + w_1z_2\mathbf{k} + \tag{3.27}$$

$$x_1\mathbf{i}w_2 + x_1\mathbf{i}x_2\mathbf{i} + x_1\mathbf{i}y_2\mathbf{j} + x_1\mathbf{i}z_2\mathbf{k} + \tag{3.28}$$

$$y_1\mathbf{j}w_2 + y_1\mathbf{j}x_2\mathbf{i} + y_1\mathbf{j}y_2\mathbf{j} + y_1\mathbf{j}z_2\mathbf{k} + \tag{3.29}$$

$$z_1\mathbf{k}w_2 + z_1\mathbf{k}x_2\mathbf{i} + z_1\mathbf{k}y_2\mathbf{j} + z_1\mathbf{k}z_2\mathbf{k} \tag{3.30}$$

$$= (w_1w_2 - \mathbf{v_1} \cdot \mathbf{v_2}, \mathbf{v_1} \times \mathbf{v_2}) \tag{3.31}$$

另外还需要知道四元数的**共轭**，就是简单的将虚部取反即可，新的四元数$q^* = (w, -\mathbf{v})$与q互为**共轭关系**，共轭四元数的标记法也是就刚才写的样子：$\mathbf{q}^*$。单位四元数顾名思义就是所有系数的**平方和**等于1，

[1] 这些内容是我2025年整理的，并非当时的内容，理解有限表达可能有失偏颇。

要归一化一个四元数，将其各个系数除以平方和的开方即可：

$$q = (wA, xA\mathbf{i}, yA\mathbf{j}, zA\mathbf{k}) \tag{3.32}$$

$$where: \tag{3.33}$$

$$A = \frac{1}{\sqrt{w^2 + x^2 + y^2 + z^2}} \tag{3.34}$$

上面就是使用四元数旋转顶点的全部基础知识，现在只需要记住两条等式即可：

$$q = \left[\cos(\frac{\theta}{2}), \sin(\frac{\theta}{2})\mathbf{u}\right] \tag{3.35}$$

$$v' = q\mathbf{v}q^* \tag{3.36}$$

其中$\mathbf{v}$表示要旋转的顶点，用四元数表示为$(0, < x, y, z >)$，它实部系数为0是个纯虚四元数。$\mathbf{u}$是旋转轴（也就是方向），θ是要旋转的角度，**$\mathbf{v}'$就是顶点$\mathbf{v}$绕着$\mathbf{u}$这个轴，正方向旋转θ弧度后新的位置**。注意右手系使用右手法则确定旋转的正向，左手系标架同理。写一个示例巩固下，计算顶点$\mathbf{v}(1,0,0)$绕着$l(0,2,0)$正方向旋转45角度后的位置。

$$l' = (0,1,0) \tag{3.37}$$

$$q = \left[\cos(\frac{\pi}{8}) + \sin(\frac{\pi}{8})(0,1,0)\right] = (\alpha + \beta\mathbf{j}) \tag{3.38}$$

$$q^* = (\alpha - \beta\mathbf{j}) \tag{3.39}$$

$$v = (0 + 1\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}) = (\mathbf{i}) \tag{3.40}$$

$$v' = qvq^* \tag{3.41}$$

$$= (\alpha + \beta\mathbf{j})(\mathbf{i})(\alpha - \beta\mathbf{j}) \tag{3.42}$$

$$= (\alpha\mathbf{i} - \beta\mathbf{k})(\alpha - \beta\mathbf{j}) \tag{3.43}$$

$$= (\alpha^2\mathbf{i} - \alpha\beta\mathbf{k} - \beta\alpha\mathbf{k} - \beta^2\mathbf{i}) \tag{3.44}$$

$$= (0.707\mathbf{i} - 0.707\mathbf{k}) \tag{3.45}$$

得到的新位置从四元数转成坐标后为$(0.707, 0, -0.707)$。如果是要计算反方向旋转，要么使用$-\theta$重新计算，要么将已经生成的四元数q与q^*变换一下顺序直接计算：$v' = q^*vq$，若要理解背后的原理，请查阅wiki：

3.2.2 错切，切变（shear）

错切变换好比对象自身将自己一层一层偏离了原来的地方，如图所示。原来的物体是淡绿色方形，经过错切变换后成了淡粉色形状，D、C两点不变，A点移动到了A'，B点移动到了B'（其y坐标没有变化）。从图中我们明显能看到y值越大，其偏里原来坐标的程度就越强（参考F点的偏移量），现在写出A与A'的坐标关系：$A'_x = A_x + A_y \cot\alpha$，写成矩阵的形式：

$$A' = \begin{bmatrix} 1 & \cot\alpha & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} A$$

3.2.3 变换的级联

级联，就是将一系列基本变换合在一起达到组合表现的效果，级联需要注意的是前后变换矩阵的顺序。从矩阵的级联中可以看出，点A先旋转α角度后再平移a就是点C，而先平移再旋转就是点D。我们一

般将矩阵写在左边（点A的左边），形如$A' = TA$，就是说点A进行T变化。变化后如果需要后续变化的话，继续写在左边（最左边），形如$A' = RTA$，意思就是，先进行T变化，再进行R变化。因为矩阵乘法满足结合律，所以我们可以先计算出RT相乘的结果M，然后再应用到A点，就像这样$A' = (RT)A$，别小看这么一个简单的优化，渲染管线中一个批次要处理少则几千几万，多则几十万甚至上百万的顶点，倘若每个顶点都先乘T再乘R，肯定比只乘M理论上慢一倍。而将矩阵R，T相乘试图在以后减少对顶点的变化次数，这个思想就是级联。

3.3 uniform变量

第三章后面捎带说了下shader中的uniform变量，它是用来修饰变量的，从单词意思就初步能了解该变量在一帧渲染的过程中是不变的。

```
uniform float uTime;
attribute vec4 vPosition;
void main(){
    vPosition.x=1.0+sin(uTime);
    gl_Position=vPosition;
}
// shader中uniform变量的值是这么与应用程序关联的:
GLint timeParam;
timeParam = glGetUniformLocation("uTime");
glUniform1f(timeParam,100.0);
```

3.4 总结

本章前面几节说了下简单的线性代数，比如向量加法、减法、内积、外积、凸性、直线、平面什么的，非常基础。核心内容是变换，我觉得主要抓住对不同仿射空间变换的理解为主，这将为第四章铺好道路，章节最后出现了四元数，这小部分内容对于交计图的基础部分（一到七章）没有用，应该到更后面的层级建模、过程建模里面讲到骨骼的时候可能会更多的用到。

Chapter 4

投影

图形学中一般使用两种投影：平行投影与透视投影。所谓平行投影，就是投影线彼此平行，可以分为平行正投影与平行斜投影，而透视投影就是“近大远小”的视觉投影。投影的目的就是将相机空间下的一块自定义区域变换到指定空间下的一块固定区域，从而将他们显示在屏幕上。这块固定的区域就是规范化视见体（canonical view volume，CVV）[1]，它是一个左手系下，边长为2的轴对齐立方体4.1。

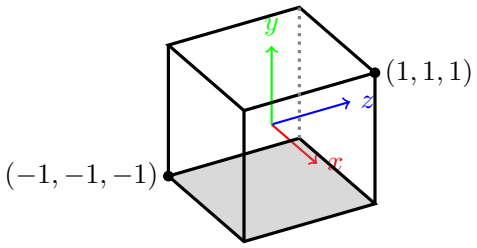

图 4.1: canonical view volume

4.1 平行投影

4.1.1 平行正投影

平行正投影不但需要投影线彼此平行，还要与投影平面垂直，这种投影也就是**正交投影**。由于投影线彼此平行，所以投影的图像也不会产生“近大远小”的视觉效果，4.2就是个平行正投影，侧视图也一样。从4.2中可以看出，想要将相机空间下的视见体转换到CVV大小与位置，分成三步进行：

1. 将相机空间E从右手系变成左手系，新的空间我们定义为E'4.3(a)。这个变换是必须的，因为流水线后面阶段在判断z序的时候，是根据**z值越大越离照相机远**的逻辑来进行的；

2. 将视见体中心平移到E'原点4.3(b)；

3. 缩放视见体成CVV一致的大小4.3(c)；

[1]后面用“视见体”表示自定义区域，用CVV表示规范化视见体。

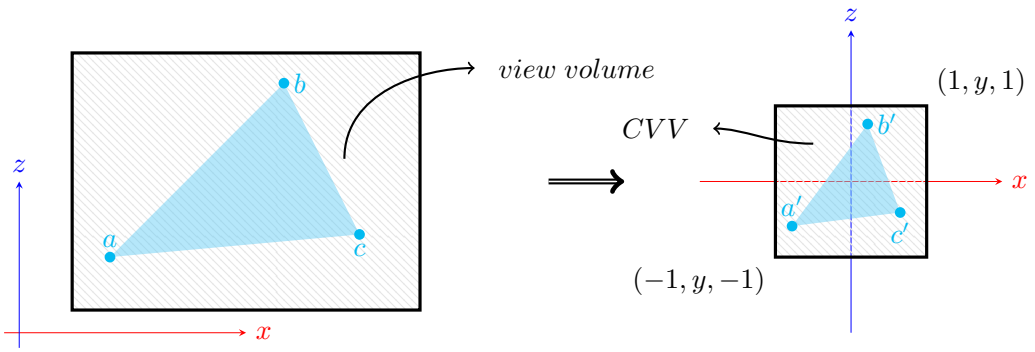

图 4.2: 左手系（顶视图）下的视见体投影到CVV

由于屏幕坐标经常指定为左手系，因此规范化观察体也常指定为左手系统。这样就可以将观察方向的正距离解释为离屏幕（观察平面）的距离。

《计算机图形学》中文第三版P.298

Because screen coordinates are often specified in a left-handed reference frame, normalized coordinates also are often specified in a left-handed system. This allows positive distances in the viewing direction to be directly interpreted as distances from the screen(the viewing plane). Thus, we can convert projection coordinates into positions within a left-handed normalized-coordinate reference frame, and these coordinate positions will then be transferred to left-handed screen coordianteds by the viewport transformation.

《计算机图形学》英文第四版P. 344

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad T = \begin{bmatrix} 1 & 0 & 0 & -\frac{m_x+M_x}{2} \\ 0 & 1 & 0 & -\frac{m_y+M_y}{2} \\ 0 & 0 & 1 & -\frac{m_z+M_z}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad S = \begin{bmatrix} \frac{2}{M_x-m_x} & 0 & 0 & 0 \\ 0 & \frac{2}{M_y-m_y} & 0 & 0 \\ 0 & 0 & \frac{2}{M_z-m_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(a) 转为左手系　　(b) 平移中心到E'原点　　(c) 缩放到CVV大小

图 4.3: 平移与缩放

在计算之前，先布置好场景及其变量：$\forall\{(m_x, m_y, m_z), (M_x, M_y, M_z) \in E'(\mathbb{R}^3) | M_i > m_i\}$ 。将矩阵级联，得到变换矩阵M_{orth}为：

$$M_{orth} = STI \tag{4.1}$$

$$= \begin{bmatrix} \frac{2}{M_x-m_x} & 0 & 0 & 0 \\ 0 & \frac{2}{M_y-m_y} & 0 & 0 \\ 0 & 0 & \frac{2}{M_z-m_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{m_x+M_x}{2} \\ 0 & 1 & 0 & -\frac{m_y+M_y}{2} \\ 0 & 0 & 1 & -\frac{m_z+M_z}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

$$= \begin{bmatrix} \frac{2}{M_x-m_x} & 0 & 0 & -\frac{m_x+M_x}{M_x-m_x} \\ 0 & \frac{2}{M_y-m_y} & 0 & -\frac{m_y+M_y}{M_y-m_y} \\ 0 & 0 & -\frac{2}{M_z-m_z} & -\frac{m_z+M_z}{M_z-m_z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.3}$$

在常见的图形引擎中，一般将m_i与M_i用更直观的名词替代，它们是left，right，bottom，top，near，

far[2]，其中near，far称作近远裁剪面，表示与相机正方向的距离，既然是距离应该大于0才对，但这只是在直观范畴内这么叫，或者被人一厢情愿的这么理解，但实际上它们只需要有这样的关系：

$$left < right \tag{4.4}$$

$$bottom < top \tag{4.5}$$

$$near < far \tag{4.6}$$

不过需要注意的是，该矩阵前提条件是约定相机空间是右手系的，如果在写图形引擎时候团队约定相机一律使用左手系，那么就不用乘以I矩阵，另外m_i与M_i是在E'空间定义的，这点希望多多注意。改成直观后的矩阵：

$$M_{orth} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{l+r}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{b+t}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{n+f}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.7}$$

4.1.2 平行斜投影*

平行斜投影就是投影线彼此平行，但投影线与投影平面不垂直的投影。下图就是一张平行斜投影的截图，从中可以简单的看出多边形$ABCDE$通过绿色投影线u投影到了平面f上，而我们处理它的方案是先将多边形$ABCDE$通过平行于f的方向错切(section 3.2.2)到多边形$A'B'C'D'E'$的位置，然后再进行平行正投影。

所以我们将问题的核心集中在如何通过错切使$ABCDE$变成$A'B'C'D'E'$。从图4可以看出(x, z)是多边形上的任何一个点，假设为P点，它想要移动到正投影的角度，必须向x轴正方向移动$(x_{p'} - x)$的距离。而这个距离正好是$\cot\theta$的长度（包括符号），所以错切后的点P的位置就应该是$(x + z\cot\theta)$，从图（b）中也可以看到错切后的y值应该是$(y + z\cot\phi)$，这样一来，整个平行斜投影进一步变成了平行正投影，所以这个错切矩阵应该是下面这样：

$$\begin{bmatrix} 1 & 0 & cot\theta & 0 \\ 0 & 1 & cot\phi & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

这样一来，整个平行斜投影进一步变成了平行正投影，将矩阵N与前面的M_{orth}级联起来就是了。

4.2 透视投影

透视投影就是大家常见的投影后看起来很自然的有“近大远小”视觉效果的投影，远处的物体之所以投影后要变小，是因为其与COP的距离相比其它物体要远，这就给我们一个处理透视投影的思路：可以用距离来决定物体要缩放多少，这种透视被称作**非均匀的透视缩短（nonuniform foreshortening）**。

问题的核心依然是利用一个怎样的矩阵，可以将自定义的四棱台变换到CVV大小，不要忘记我们变换的重要思想：级联，所以完全可以尝试将四棱台变换成平行正投影下的轴对齐长方体，然后与正交投影矩阵级联得出最终透视投影结果。4.4所示任意一点$a(x, y, z)$，将其投影到近裁剪面后，新坐标的x'分量变小了，通过相似三角形很容易计算出来$x' = near\frac{x}{z}$，y轴也是一样的，而z分量不变，写在一起后：

$$P' = \begin{bmatrix} \frac{n}{z} & 0 & 0 & 0 \\ 0 & \frac{n}{z} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

[2]公式中使用l，r，b，t，n，f依次表示它们

上面计算没有问题，但是初始化该矩阵需要用到各个顶点的z值，整个逻辑是先确定顶点$(x, y, z, 1)$，然后拿着z值确定矩阵，最后回头再去变化顶点，这就意味着每个顶点需要配备唯一的变换矩阵。而图形管线想要的是事前确定好矩阵，然后去处理一堆顶点。尝试后发现先无法确定一个固定的四维矩阵来完成上面的变化。其实出现这样的问题也很简单，透视变换本就**不是线性变换**，不能使用矩阵变换得到，因为它是**线性代数**的范畴。

但办法总比问题多，前辈们通过一个叫透视除法的后期固定管线完成了四维矩阵无法完成的分量乘除运算。简单来说是这样的：因为整个变换使用的都是四维向量，w分量在三维世界中没有意义，可以说是为了“迎合”3×4标架矩阵能顺利进行而添加上的。那么我们何不在透视投影中利用w分量先保存顶点的z值（或者相关的值），等以后再利用？尝试将上面的计算分成两步，第一步先乘以$near$，第二步除以z。

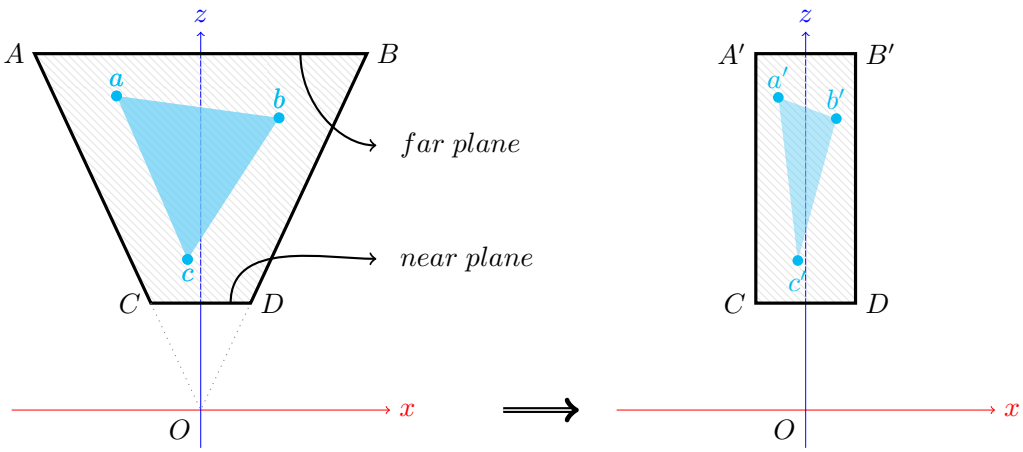

图 4.4: 透视投影顶视图

4.2.1 透视除法

渲染管线在完成MVP变换后，会对每一个四维齐次坐标执行一次名叫**透视除法**的运算，它非常简单就是将每一个分量除以w分量的值，在工程实现上，姑且认为前后相等，就像这样：

$$\begin{pmatrix} x & y & z & w \end{pmatrix}^T = \begin{pmatrix} \frac{x}{w} & \frac{y}{w} & \frac{z}{w} & 1 \end{pmatrix}^T$$

看样子就是缩小了w倍，那为什么叫“透视除法”？这需要联系整个管线来说，因为矩阵变换是线性关系的组合，无法实现“透视”的效果，而w本身保存着z的数据，只有除以z后，才能满足非线性的变换，模拟出透视的效果，其实变换后的顶点依然在三维笛卡尔空间，只是将远处的模型缩小了距离倍，以实现透视的效果，故此将本应该叫“w分量归一”或者“顶点化”的计算取名“透视除法”。

既然管线会在固定的阶段，固定的除以w，完全可以将顶点z值保存进w分量，以达到除以z的目的，省去人为运算。

$$P' = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} nx \\ ny \\ z^2 \\ z \end{pmatrix} = \begin{pmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ z \\ 1 \end{pmatrix}$$

新的矩阵简单一些，但依然存在z的情况，属于定制矩阵，尝试从矩阵计算过程入手，专注看第三行，令它是(a, b, c, d)；

$$ax + by + cz + d = z^2 \tag{4.8}$$

$$a \text{ and } b \text{ must be } 0 \tag{4.9}$$

$$z^2 - cz - d = 0 \tag{4.10}$$

已知near与far是事先给定的裁剪距离，带入方程：

$$\begin{cases} n^2 - cn - d = 0 \\ f^2 - cf - d = 0 \end{cases} \Rightarrow \begin{cases} c = n + f \\ d = -nf \end{cases}$$

得到最终的形式，记作矩阵N：

$$N = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

4.2.2 矩阵级联

现在级联起来平行投影矩阵M_{orth}，注意需要先转化坐标系：

$$(ST)(NI) \tag{4.11}$$

$$= \begin{bmatrix} \frac{2}{M_x - m_x} & 0 & 0 & 0 \\ 0 & \frac{2}{M_y - m_y} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & \frac{m_x + M_x}{-2} \\ 0 & 1 & 0 & \frac{m_y + M_y}{-2} \\ 0 & 0 & 1 & \frac{n+f}{-2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.12}$$

$$= \begin{bmatrix} \frac{2n}{M_x - m_x} & 0 & \frac{m_x + M_x}{M_x - m_x} & 0 \\ 0 & \frac{2n}{M_y - m_y} & \frac{m_y + M_y}{M_y - m_y} & 0 \\ 0 & 0 & -\frac{n+f}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{4.13}$$

透视可视体一般是正棱台，且其是以相机空间z方向对称的，另外设水平张角是θ，垂直张角是ϕ：

$$min_x + Max_x = 0, Max_x - min_x = 2Max_x \tag{4.14}$$

$$min_y + Max_y = 0, Max_y - min_y = 2Max_y \tag{4.15}$$

$$\tan\frac{\theta}{2} = \frac{Max_x}{n}, \tan\frac{\phi}{2} = \frac{Max_y}{n} \tag{4.16}$$

$$M_{perp} = \begin{bmatrix} \cot\frac{\theta}{2} & 0 & 0 & 0 \\ 0 & \cot\frac{\phi}{2} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{4.17}$$

4.3 透视除法带来的深度变化*

从最终的透视投影矩阵中可以计算顶点深度的变化，假设输入的顶点为$(x, y, z, 1)$，经过变化与透视除法后得到新的深度为：

$$f(z) = \left[-\frac{(f+n)z}{f-n} - \frac{2fn}{f-n} \right] \frac{1}{-z} \tag{4.18}$$

不难看出，投影后的深度不再具有线性变化，随着物体距离相机的增大，深度变化被压缩的越厉害，4.5右图显示顶点距离相机在$[10, 20]$的时候，投影后会被映射在$[-1, 0.15]$的范围，而距离在$[20, 100]$的所有位置只能映射到$[0.15, 1]$的范围上。这也正是越远距离物体越容易产生z-fighting的原因，可以尝试通过反向计算得到(4.18)。需要注意的是，在片元着色器中通过gl_FragCoord.z拿到的z值又是在$f(x)$的基础上做

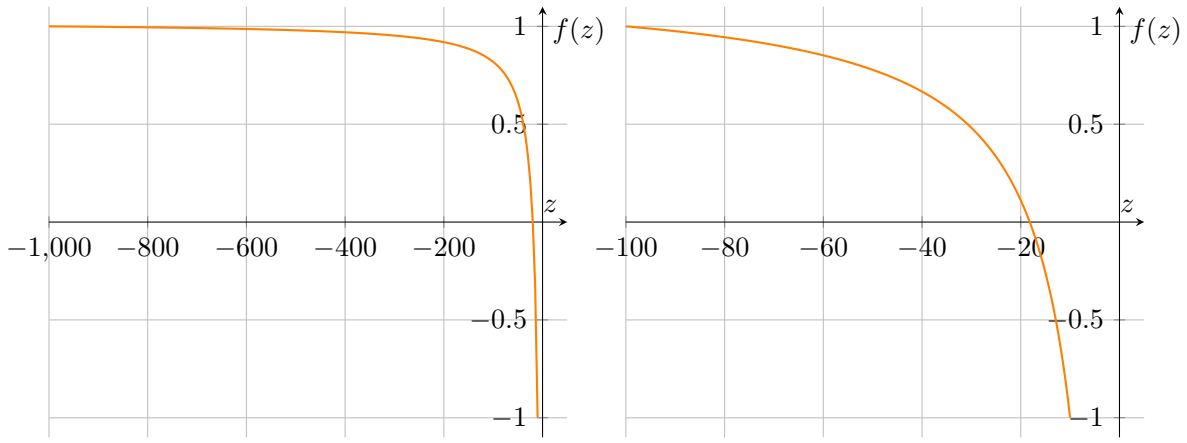

图 4.5: 不同的近远裁剪面对深度值的改变，水平轴是相机空间顶点z值，垂直轴是投影后的深度

了归一化的，需要先映射到NDC空间的$[-1,1]$，再通过公式(4.19)得到的是相机空间下的z坐标值，如果你只想知道深度，需要取绝对值，查看Web端可交互示例。

$$f(z)=\frac{2fn}{(2z-1)(f-n)-(f+n)} \tag{4.19}$$

4.3.1 深度缓存与写入

方程(4.19)一般是在片元着色器中进行的，在写入深度缓存之前，改掉标准管线计算的深度即可。片元中的计算精度可以比较高，或者与主机端没有差别，现代GPU处理float与double的能力甚至远超主机端，如此高的计算精度（比如double64）为什么z-fighting还比较常见呢？这就需要提及深度缓存了。在Learning to Love your Z-buffer这篇文章中提及现代GPU的深度缓存一般是24位，老一些的只有16或者8位。如果开启了模板（stencil）功能，它会与深度缓冲区共享，一般模板占用8位，这会进一步降低深度缓冲区的大小，从而加剧z-fighting。

片元的深度（gl_FragCoord.z）在片元着色阶段是归一化的值，这也是(4.19)需要乘2减1还原到NDC的原因。在管线的末端，也就是将其写入深度缓冲区的时候，一般是显卡驱动程序或者硬件决定写入的内容，这是种自定义的转化过程（由驱动程序或者硬件制造商决定），其存储的数据要么是线性的，要么是非线性的，最简单直接的整数线性写入就是将$(0,1)$范围的深度映射到$(0,2^{24})$的整数（24位深度，开启sencil功能后，只有16），然后保存；复杂些的可能会是定点小数，这一般是下游程序员无法接触与改变的。

4.4 NDC空间逆变换

在一些应用下，需要将已知的NDC空间坐标变换到相机空间或者世界空间，后者很直观乘以世界到相机的逆矩阵即可，我们重点推到下如何将NDC下的坐标转到相机空间。NDC空间是在透视除法之后的结果，坐标是齐次的，除法之后的空间是透视的非线性空间，不能直接乘以相机的逆矩阵，但是mvp变换是线性的，假设相机空间坐标乘以p矩阵后得到的是新坐标，其w分量的值是w，那么我们先从已知的NDC坐标转向裁剪空间坐标：

$$P^{-1}\times\begin{pmatrix}aw\\bw\\cw\\w\end{pmatrix}=\begin{pmatrix}(a.b.c)w\\(a.b.c)w\\(a.b.c)w\\(a.b.c)w\end{pmatrix}=\begin{pmatrix}x\\y\\z\\1\end{pmatrix} \tag{4.20}$$

逆矩阵与坐标相乘的结果是关于w的线性组合，想要与未知的$(x,y,z,1)$相等，顶点w分量就等于1，带入xyz即可，也就是先做透视逆变换，再做透视除法。

4.5 观察API

观察API就是用简单的方式指定照相机或者眼睛的位置与方位的函数，理论上只要能实现具体指定眼标架即可，这部分内容有些老旧。书中特意说到了两个常用API：

- VRPvrp、VPNvpn、VUPvup；
- LookAt()；

4.6 剔除、多边形偏移、阴影

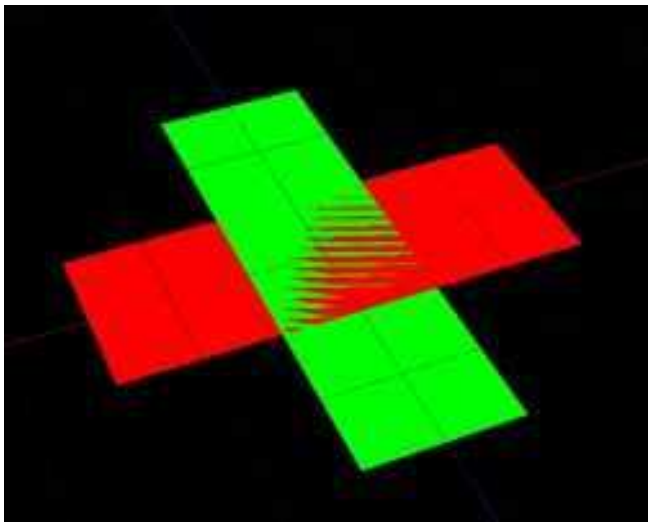

图 4.6: z-fighting可能出现的效果，图中两个平面在同一水平位置。图片来自网络

三角形都有2个面，这就需要指定哪个是正面，哪个是反面，OpenGL默认使用逆时针方向为正：右手四指沿着给出三角形三个顶点的顺序握住，拇指向上所指便是正向。我们可以利用glFrontFace函数来指定想要的顺序。OpenGL这么做的原因在于想给应用更多的灵活性。指定了三角面正向后，需要启用剔除功能glEnable(GL_CULL_FACE)，再使用glCullFace指定要剔除的方向会有所改善。如果你既想给三角形涂纯色，又想给它描边（纯色与描边色不同），这就需要用到多边形偏移，否则会出现z冲突效果。使用的话依然需要先开启多边形偏移：glEnable(GL_POLYGON_OFFSET_X)，然后使用glPolygonOffset函数指定必要的参数，不过不要太指望这项技术，有必要的话详细学习相关参数及其参数；关于阴影，可查阅shadow map和shadow volume等相关技术。

一些图形系统利用在定义多边形时指定的前三个顶点自动计算出法向量。OpenGL没有这样做，把计算法向量的任务留给应用程序可以为应用光照模型提供更多的灵活性。 《计算机图形学》中文第三版。

4.7 总结

第四章利用空间变化为基础，逐步说明了如何将空间中指定大小、方向的长方体、平截椎体（视锥）变换到CVV的计算过程。CVV是整个绘制流水线中连接顶点处理与光栅化、片元处理的中间节点，是对顶点位置的归一化呈现，便于后期处理，普通应用没有更多的用处，也没有什么好纠结为什么要转换到CVV下。章节中还介绍了几个观察API，书中安排的较前边一些，我第一次读感觉很糊涂，不能明确联系它与后面推导的关系，如果有前辈指导自然方便很多。后边还介绍了剔除面、多边形偏移与阴影，这些都对第四章核心没有多少帮助，但在实际使用中非常必要。

我读这一章总觉得应该抓住第三章的关键词“空间变换”去帮助理解。初学者可能很容易想到空间变换是诸如平移、缩放等等的变换，的确但同时比较狭隘。第四章的各种投影无非就是空间之间的变换，比如平行投影与透视投影都是将用户给定的视见体变换到一个被称作规范视见体CVV的空间里面，目的就是为了在后期好进行裁剪、设备坐标系好映射。我读第四章始终觉得Edward没有强调这个概念，以至于读者难以从第三章平滑过渡到第四章。还有一点我觉得不太好的地方在于Edward在讲解透视规范化矩阵的时候，有点先结果后原因的意思，这也不利于读者对其进行最初的认知。如果能从那些图形学前辈如何慢慢形成现在的图形学科学的角度去讲，肯定是最好，因为这肯定符合认知过程。

Chapter 5

光照基本组成

有了光，我们的物体才有了影，光影不分家。“光”有强弱的不同，物体表面不同的光强表达出了物体的形。“影”很少说到强弱（有些影比较淡是因为有环境光或者物体表面的反射光参与了进来），但影的存在表达出了物体的依托，也就是位置关系。在渲染中经常使用的光有三种：环境光（ambient）、漫射光（diffuse）和镜面光（specular）。

环境光：这种光在真实世界中其实是不存在的，但在CG中必须有的原因在于我们非常有必要给物体整体渲染出最基本的亮度与轮廓，不然在监视器上就是漆黑一片（如果光照不到），想象一个全封闭的房间，墙是纯黑的，里面有一盏灯，于是整个房间昏昏暗暗，但是如果把墙涂成白色，同样的孤灯下马上就觉得明亮许多，这个整体的亮度差，我们姑且就拿来形容是真实世界的“环境光”。

漫射光：漫射光的存在给世界提供了粗略的明暗效果，使得物体出现了基本的形状，想象一下阴天的户外吧，看不见太阳，但外面的东西依然可以辨认，注意观察的话你会发现，这种情况下基本或者根本看不见影子，较远处有2棵树，不熟悉环境的话便会很难判断出孰近孰远（抛开树干粗细、树高等）。想象到这里你应该能对漫射光有了正确的感性认知。

镜面光：就像激光一样，“直来直去”。照不到的地方肯定形成影子，镜面光的存在让物体有了鲜亮的明暗效果，有了质感与细节。

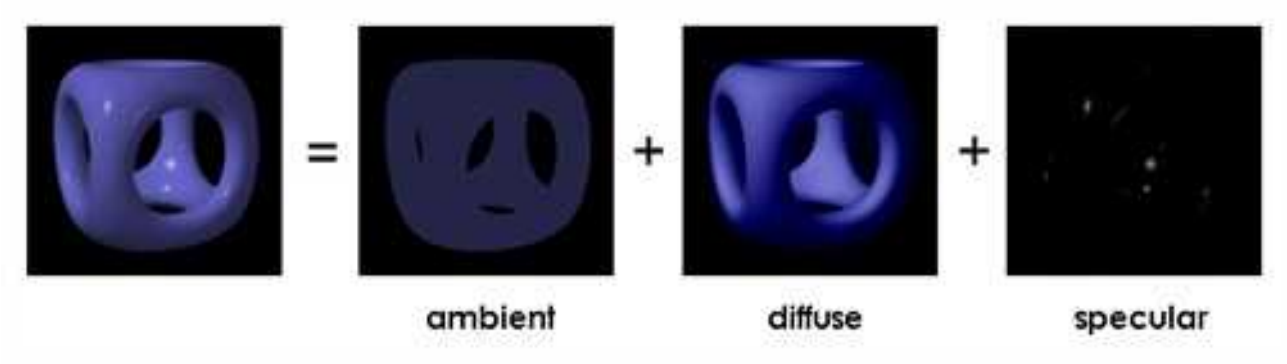

图 5.1: 三种不同光线对物体的贡献，图片来自维基百科

5.1 Phong模型

我们来简单介绍下一个被称为**Phong**（冯式）的光照模型，三种光的强度分别用L加其英语单词首字母做下标表示。

5.1.1 漫射光贡献

picture here! mimg Attach:image-cg-lighting.jpg 光束照射到平面上发生漫射现象，$\hat{u}$为光线反方向的向量，$\hat{n}$为平面法向量

一束光照射在表面上，图中的$-\hat{u}$方向，假如该物体是个石灰平面（非常粗糙），那么这束光将会被均匀反射到四面八方（实际情况是虽然反射到了四面八方，但是并不均匀，可能某些方向比较集中些）。我们将能均匀反射到四面八方的表面叫做“理想漫反射表面”（所谓理想的就是不存在的，CG中部分渲染是将表面看作理想的漫反射表面），以便简化问题，简化模型。根据Lambert定理可知只有光的垂直分量才对物体表面的亮度有贡献，也就是图中OC向量，OC的计算可以用点积进行。所以漫射光对最终光强度的贡献为：

$$L_d^{'} = L_d \mathbf{u} \cdot \mathbf{n}$$

我们分析下w，它是光从表面的‘背面’射向了O点，如果表面透明，完全需要考虑，不过我们这里只考虑不透明的情况，这种情况下w光对表面没有任何贡献，观察到对表面不做贡献的所有光线在图中的方向都是与n反向的，于是我们需要调整上面的模型如下：

$$L_d^{'} = L_d \max(\mathbf{u} \cdot \mathbf{n}, 0)$$

公式到此依然需要解决2个问题：

- 现在的漫射光模型应该得到的是光线$\hat{u}$对表面O的所有光强贡献，但理想的漫射表面是将该贡献强度射向了四面八方，怎么着也不可能是全部的强度射向一个方向。解决这个问题我们只需要给模型加上一个大于0，小于1的参数k_d，表示所有贡献的百分之多少射向了周围（均匀反射），于是合乎情理；

- 第二个问题就是光的能量会随着传播距离而衰减（反比于距离的平方），因此需要加上距离衰减因子,最基本的想法就是将漫射强度除以表面到光源的距离d的平方，这样距离越远，最终的强度就越小。但在CG大部分实际应用中是将衰减因子写成一个一元二次代数式的形式（a、b、c为常数）：$F = \frac{1}{a+bd+cd^2}$ 这样写的原因在于不至于将距离光源近的物体照的太亮，而远的太暗。究其根本原因是在于我们真实的世界是各种光源、光线（反射光、折射光、漫射光、散射光等等）复杂叠加的结果，而此处仅仅将光源想象成一个点，将复杂叠加尽量简单化。

我们在Phong模型中不考虑光从物体照射进眼睛这段距离的衰减。 《交互式计算机图形学》
用点发光体照明一个场景是真实光照效果的一个简单逼近。 《计算机图形学》第三版中文。

$$L_d^{'} = \frac{k_d L_d}{a + bd + cd^2} \max(\mathbf{u} \cdot \mathbf{n}, 0)$$

5.1.2 镜面光贡献

picture here

镜面光在推理上与漫射光一样，只是加了额外的“高光系数”α（或者叫做“角强度衰减系数”，参考《计算机图形学》），最终公式见下：

$$L_s^{'} = \frac{k_s L_s}{a + bd + cd^2} \max((\mathbf{r} \cdot \mathbf{v})^{\alpha}, 0) \tag{5.1}$$

公式(5.1)中向量$\mathbf{r}$与$\mathbf{v}$分别表示入射光线$\mathbf{u}$针对平面法线$\mathbf{n}$的反射光线与着色位置到观察者的方向向量。由于$\mathbf{r}$与$\mathbf{v}$都是归一化向量，所以$\mathbf{r} \cdot \mathbf{v} = \cos\theta$。$f(\theta) = (\cos\theta)^{\alpha}$这个函数在不同参数下的曲线图：

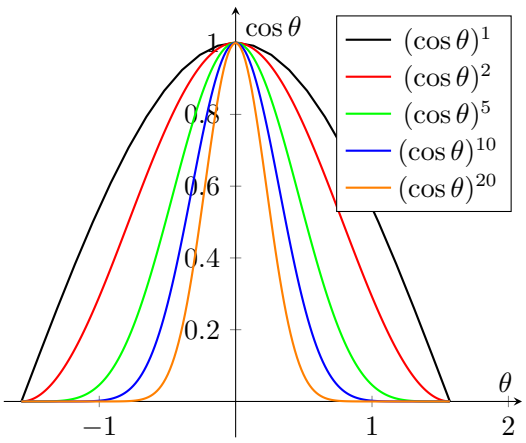

图 5.2: 函数$(\cos\theta)^\alpha$在不同α下的图像，当α越大时，图像表现得更加紧密，表示只有在θ角度越小时镜面光才能表现出来。

在图像5.2中，当$\alpha = 1$时图像就是标准的余弦函数，只要θ不超过90度，或多或少会有反射光射向$\mathbf{v}$方向，而曲线又比较平滑，渲染的结果是明显的渐变过度。当$\alpha = 20$时，预估在$\theta = 0.6$时，函数值已经很小，也就是说只有相机与反射光夹角小于34.38度时，才有可能看到镜面光的效果。

picture here

5.1.3 环境光贡献

由于环境光随处都在，就是为了整体提高物体的亮度，所以它对最终亮度有全部的贡献，为了统一起见，我们也加上一个系数k_a。整合起来，最基本的Phong光照模型就是这三种光的线性组合，公式见下。因为公式中使用了不符合物理规律的地方（比如不正确的距离衰减系数、将光源模拟成一个点等），所以我们将其称作**经验公式**。

$$L = \frac{k_d L_d}{a + bd + cd^2}\left[\max(\mathbf{u}\cdot\mathbf{n}, 0) + \max((\mathbf{r}\cdot\mathbf{v})^\alpha, 0)\right] + k_a L_a$$

跟着公式与上面的图像，我们再次进行说明。第一张子图是只有环境光效果，由于处处存在环境光且强度相同，因此没有明暗的变化，仅仅给物体涂上了一层基础亮度与基本的轮廓，让观察者知道那里有个东西。从图中我们明显看不出其表面的形状，直观感觉像是2个哑铃。第二章子图是仅仅有漫射光的情况，我们看到了物体表面的形，但是阴暗的地方与背景融为一体，因为我们没有给物体涂上基本的亮度。第三张子图是只有镜面光的效果，由于α设置可能比较高，导致我们只能视觉感知到反射光线明显朝向观察者方向的光亮，其他方向一律剔除，这才形成了图中星星斑点。第四张子图就是前三者的组合，他比第一张、第二张整体明亮，因为亮度叠加了。既能看到整体轮廓，又能看到正对表面的形状，还有高光表达物体表面“光滑”“明亮”的物体特征，这样就算完成了一次较好的渲染。反射向量的计算

picture here

向量$\mathbf{r}$是$\mathbf{u}$针对$\mathbf{n}$的反射向量，很明显$\mathbf{r}$是通过先给出$\mathbf{u}$与$\mathbf{n}$后才能计算出来的。从公式1.2中知道想要得到最终镜面光的贡献，必须事先计算$\mathbf{r}$，参见图4。我们将$\mathbf{r}$平移到以U为起始点的位置，从而得到一个等腰三角形OUU'，其中U'点肯定在向量$\mathbf{n}$所在的直线上。假设 $\mathbf{u} + \mathbf{r} = x\mathbf{n}$ 如果我们能得到$x\mathbf{n}$的具体数据就能通过减法得到$\mathbf{r}$。

$$\mathbf{r} = 2(\mathbf{u}\cdot\mathbf{n})\mathbf{n} - \mathbf{u}$$

5.2 改进了的Phong模型

5.2.1 Blinn-Phong模型

picture here

接下来我们继续查看公式1.2。现在我们知道了**r**的具体数据，那么计算镜面光不是问题。进一步的发展要么就是设计新的光照模型要么就是优化Phong模型。改进的模型就是从优化的角度进行了发展。前辈们通过一个叫**半角向量**的思想优化了这个模型。

图中所示，红色向量**h**就是**u**与**v**的所成角的半角向量，其方向是$\mathbf{u}+\mathbf{v}$，大小的话一般都使用归一化的向量。可以简单的证明n与h形成的角度是r与v形成角度的一半，如此一来我们可以近似的将**vr**的点积计算写成**nh**的形式。简单分析一下计算**r**时候的开销：7个乘法运算、5个加法运算。而计算h的开销：3个乘法、4个加法。除了性能上的提升以外，新向量h也会在其他方面有所贡献。通过将**rv**的运算改成近似的使用**nh**的形式，我们将此改进的Phong模型称为：**Blinn-Phong模型**。

5.3 着色

这里介绍三种基本的着色方案：

1. **flat着色**：就是一个三角面一种颜色，具体颜色值是要么来自第一个顶点、要么来自最后一个顶点颜色，通过下面OpenGL函数指定：
   (a) glProvokingVertex(GL_FIRST_VERTEX_CONVENTION)
   (b) glProvokingVertex(GL_LAST_VERTEX_CONVENTION);
2. **高氏着色**：对光照进行插值的着色方案；
3. **冯氏着色**：对法线进行差值的着色方案；

5.4 Bibliography

Chapter 6

从顶点到片元

在计算机图形学中，图形的生成过程以应用程序为起点，而以图像的生成为终点。
《交互式计算机图形学》中文第六版P.213

图形系统必须做的事情有两点，1）每个几何对象必须通过图形绘制系统；2）帧缓冲中要为显示的颜色赋值；目前主流的绘制策略有两条：

- **面向对象的（object-oriented)**：面向对象的渲染策略需要大量的内存以及单独处理每个对象所需的大量时间开销，随着高密度存储技术的发展，这个已经不是主要问题，其局限性是：无法处理大部分的全局计算，比如光的反射计算；
- **面向图像的（image-oriented)**：主要问题是需要事先使用优秀的数据结构为后期处理做数据基础，不然从像素出发的逆向绘制策略难以辨识哪些几何对象影响哪些像素，典型的面向图像的策略是光线追踪；

6.1 片元处理

片元处理需要至少从这两点去理解，第一需要注意经过光栅化后的几何图元，它可能自带有一定的插值或者平面值颜色（来自顶点颜色属性），同时颜色又需要来自纹理，这两者之间需要一定的逻辑柔和起来，通常在片元着色器中使用分量积的逻辑计算方式；第二需要明白整个片元在计算颜色完毕后写入帧缓存的方式，我们可能需要用到glBlendFunc与glBlendEquation，一个用来调整等式的因子，一个用来调整等式的方程，比如用加法还是减法还是其他。这里还说到隐藏面消除与背面剔除功能，这是两个概念，前者是说物体A被物体B全部遮挡，那么A就不要绘制；后者是说对于一个面是否要绘制正面与还是背面或者都绘制。

6.1.1 裁剪

对裁剪的理解，不能局限与流水线中对图元的处理（在CVV中就保留，不在或者部分在的话就裁剪）。应用程序中采用一定的逻辑在流水线开始之前就屏蔽需要绘制的部分对象也是一种裁剪，比如采用八叉树等数据结构首先屏蔽在摄像头后面的物体，或者采用某种算法轻松计算出完全处于不透明实体后面的物体，然后排除此次渲染。这无疑是提高程序整体性能的关键步骤。下面主要说下《交互式计算机图形学》中提到的两个2D（可扩展到3D）线段裁减算法：

- Cohen-Sutherland算法
- Liang-Barsky算法；（Liang是梁友栋）

- Sutherland-Hodgeman算法

Cohen-Sutherland Clipping

Cohen-Sutherland Clipping又叫**编码裁减算法**，是首个使用浮点减法和位操作相结合代替大量的高开销浮点乘法和除法的裁减算法。优点：

- 要处理的线段非常多，而实际显示在屏幕上的比较少时；
- 可扩展到三维空间；

算法思路：

算法将四条屏幕的边无限延长，这样整个屏幕所在的平面被划分成了9个区域，并且进行编码，如上图所示。编码并非随意经行，假设四位编码为$abcd$，a位表征的是所有y值大于$ymax$的情况（大于的话就是1，小于为0），b位是所有y值小于$ymin$的情况（小于就是1，大于为0），c位表示所有x值大于$xmax$的情况，d位表示所有x值小于$xmin$的情况。屏幕所在区域（中间部分），全部为0。当给定一条线段后，首先对端点经行编码，端点在哪个区域就用哪个4位的编码表示即可。再次假设两个端点A、B的编码分别为：**o1**，**o2**.现在**o1**与**o2**之间就会存在下面四种情况：

1. **o1=o2=0**：两个端点都在中间区域，肯定不需要裁剪；
2. **o1=0,o2!=0 or (o1!=0,o2=0)**：说明有一个端点是在中间区域，另一个在外面，需要裁剪1
3. **o1&o2!=0**：说明两个端点在屏幕的同一外侧，这是可以完全丢弃的线段；
4. **o1&o2=0**：说明两个端点不在同一外侧，有可能横（或者斜）穿中间区域，此时就需要进行裁剪2

那么我们：

1. 需要计算出线段与屏幕某条边缘的交点，之后使用在屏幕内的一部分；
2. 需要先计算出一个端点与屏幕某条边缘的交点，然后再利用新计算的交点与另一个端点计算出第二个交点，最后使用两个交点中间的部分（也就是留在屏幕中的线段）；

Liang-Barsky Clipping

由于在图形学中对于线段的表示一般是使用参数形式（因为这种表达方式有很好的鲁棒性），那么可否从其参数中得到有关线段裁剪的启发呢？图一是个简单的线段与裁减窗口的示意图，从线段的参数方程可以知道四个绿点的α值肯定介于0到1之间，同时F点的α值应该大于E点的α值（假设从$P0$到$P1$），而如果小于的话，就像图二那样，此时线段根本不可能在裁剪区域内。所以整个算法的核心就是在必要的时候计算α，然后比较$EFGH$四个绿点的α值的大小关系，从而确定是否裁剪与裁减范围。该算法完全可以在其他方面使用，比如碰撞检测中射线是否与物体碰撞等[1]。

这里有一篇关于该算法的很棒的文章，不过文章最后的完整代码中有一点可以优化的地方就是：for循环中计算r的值与下面的if判断可以互换，就像下面3,4行这样：

```
//others...
if (edge==3) { p = ydelta; q = (edgeTop-y0src); }
if(p==0 && q<0){
    // Don't draw line at all. (parallel line outside) r = q/p;
    return false;
}
if(p<0) //others ...
```

[1]可以参考《Real time collision detection》英文版179页章节5.3.3：“Intersecting Ray or Segment Against Box”

Sutherland-Hodgeman Clipping

上面的算法都是将线段数据传入，函数结束后得到的是最终的结果。整个函数体是将裁减区域看成一个4条直线组成的整体来对待，如果我们将他们拆开对待，变成4个子模块（这种情况下每个子模块也比较简单），每个模块处理一条具体的裁剪边。如此一来更适合流水线绘制结构，见下图；

6.1.2 光栅化

DDA算法（数字微分分析器算法）

该算法就是利用自变量变化，计算因变量的值，从而确定必要的像素位置。看下图中斜率小的直线，自变量（假设是x轴）每次增加1，计算出来对应的y值，因为像素的位置是整数，所以将y四舍五入取整，得到形如右边图6.29中可见的像“台阶”式的一组像素。但是当直线斜率大于1的时候（就像图中比较垂直的直线那样），每次计算出来的y值跨度太大，导致对应的像素不能连贯起来。解决的办法是当直线斜率大于1后，我们将直线方程的自变量、因变量互换（y值为自变量，每次增加1，计算x），这样就能解决。

Bresenham算法

Bresenham算法网上说是DDA的改进，不过从算法的整体思路来看两者确实有些相似。首先他们都事先需要判断斜率是否大于1，是的话就需要互换两个变量；其次因变量也是以1为单位步进计算。这里我打算从4个层面说下Bresenham算法：

1）约定：

监视器上有像素，像素是离散的，我们不会去说$(0.3, 0.3)$像素在哪里，只会说$(3, 1)$像素，但是我们在二维屏幕上说明一些问题的时候一般都是用卡尔坐标系，它是连续的，这并没有什么不妥。但是当我们不做人为约定的话，就会在理解上出现一时的紊乱，比如像下面两张图一样。

如果使用第一种坐标系的位置的话，那么区间$x, y \mid x \in [3, 4), y \in [1, 2)$是属于像素$(3, 1)$的，如果使用第二种，那么区间$x, y \mid x \in [2.5, 3.5), y \in [0.5, 1.5)$是属于像素$(3, 1)$的。下面所说算法约定使用第二种坐标系。

2）算法理论：

整个算法的核心就是利用好直线的斜率，很明显自变量每次增加1后，因变量肯定增加一个单位斜率的大小（或者一个单位斜率倒数的大小，如果变量互换的话），这样的好处就是不需要代入直线方程计算出全新的y值。然后我们利用历史上累计的y变化量与0.5做比较，根据比较的大小关系确定本次垂直像素的选取。举个例子：假如射线从原点出发，斜率0.3。当自变量（针对该例应该选x）增加1个单位后，y增加0.3，0.3小于0.5，表示该点还没有“逃脱”像素$(1, 0)$的范围，所以这点必须是像素$(1, 0)$，如图十所示。

紧接着x值继续增加1，y历史变化0.3，此时再加上斜率0.3为0.6，比较后发现超出了0.5，于是全新的点$(2, 0.6)$属于像素$(2, 1)$，如图十一所示。此时y的历史变化值0.6就需要进行一次修正，因为0.6是针对像素$(2, 0)$而言的，也就是说：以像素$(2, 0)$来说，因变量的0.6表示向上增长了0.6个单位，但对于像素$(2, 1)$而言，此时的因变量应该为$0.6 - 1 = -0.4$，表示图中点M的y值相对于像素$(2, 1)$的中心点在下方0.4个单位。如果继续增长想“逃脱”第二行像素$(x, 1)$的范围，至少还需要$0.5 + (0.4) = 0.9$个单位。

修正就是减去1，从而y的史变化值成了-0.4，后面依然还是循环（直到线段的另一个端点结束），从图十二中知道点Q的历史变化y为$0.2 + 0.3 = 0.5$，此前约定像素的范围得知点Q属于像素$(5, 2)$,修正后为-0.5。

3）算法优化：

从上面的说明中可以看出，整个算法循环的部分是不停的增加因变量斜率值大小的数值，然后比较，条件成立的话修正因变量。虽然是些性能不错的加减法，但都是浮点运算，因此我们尝试给统一乘以一个

具体的值，化浮点为整型。注意到直线斜率的计算是这样的：$k = \frac{\Delta y}{\Delta x}$，因此这个具体的值就是$\Delta x$，外加对0.5的修正（再乘以2），就是整个优化的过程了。

Chapter 7

采样与蒙特卡罗积分

采样就是采集样本，它是个过程，既然是个过程，就需要花时间。我不能很明确的解释在计算机图形图像领域采样的定义，但在信号处理领域，采样是将信号从连续时间域上的模拟信号转换到离散时间域上的离散信号的过程，这个解释确实可以采用，并且我觉得图形学中的采样就是来自信号处理领域。

- To avoid aliasing when sampling pixels
- Cameras with a finite-area lens are used to show depth of field. The lens must therefore be sampled.
- To represent area lights and soft shadows more accurately, we must sample the light surfaces.
- Global illumination and glossy reflection requires sampling of BRDFs and BTDFs.

7.1 低偏差采样（Low-Discrepancy Sampling）

想象这样一个场景：你去采集湖水样本，在湖边就装满了一瓶子带回实验室，得出了结果：湖水成碱性。请问可信度如何？当我告诉你在他来采集湖水之前一分钟有个小孩就在他采集的地方撒了泡尿你现在会是什么态度？所以单凭一瓶子湖水的测试结果不能构成整个湖水水质的真实属性，我们需要一些个样本。于是采集人员跑去在湖边采集了1万瓶样本，测试结果如何？依然不太可信（但至少比之前好多了）。因为他没有去湖中心（远离湖边）的地方采集样本，那里的水质可能更好。从假设的场景中我们可以看到样本的采集是需要一定的数量与合理的分布的。盲目的采样不但不能达到正确的目的而且浪费采集与测试时间。于是我们说采样的质量有好有坏，低偏差采样就是一类质量好的采样，注意这里说的是一类而不是一种或一个。那么我们如何判断一组采样数据就是质量好的采样呢？这就需要首先了解偏差的概念：

$$D_N(P) = sup\left|\frac{A(B)}{N} - \lambda(B)\right|$$

上面公式中N是采样数量，B是统计的一片区域（比如一段长度，一块面积或者体积），$A(B)$是这块区域中采样点的数量，$\lambda(B)$是这块区域在所有区域中的百分比，双竖线就是绝对值，sup表示集合中的上界，也就是最大值。整体而言就是将整个采样区域分成任意个子区域，每个子区域都计算差的绝对值，然后绝对值最大的那个数就是偏差值。顾名思义，低偏差就是偏差值比较低的偏差，至于多少算低，我读书少没的看到哪里有写，显然这个“低”不是绝对的（或者真是没有看到8-O）。

低偏差采样就是用具有低偏率的采样方式去采集样本，因为分布相对均匀，所以采集的样本会更客观。

7.1.1 Van der Corput Sequence

我们的日常生活中使用的全部是十进制的数字，逢十进位，在计算机中也会用到二进制、十六进制，偶尔用到八进制。其他进制计算机科学也不常使用，比如三进制，七进制等。他们的转换理论肯定是一致的，也好理解。我觉得唯一需要注意的地方是如何表示，比如数字7是哪个进制的？一般情况下我们会在数字前面加上0（数字0）以表示它是个八进制数字7，前面加上0x（数字0与字母x）以表示它是个十六进制的数字7，后面加上下标2表示为二进制数字，什么都不加（默认情况）表示为最常用的十进制。进制之间的转换是计算机科学的基础知识，这里不做赘述。总结一下就是这样（依然是数字7）：

- 二进制：111_2
- 八进制：07
- 十进制：7
- 十六进制：$0x7$或者$0X7$

倒根（Radical Inverse）

思考下这样一个问题：如何将1到N均匀的映射到$[0,1]$之间？

我们肯定不能用A除以自身，因为那样所有正整数全部映射为数字1，而且重复N次。你可能想到了用1除以$A \in [1,N]$，这样一来虽然彼此不重复，但会趋向0。

前辈们提出了一种方式，就是在小数点的位置放置一面镜子，从镜子里读取的数字就是想要的结果。比如1（小数点在1的后面），小数点位置就是一面镜子，现在从镜子里面看数字1它依然是1，但小数点跑到了1的前面。于是1的镜像就是0.1；再看一个数字8550014，我们不赘述，镜像后是0.4100558；其他进制的数字是一样的，比如二进制数字110100111_2，镜像后是0.111001011_2。

7.1.2 Halton Sequence

7.1.3 Hammersley Sequence

7.2 蒙特卡洛积分（Monte Carlo Integration）

蒙特卡洛积分（Monte Carlo Integration）是一种使用“随机采样”来近似计算积分值的方法，特别适合于高维、解析难以求解的积分问题。可以想象向边长为1的正方形面板上投掷飞镖，当投掷的足够多后，数一数其上半径为0.5的圆内飞镖数与总飞镖数的比例，大致就能知道圆的面积。这个过程可以用程序模拟：

```
import random
import math
TOTAL = 100000;
count = 0;
for i in range(TOTAL):
        x = random.random() - .5;
        y = random.random() - .5;
        if math.sqrt(x*x + y*y) < .5:
                count += 1;
print('points within circle:(',count,'/',TOTAL, '), circle area:',count/TOTAL);
# points within circle:( 78568 / 100000 ), circle area: 0.78568
# PI*0.5*0.5 = 0.7853981633974483096156608458198
```

在估计积分结果的时候，可以考虑将积分区间平均的分成N份，计算每一份的函数值，然后N份面积相加：

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{N} f(x_i)\frac{b-a}{N} = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{\frac{1}{b-a}} \tag{7.1}$$

蒙特卡洛积分与该形式类似，是给被积函数预先除以概率密度函数（PDF）[1]，然后计算整体的数学期望，由于连续函数的数学期望是乘以PDF，两者相消，最终的平均数其实就是原来函数的积分结果。

$$\int_a^b f(x)\,dx \tag{7.2}$$

$$= \int_a^b \frac{f(x)}{p(x)}p(x)\,dx \tag{7.3}$$

$$= E\left[\frac{f(x)}{p(x)}\right] \tag{7.4}$$

$$\approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)} \tag{7.5}$$

重要性采样（Importance Sampling）
多重重要性采样（MIS）
马尔可夫链蒙特卡洛（MCMC）
自适应采样、自举方法、控制变量法
准蒙特卡洛（Quasi-Monte Carlo），用低差异序列提高收敛

7.3 重要性采样

最基本的采样方法就是随机采样，使用伪随机算法产生采样位置，然后执行采样，这种无差别采样的方法简单直观但是缺乏准确性，虽然可以通过增加采样数量而增加精度，但有的时候我们希望在一些地方能有更多采样点，而其他地方尽可能稀疏以便提高收敛效率，这样一来单纯的增加数量无法达到目的。重要性采样的核心观点是：被积函数值越大的区间对积分的贡献越大，越应该重要对待，或者说如果这段区间计算的越精准全面，那么整体区间的积分值也就越接近真实值。

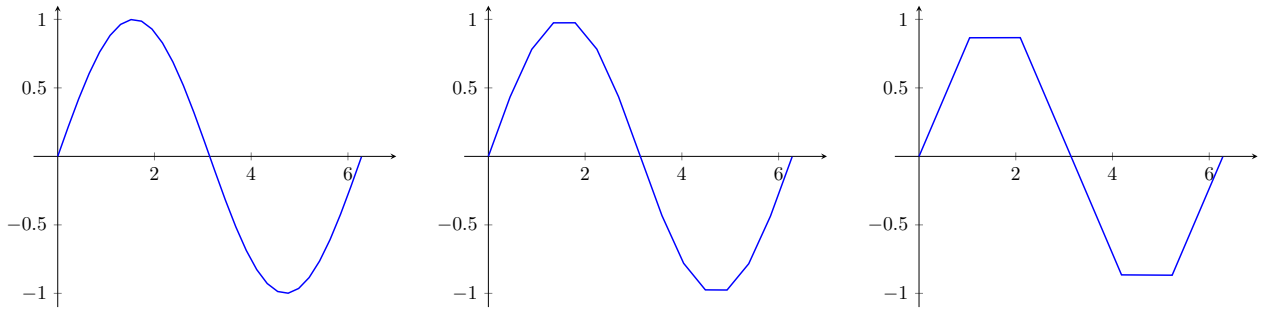

图 7.1: 随机采样后对正弦信号的还原（从左到右依次30，15，7个采样点），可以看到在信号拐点的地方失真最为严重。

这里需要注意并不是被积函数斜率越大越应该仔细面对，重要性采样是说函数值越大的地方越需要被全面考虑到，随机变量越应该采样这片区域，如果这片区域采样率低，那么更有可能采样不到这片区域，从而丢掉

[1] 概率密度函数后面有说明。

7.4 采样分布

7.4.1 概率密度函数（PDF）

对于离散型随机变量，比如投掷筛子，一共可能的结果只有6种，每一种可能出现的概率P为1/6，但连续型随机变量不能这么说，我们不能说某个具体数值的概率是多少$P(X = x)$，只能说某个区间的概率是多少$P(X \leqslant x)$。统计学中一般用概率密度函数$p(x)$[2]来描述随机变量在某个区间内的概率分布情况，然后在指定区间做积分，其值就是概率大小。

$$P(a \leqslant X \leqslant b) = \int_a^b p(x)\,dx$$

可以将概率密度函数想象成木头的密度，有些木头密度很大质量很重，有些很稀疏。即便相同的一块木头，可能一些地方密度相对较大，而另一处较小，我们不能把木头的密度直接等同于木头的质量，它还需要乘以体积，而体积就相当于积分的区间，故此即便体积再小也能算出质量，但无论如何也不能说“这块木头在这一点处有多重？”，因为体积为0，可以说质量为0，或者没有质量。
并不是任意函数就能作为PDF使用，它应该具备两个基本的要求：

1. 非负性：$p(x) \geqslant 0$
2. 定义域内积分为1：$\int p(x)\,dx = 1$

7.4.2 常见的连续型随机变量分布函数

均匀分布（Uniform Distribution）

设随机变量$X \sim U(a, b)$，即X在区间$[a, b]$上服从均匀分布，其PDF为：

$$p(x) = \begin{cases} \frac{1}{b-a} & a \leqslant x \leqslant b \\ 0 & \text{otherwise} \end{cases} \tag{7.6}$$

正态分布（Normal Distribution）

正态分布是描述随机变量集中在平均值附近，越远离平均值概率越小的一种连续型分布，记作：$X \sim \mathcal{N}(\mu, \sigma^2)$，其中$\mu$为均值，$\sigma$为标准差。关于正态分布，有一个68, 95, 99.7法则，也就是中心轴左右两侧1个标准差，2个标准差，3个标准差范围的钟形面积占整体面积的百分比，面积就是积分的结果，故此它们也是范围内的概率大小。

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{7.7}$$

指数分布

如果一个随机变量 X 表示某件事发生前的等待时间，比如：下一个顾客到来的时间；灯泡使用多久会坏；火车多久后进站。这种“等待某事件发生的时间”就经常服从指数分布。记作：$X \sim \text{Exp}(\lambda)$

$$p(x) = \begin{cases} \lambda e^{-\lambda x}, & x \geqslant 0 \\ 0, & x < 0 \end{cases} \tag{7.8}$$

[2]以下数学符号中，概率一律用大写的P，而概率密度用小写的p

7.4.3 分布之间的转换

如果我们知道一种采样分布$p_1(y)$，当y用另一个函数$f(x)$表达的时候，新的分布$p_2(x)$与原来$p_1(y)$分布是什么关系？我们知道PDF在定义域内的积分是1，如此便有：

$$\int p_1(y)\,dy = \int p_1(y)y'_x\,dx = \int p_2(x)\,dx = 1 \tag{7.9}$$

从CDF的积分可以直接得出pdf之间的关系：

$$p_1(y)\frac{dy}{dx} = p_2(x) \tag{7.10}$$

可以看到y'_x其实就是两个PDF之间的分布关系，对于二维或者多为的情况，可以使用雅可比（Jacobian）行列式，

$$p_1(x,y)|Jacobian| = p_2(z,w) \tag{7.11}$$

极坐标与笛卡尔坐标

$$p(x,y)r = p(r,\theta) \tag{7.12}$$

3D极坐标与笛卡尔坐标

$$p(x,y,z)r^2\sin\theta = p(r,\theta,\phi) \tag{7.13}$$

下面是在三角形内均匀采样的示例，参考《physically based rendering 3rd edition》p.771

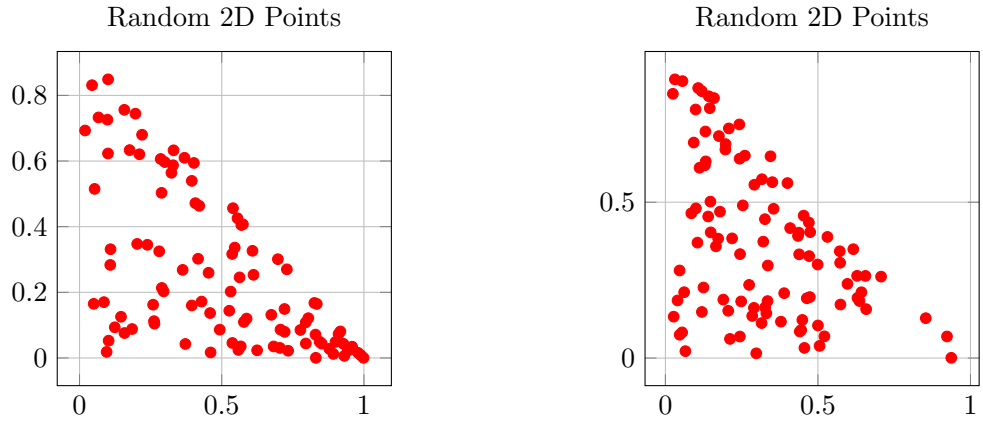

图 7.2: 平移与缩放

Chapter 8

基于物理的渲染（PBR）

8.1 HDR与纹理提交

HDR图像的存储格式可以是RGBA32F，也可以是RGBA16F（每个通道32位），这是容易理解的，不过为了数据体量更小，也有RGB9_E5（一共32位）这种格式，它是将RGB的缩放系数保存在了alpha通道，在解析的时候RGB乘以$2^{alpha-128}$就是最终的数据。我们可以简单的使用RGBA8这种内部格式，然后在shader中自己做上述计算，也可以在主机端将RGBAxF转成UINT格式，这样就可以直接使用RGB9_E5这种内部纹理格式了，除了需要预处理纹理麻烦外，在shader中可以直接通过texture()函数获取最终的数据，算是有得有失。

```
gl.texImage2D(
        gl.TEXTURE_2D, 0,
        gl.RGB9_E5, width, height, 0,
        gl.RGB, gl.UNSIGNED_INT_5_9_9_9_REV, null
);
```

8.2 BRDF

$$\int_H f(l,v)L_i(l)\cos\theta_l\,dl \tag{8.1}$$

8.3 微表面模型

8.3.1 GGX

几何遮蔽项

$$k = \frac{(roughness+1)^2}{8} \tag{8.2}$$

$$G_1(h,v) = \frac{h\cdot v}{(h\cdot v)(1-k)+k} \tag{8.3}$$

$$G_{ggx}(h,v,l) = G_1(h,v)G_1(h,l) \tag{8.4}$$

法线分布项

$$D_{ggx}(h) = \frac{roughtness^2}{\pi[(n \cdot h)^2(roughness^2 - 1) + 1]^2}$$

菲涅尔反射系数

$$F(h, v) = F_0 + (1 - F_0)(1 - h \cdot v)^5 \tag{8.5}$$
$$= F_0(1 - (1 - h \cdot v)^5) + (1 - h \cdot v)^5 \tag{8.6}$$

8.4 间接光照

PBR光照模型(8.1)中有BRDF项与辐射度L_i项目，如果我们独立采样L_i与BRDF项，那么模型的数学期望可以分解为单项的期望乘积[1]。从近似的结果得知PBR光照模型可以用两份数据（分别采样计算）相乘得到。

$$\int_H f(l, v)L_i(l)\cos\theta_l \, dl \tag{8.7}$$
$$\approx \frac{1}{N}\sum_{i=1}^{N}\frac{L_i f(l, v)\cos\theta_l}{p(..)} \tag{8.8}$$
$$= \left(\frac{1}{N}\sum_{i=1}^{N} L_i\right)\left(\frac{1}{N}\sum_{i=1}^{N}\frac{f(l, v)\cos\theta_l}{p(..)}\right) \tag{8.9}$$

8.4.1 预过滤环境贴图(prefiltered environment map)

(8.9)前一项是环境贴图的期望，用于镜面反射，从表达式知道完全可以用原始的环境贴图，如此一来就是清晰的图像，如果物体表面较为粗糙，就需要预先模糊这张图，这就是“预过滤”的意思。物体会有不同的粗糙度，这就需要计算出不同程度模糊的环境贴图，可以将粗糙度作为参数之一，用于约束反射光的方向，进而得到想要的结果。Unreal Engine 4在该过程中使用权重去计算，这也符合“重要性采样”的思想，具体而言就是光线与法线夹角作为权重值，因为物体表面对贴地角小的辐射能量会有余弦衰减项。

```
for( uint i = 0; i < NumSamples; i++ ) {
        // ...
        float NoL = saturate( dot( N, L ) );
        if( NoL > 0 ) {
                PrefilteredColor += texture( sampler, L).rgb * NoL;
                TotalWeight += NoL;
        }
        return PrefilteredColor / TotalWeight;
}
```

[1] $E[f(x)g(x)] = E[f(x)]E[g(x)]$，$f$与$g$独立采样为前提。

图 8.1: 预过滤环境贴图，左图为原始环境贴图，中图粗糙度0.3，右图粗糙度0.6

8.4.2 LUT查询表

(8.9)后一项是BRDF与余弦函数的积分，它是蒙特克罗方法表示，如果我们写成积分形式，并作简单变换：

$$\int_H f(l,v)\cos\theta_l\,dl \tag{8.10}$$

$$=(F_0(1-(1-h\cdot v)^5)+(1-h\cdot v)^5)\int_H \frac{f(l,v)}{F(h,v)}\cos\theta_l\,dl \tag{8.11}$$

$$=F_0\int_H \frac{f(l,v)}{F(h,v)}(1-(1-h\cdot v)^5)\cos\theta_l\,dl+\int_H \frac{f(l,v)}{F(h,v)}(1-h\cdot v)^5\cos\theta_l\,dl \tag{8.12}$$

加号左边的积分项结果在0-1之间，而右边的也是在这个区间，是否可以将他们的结果保存在2D纹理中，通过uv直接查询他们的节分结果呢？将公式(8.12)分开来看：

$$F_0\int_H \frac{f(l,v)}{F(h,v)}(1-(1-v\cdot h)^5)\cos\theta_l\,dl \tag{8.13}$$

$$\approx\frac{1}{N}\sum_{i=1}^{N}\frac{f(l,v)}{F(h,v)p(..)}(1-(1-v\cdot h)^5)\cos\theta_l \tag{8.14}$$

在Unreal Engine 4[2]中，概率密度函数使用了$(n\cdot h)/(4(v\cdot h))$，BRDF函数使用了Cook-Torrance，其FGD三项使用GGX函数版本（见上xxxxx），这样化简下来该积分项变成了：

$$\frac{1}{N}\sum_{i=1}^{N}\frac{G(h,l,v)(v\cdot h)}{(n\cdot h)(n\cdot v)}(1-(1-v\cdot h)^5) \tag{8.15}$$

简单说明下：菲尼尔项直接约掉，$\cos\theta_l$结果等于nl，它也会与分母项中的nl约掉；

[2]https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf

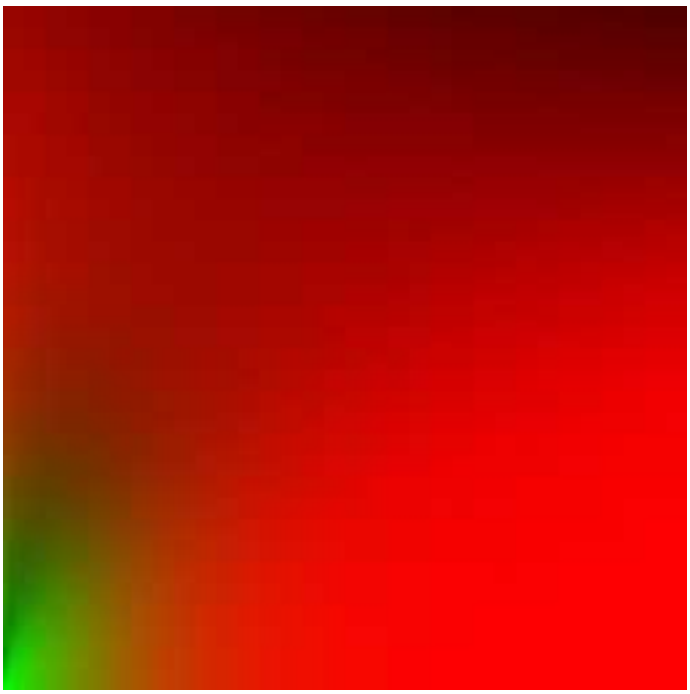

图 8.2: BRDF查询表，水平方向是法向与视线的余弦值，垂直方向是粗糙度。

同样，加号右边的积分项也是一样的化简过程，这样一来，我们就能生成一张贴图，用其直接查询积分结果，在实时渲染中两者相加即可，由于这张贴图是用来查看BRDF积分结果的，我们称其为BRDF查询表（BRDF Lut）[3]

$$\int_H f(l,v)\cos\theta_l\, dl \tag{8.16}$$

$$\approx F_0 LUT.x + LUT.y \tag{8.17}$$

这样就能快速的得到IBL光照下的PBR渲染光照模型：

$$\int_H L_i(l) f(l,v)\cos\theta_l\, dl \tag{8.18}$$

$$\approx (\text{PrefilteredColor})(F_0 LUT.x + LUT.y) \tag{8.19}$$

8.4.3 IBL漫反射

以上的计算结果是基于IBL的镜面反射，而对于漫反射而言，我们一般使用Lambert经典漫反射模型，也就是完全漫射，无论光线从哪个方向入射过来，全部无差别的、均匀的反射向四面八方，对此也可以进行预计算。

$$\int_{H^+} \frac{1}{\pi} L_i \cos\theta\, d\omega_i \tag{8.20}$$

$$= \frac{1}{\pi} \int_{H^+} L_i \cos\theta \sin\theta\, d\theta d\phi \tag{8.21}$$

$$\approx \frac{\pi}{N} \sum_{n=1}^{N} L_i \cos\theta \sin\theta \tag{8.22}$$

由于θ与ϕ独立采样，其蒙特卡洛表达式可以像(8.9)一样独立计算θ与ϕ项然后相乘。上式中使用了常数项PDF，根据蒙特卡罗公式在半球面的积分可以算出PDF等于$\frac{1}{\pi^2}$，这样与LambertBRDF常数项相乘化简后就只剩下π了(8.22)。这是完整的光照模型计算，因此在最后的使用中不必除以π。

这里贴上图片HHHHHHHHHHHHHHH。

[3]查询表可以用好多用途，出于不同的目的，比如AO查询表，辐照度查询表等，不要混淆。

Chapter 9

渲染技术

9.1 后期处理

“渲染后期处理”通常是指在完成基础渲染之后，对图像或视频进行进一步的视觉优化和效果增强处理。这一环节常见于 3D动画、电影、游戏开发、建筑可视化等领域。它的主要目的是让最终图像更具真实感、艺术感或视觉冲击力。

下面是后期处理常用的内容分类：

9.1.1 色彩调整

9.1.2 光效增强

9.1.3 景深与模糊

9.1.4 环境效果

9.1.5 相机模拟

9.1.6 合成与遮罩

9.1.7 图像降色与压缩

这是我学习《计算机图形学》第三版，第15章图形文件格式时的笔记

全彩色的光栅文件非常大，大多数文件格式采用某种压缩方法来减少存储和传输的文件尺寸。使用RGB颜色并且未压缩的光栅化文件称为原始数据(raw data) 向量文件格式(vector file format),通过一系列坐标位置和描述性直线段、填充区域、圆弧、样条曲线和其他图形元素的信息来存储图画。既支持几何表示、又支持光栅表示的文件格式称为混合格式(hybrid format)或元文件(metafile)；先说降色，有时降色方案也称为**量化**，量化就是从一个连续的分布中得到离散数据的过程，常见的技术实现有：

- 均匀降色：整体除以2，让颜色从8位255级降为7位127级；也可以不同的分量降不同的位数，比如红色分量降3位，蓝色降1位等。用一块区域的颜色的中值或平均值代替整个这块区域的颜色；
- 频率降色：统计一共出现的颜色的频率，然后按照需要保留使用频率最高的颜色；
- 中值切割降色： 1）分别统计RGB三个分量颜色出现的范围，形成区间颜色立方体； 2）选择差距最大的分量经行中值计算，将得出的中值作为分割线将区间颜色立方体分割成更多的子立方体； 3）选择所有子立方体中最大的子块作为下一步要处理的立方体，然后执行2； 4）直到用户想要的分割精度为止； 5）将所有在同一个子块中的像素填充为该块的中值颜色；

压缩分为有损压缩（lossy compression）与无损压缩（lossless compression），常见压缩技术有：

- 行程编码：将一行中的颜色按照：重复次数+颜色值来表示，比如: 20，20，20，34，44，78，30，30，30，30，30，30；
  经过行程编码后是：
  03，20，01，34，01，44，01，78，06，30；
  可以看到不重复的也需要加上数量（数量1），这样压缩比在颜色比较艳丽的图像中不但不高反而会小于1；改进了的行程编码是将不重复的颜色数量用负数表示，同样上面原始数据压缩后为：
  03，20，-03，34，44，78，06，30；
  意思就是34，44，78这三个颜色没有重复；

- LZW编码（Lempel、Ziv、Welch）： LZW编码是一种基于字典的编码，或者说是替代编码，就是将原始数据中一段重复较高的数据用另一个简单的字符/字符串代替，从高降低文件大小。这就需要维护一个表用来映射简短字符/字符串与原始数据。

- 霍夫曼编码（Huffman）：霍夫曼编码是种变长编码，它的核心思想是为原始数据中出现频率最高的赋予最简短的编码，它的特点是任何一个编码都不会是其他编码的前缀，比如如果0是一个编码的话，就不可能出现01，001，011等编码，关于霍夫曼编码的介绍特别多，我知道的有一期《程序员》杂志上就用了上下两篇文章介绍了霍夫曼编码，内容涉及霍夫曼实现该编码的简单历史，读来有趣且实用。

- 算数编码（Arithmetic coding）：算数编码相较霍夫曼编码比较晚才出现，可能是为了解决诸如霍夫曼编码压缩比理论上不足的问题。算数编码利用原始数据中各个基础码出现的频率而计算的，其最后的结果是个浮点数，去掉小数点与之前的0，就是一组有限的01序列。当初我有个小疑惑就是，这玩意怎么解码，不知道频率，不知道字符的。后来突然反应过来；算数编码仅仅是个信息的编码思想而已，不是加密算法。所以在提供最后的01序列的同时，还要提供有多少个基础码，分别什么频率等等。解码的时候根据给出的频率像编码一样，计算一个区间，查看给出的序列是否在这个区间，在的话就继续细分计算、比较，最后得到给出的原始信息。

- 离散余弦变换：这个压缩算法没有细看，仅作了了解。

9.2 延迟渲染