计算机图形学

——软光栅渲染器

指导老师　　贾金原

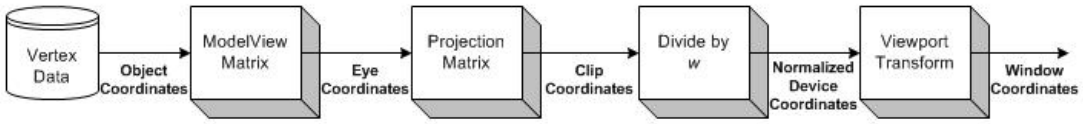
1652690　　苏昭帆

1551534　　孙允鑫

目录

一、顶点变换、裁剪以及画线

1、顶点变换过程



2、渲染过程

3、项目中定义的一些结构体

a、矩阵、向量结构体

b、坐标变换结构体

c、几何结构体

rhw，为某个顶点（x，y，z，w）的齐次点的 w 的倒数。在戈劳德着色模式下，相邻顶点的颜色差值将偏向 RHW 值高的一边，也就是说 RHW 高的一边在图元渲染过程中占有较多的比重。如果某个点的 RHW 值为 0，则使用另一个顶点的颜色值。用戈劳德着色模式渲染一个多边形的时候，先用顶点法向量和灯光参数计算每个顶点的颜色。然后，在该多边形的表面上进行线性插值，进而得到每个像素的颜色。比如，顶点 1 的颜色的红色值为 0.8，顶点 2 的颜色的红色值为 0.4，使用戈劳德着色模式和 RGB 颜色模式，这两个顶点连线上中点的颜色的红色值为 0.6。戈劳德着色模式能使图形的表面看上去像曲面一样光滑。

4、坐标变换

a、一些基本知识

3D 物体从三维坐标映射到 2D 屏幕上，需要经过以下一系列的坐标系的变换：

1）model：局部坐标系（物体本身的坐标系），是一个相对的坐标系，主要进行模型空间的绘制。

2）world：世界坐标系，物体放在世界里的坐标。系统的绝对坐标系，在没有建立用户坐标系之前所有点的坐标都是以该坐标系的原点来确定的。

3）camera：相机坐标系，相机也是世界里的一个物体，相机坐标就是以相机位置为坐标原点，相机的朝向为 Z 轴方向的坐标系。因为我们在电脑里看到的物体其实都是“相机”帮助我们看的，“相机”就是我们的眼睛，所以要以相机为标准进行坐标转换。

4）perspective：透视坐标系，三维坐标向二维平面进行映射。（x，y）的范围在[-1,1]，z 的范围在[0,1]。

5）screen：屏幕坐标系，原点在屏幕的左上角，x 轴朝右，y 轴朝下。x 的范围在[0,xres-1]，y 的范围在[0,yres-1]。

物体的位移、缩放、旋转会改变它的世界坐标，不会改变它的 model 坐标。

在 model、world、camera 坐标系下，X，Y，Z 的范围都是无穷大，但是坐标系的基准不一样。

b、从 model 到 world 的变换

从模型本身的相对坐标变换到世界坐标，就是平移、旋转和缩放。

在本项目中 world = [1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1]

平移：

每次摁下键盘方向上下键改变 pos

调用函数：camera_at_zero(&device, pos, 0, 0)

平移操作即改变 camera 中 eye 位置，即改变坐标变换中 view 矩阵

旋转：

每次摁下键盘方向左右键改变 alpha

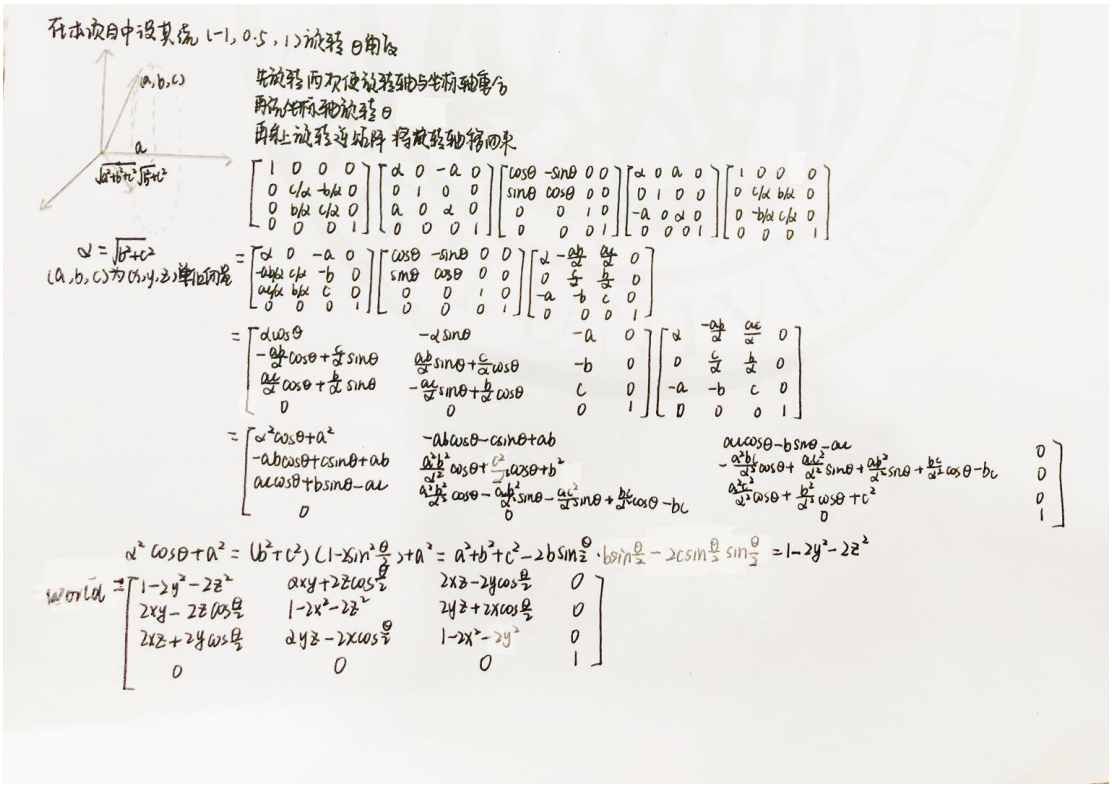
调用函数：matrix_set_rotate(&m, -1, -0.5, 1, alpha)

例：

旋转改变 world 坐标系

在本项目中，设其绕(-1,0.5,1)旋转，先旋转两次使旋转轴与坐标轴重合，再绕坐标轴旋转 alpha 角度，再乘上旋转逆矩阵，将旋转轴转回来。

按照下图的方式进行 world 矩阵的求解：



c、从 world 到 camera 的变换

相机也是世界里的物体，假定相机的中心点在世界里的位置是 C（Cx, Cy, Cz）

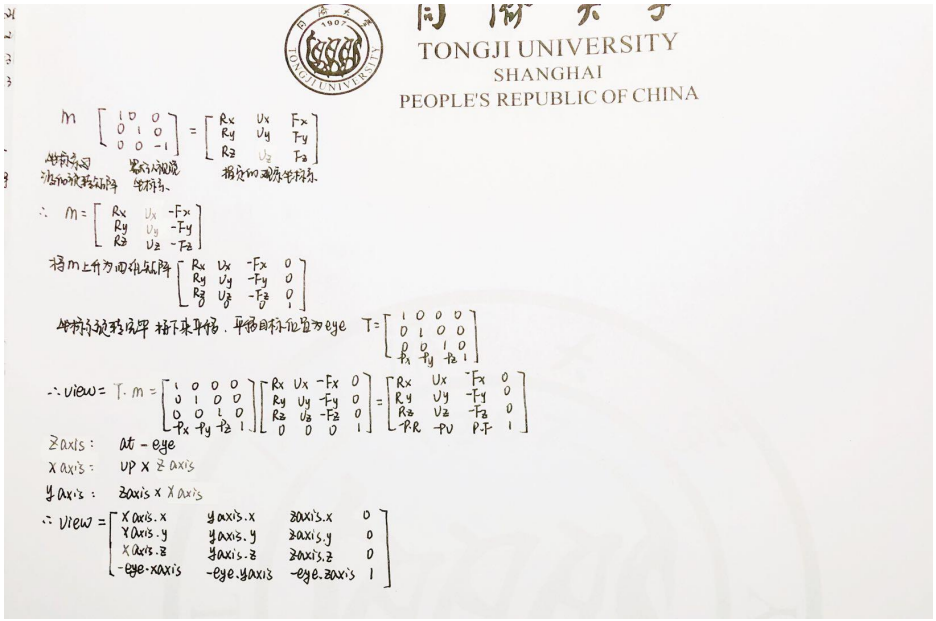
相机正在看着某个方向，我们假定相机正在看的点的位置是 I（Ix, Iy, Iz）

那么，相机的 Z 轴就是它看的方向的向量，即 CI 向量，也就是 I - C =（Ix-Cx, Iy-Cy, Iz-Cz），我们将其标准化得到 Z 轴单位向量。

然后我们取世界坐标系里的 up 向量（0，1，0），up' = up - (up • Z)Z，将 up' 标准化得到 Y，Y×Z = X

通过 up 叉乘 Z，可以得到一个向量 X1，将 X1 标准化（即使其模为 1），就得到了 X 轴的单位向量。再通过 Z 轴的单位向量与 X 轴的单位向量叉乘，即 Z×X，就得到了 Y 轴的单位向量。

证明如下图所示



构造坐标系转换的变换矩阵：

1）世界坐标系中，相机原点为（Cx, Cy, Cz），在相机坐标系中为（0，0，0）

所以，（0, 0, 0） = Xiw*（Cx, Cy, Cz）

2）世界坐标系中，相机的三个轴为 X+C（Xx+Cx, Xy+Cy, Xz+Cz）, Y+C（Yx+Cx, Yy+Cy, Yz+Cz）, Z+C（Zx+Cx, Zy+Cy, Zz+Cz），但在相机坐标系下为（1，0，0），（0，1，0），（0，0，1）

所以

（1, 0, 0） = Xiw*（Xx+Cx, Xy+Cy, Xz+Cz）

（0, 1, 0） = Xiw*（Yx+Cx, Yy+Cy, Yz+Cz）

（0, 0, 1） = Xiw*（Zx+Cx, Zy+Cy, Zz+Cz）

结合上面 2 步的四个式子，可以求出 Xiw

$$X_{iw} = \begin{pmatrix} X_x & X_y & X_z & -X \bullet C \\ Y_x & Y_y & Y_z & -Y \bullet C \\ Z_x & Z_y & Z_z & -Z \bullet C \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Xiw 表示先把坐标系移动到相机的原点处，然后再旋转来调整到相机的 X，Y，Z 轴。

d、从 camera 到 perspective 的变换

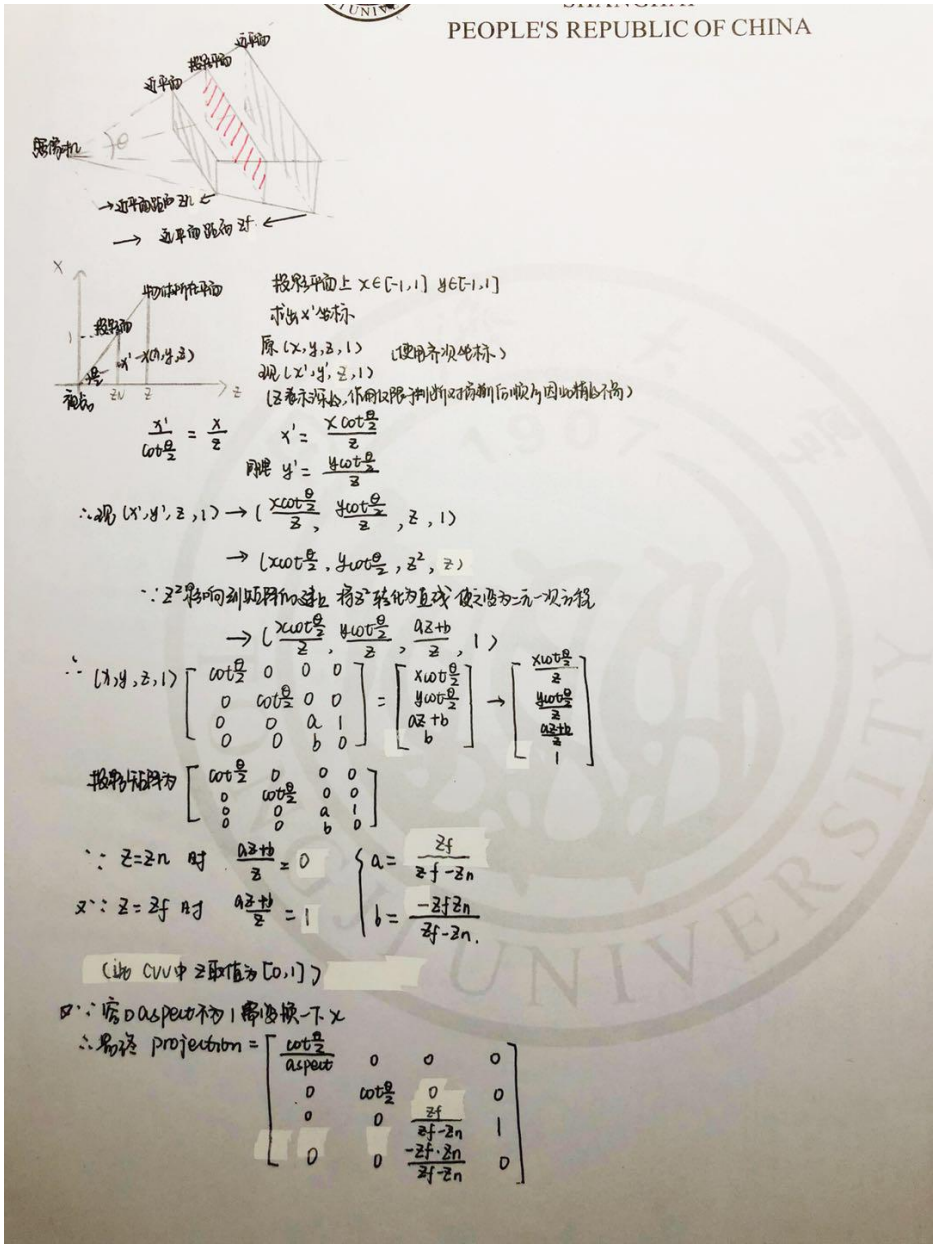
投射矩阵的作用就是把一个视椎体变换到一个规范化设备坐标上。

视椎体是能看到的区域，从眼睛或者摄像机的焦距一直到能看到的最远距离其中的空间。由于越远的东西看上去越小，可以看到的范围也就越大。这样，从焦距到可以看到的最远距离的可见区域增大，使得整个可见区域变成一个椎体，这就是视椎体。原点到焦距的距离称为近平面距离，用字母 n 表示；原点到最远可见距离称为远平面距离，用字母 f 表示。这两个平面的距离就是视椎体的“高（z 轴）”用字母 d 表示。视角代表一个椎体的顶点角度，这个角度越宽视野就越开阔。但是电脑屏幕是矩形的，这个椎体不是圆锥，而是一个四棱锥，因此存在垂直视角和水平视角的问题。

规范化设备坐标是一个立方体，在 WebGL 中，它的坐标是从(-1,-1,-1)到(1,1,1)。在这个区域中的东西就被正交投影到屏幕上。因此需要构造一个投射矩阵把视椎体的东西变换到规范化设备坐标上。因此只要把视椎体使劲捏成立方体就可以了。由于上面规定了它是个正四棱锥，所以 x 方向和 y 方向的处理是完全一样的。

首先，应该把这个视椎的 x 和 y 坐标都捏到规范设备区域坐标的范围内。完成这个步骤通常就可以得到一个细长的长方体。

证明如下图所示



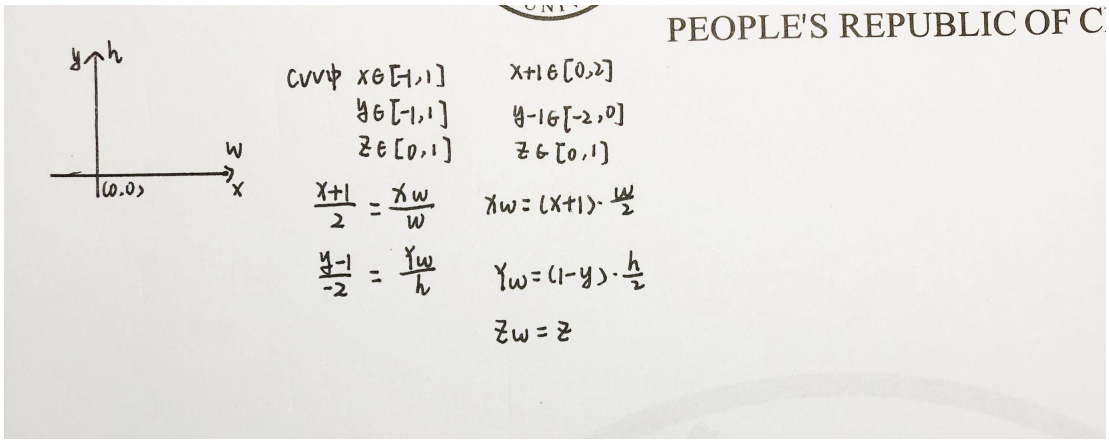
e、从 perspective 到 screen 的变换

在这里进行两步操作：归一化设备坐标和归一化的设备坐标转为屏幕坐标

归一化设备坐标就是透视除法，都除个 w，将最后一项变为 1。

定义屏幕分辨率为 xs × rs，要将透视坐标系里的点映射过来。屏幕坐标系的原点是左上角，而 perspective 里的原点（0，0）在屏幕中应该位于屏幕中央，即（xs/2, ys/2）。那么 perspective 里的原点（0，0）会映射为（xs/2, ys/2），即位移一个（xs/2, ys/2，0）。屏幕坐标系中 x 的范围是[0, xs)，y 的范围是[0, ys)。而 perspective 坐标系中 x 和 y 的范围是[-1,1]，所以这个映射还要满足-1 映射到 0，1 映射到 xs 或 ys。

证明如下所示：



代码如下：

```
//归一化得到屏幕坐标
void transform_homogenize(const transform_t *ts,vector_t *y,const vector_t *x)
{
    float rhw=1.0f/x->w;
    y->x=(x->x*rhw+1.0f)*ts->w*0.5f;
    y->y=(1.0f-x->y*rhw)*ts->h*0.5f;
    y->z=x->z*rhw;
    y->w=1.0f;
}
```

之后，使用公式 Transform = world * view *projection 将顶点坐标乘以 transform 后再转换为屏幕坐标。

f、一些总结

一个 3D 物体显示到电脑屏幕上，需要经过 4 重坐标变换：

screen Xsp **perspective** Xpc **camera** Xcw **world** Xwm **model**

在实际的渲染引擎运行中，Xsp 和 Xpc 基本不会改变，因为屏幕分辨率很少会改变。Xcw 会在相机移动和旋转时改变。Xwm 会在物体平移、旋转和缩放时改变。

5、绘制——以线框形式绘制立方体

a、立方体每四个点构成一个平面，因此绘制立方体需画六个四边形。

b、四边形中每三个点构成一个三角形，因此画一个四边形需画两个原始三角形。

c、 将每个顶点坐标乘以 transform 矩阵转换到 CVV 中，判断该点是否在 CVV 范围内，若三个点中有一个点不在，不画这个三角形。

d、符合要求后，将顶点坐标再从 CVV 坐标归一化为屏幕坐标。（所有坐标除以 w，齐次坐标最后一项变为 1，并转化为屏幕坐标）。

e、一个三角形由三条线段构成，因此已知两点，绘制出一条线段。

已知两点画线段

1）两点为同一个点时，画出该点即可。

2）两点横坐标或纵坐标不同时相等时，即该线段为水平或垂直的，可通过循环画出每个点构成一条线段。

3）其他情况下，采用中点画线算法画直线，需要考虑 x、y 方向上哪个方向变化快。

f、中点画线算法：在画直线段的过程中，当前像素点为（xp ，yp ），x 方向上变化快，下一个像素点有两种可选择点 P1(xp +1,yp)或 P2(xp +1,yp +1)。若 M=(xp +1,yp +0.5)为 P1 与 P2 之中点，Q 为 P 理想直线与 x=xp +1 垂线的交点。当 M 在 Q 的下方，则 P2 应为下一个像素点；M 在 Q 的上方，应取 P1 为下一个像素点。

二、光栅化及纹理实现

1、基本要求及本质

将一组三位顶点光栅化为 2d。

光栅化其实就是将几何数据经过一系列变换后最终转换为像素，从而呈现在显示设备上的过程。它的本质是坐标变换、几何离散化。

2、基本知识

a、图元是通过顶点定义的图形元素，包括点、线段、多边形、位图等等。

b、片元是带有一系列属性的图像元素，比如位置、颜色、深度值、纹理坐标等属性。

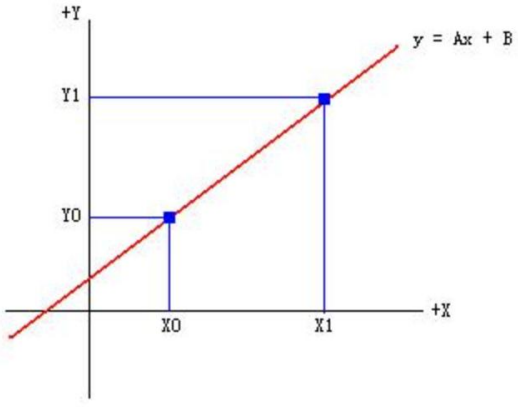
c、光栅化就是通过插值把一个图元过滤成能够在屏幕上表示它的一系列离散的片元，并通过片元操作把它们最终以像素的形式显示在帧缓冲中。

d、过滤的意思就是通过样本重建信号，这里的意思就是通过对多边形在 3D 空间中的顶点以及相关属性的采样重新在屏幕上建立可见图像。

e、在实时图形学中，光栅化基本上都是基于对多边形进行扫描线转换（scan-line converting）。把一个三角形的三个顶点所包围的区域转换成和屏幕水平方向平行的由像素组成的一条条扫描线。

3、线性关系与线性插值

在 2D 平面上，一条直线可以表示为斜截式：y = Ax + B。这个形式也表示变量 x 和 y 是线性关系。也就是说，只要参数 A 和 B 定下来，则 x 和 y 就有了一个固定的对应关系，有一个 x，在直线上就有唯一一个 y 和它对应。更具体地说，比如 x 的范围是[X0, X1]，则对应的 y 范围就是[Y0, Y1]。如下图所示：



这同时也是一个简单的线性插值的关系。从线性插值公式来看：

$$\frac{y-y0}{y1-y0}=\frac{x-x0}{x1-x0}\Rightarrow$$

$$y=\frac{y1-y0}{x1-x0}x-\frac{y1-y0}{x1-x0}x0+y0\Rightarrow$$

$$y=Ax+B\begin{cases}A=\dfrac{y1-y0}{x1-x0}\\ B=-\dfrac{y1-y0}{x1-x0}x0+y0\end{cases}$$

上面的式子表明，线性关系的两个变量可以进行线性插值，线性插值公式和直线公式其实是一致的。也就是说，如果两个变量是线性关系，就可以对它们进行线性插值，从而从一个变量 x 得到另一个变量 y。

4、cvv 裁剪

经过相机矩阵的变换，顶点被变换到了相机空间。这个时候的多边形也许会被视锥体裁剪，但在这个不规则的体中进行裁剪并非那么容易的事情，裁剪被安排到规则观察体(CVV)中进行，CVV 是一个长方体，x, y,的范围都是[-w，w]，z 的范围是[0，w]，w 规范化后因为 1。多边形裁剪使用这个规则完成的。

5、视口变换

a、对顶点实施透视投影变换之后，顶点变成了 4D 的齐次坐标的形式，从而进入了固定流水线中的裁剪空间，等待进行图元装配（Primitive Assembly）并针对图元进行 CVV 裁剪。裁剪之后的图元顶点接下来即将接受的处理是透视除法。透视除法把顶点从 4D 的齐次形式再次变回 3D 的普通形式，变化之前由于采用了精心设计的透视投影矩阵进行过处理，因此能够在裁剪过程中幸免于难的顶点以及新生的顶点都被“透视除法”成了归一化的设备坐标（NDC）（归一化成屏幕坐标）。这个坐标系实际上就是规则观察体——CVV 所包围的有限空间。

b、屏幕处理：

首先要把 CVV 中的顶点变到视口中。视口是可以在屏幕的窗口上看到 3D 图元的一个矩形区域，它把虚拟的 3D 世界与真实的计算机屏幕连接起来。 视口不是唯一的，可以有任意多个，每一个视口都可以有自己的观察姿态，可以用窗口坐标所表示的 left，top 以及 width 和 height 来定义，分别表示视口在窗口中的左上角坐标以及视口的长和宽。采用线性插值技术把顶点从 CVV 的 xy 平面上变换到视口中。把 CVV 中的 x 和 y 从[-1, 1]变换到视口的[left, left+width]以及[top, top + height]中，使用如下公式：

$$\frac{x_{cvv}-(-1)}{1-(-1)}=\frac{x_{vp}-left}{width}$$

$$\frac{y_{cvv}-(-1)}{1-(-1)}=\frac{y_{vp}-top}{height}$$

其中 x_{vp} 和 y_{vp} 是视口中的 x 和 y，可以通过这两个公式计算出来。处理之后，所有图元的顶点都从 CVV 中变换到视口中。

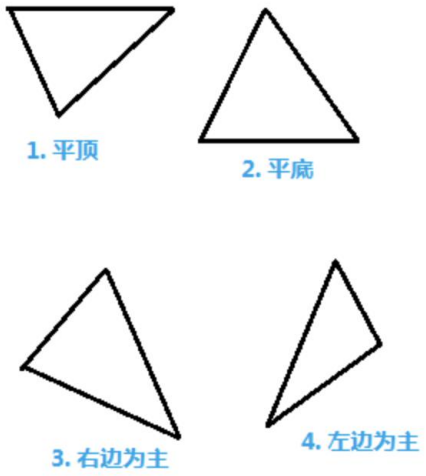
之后，把图元进行光栅化——从连续的虚拟空间变换到离散的屏幕空间——对

图元以及它们的所有属性进行过滤（Filter），以一定的规则变成像素的前身——片元（Fragment）。因为 图元顶点经过了视口变换后，虽然进入了窗口坐标中，但这个时候还看不到。

6、三角形的光栅化

a、三角形的分类

为方便光栅化，将三角形划分为以下 4 类：

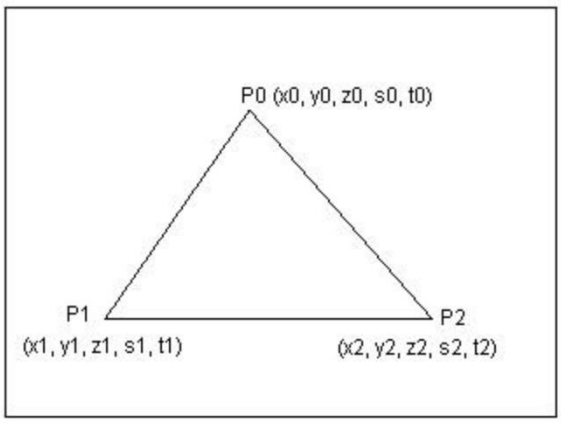


b、仿射纹理映射

假设纹理坐标 s、t 和屏幕 x 和 y 是线性关系。即：

$$s = Ax + B$$
$$t = Cx + D$$
$$s = Ay + B$$
$$t = Cy + D$$

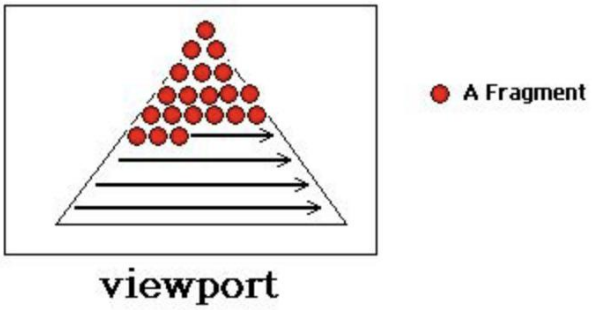
其中，s 和 t 是每个顶点的纹理坐标值，而 x 和 y 是视口中的屏幕坐标值。在这个假设的基础上进行了 s 和 t 对 x 和 y 的线性插值，为仿射纹理映射。如下图，是视口中的一个平底三角形：



它有三个顶点 P0，P1 和 P2，分别有相应的 x，y 和 z 三个坐标，其中 x 和 y 是从 NDC 通过视口变换转换过来的，而 z 可能是 NDC 的数据，也可能是从 NDC 变成视口自己的 z 范围中。

s 和 t 是每个顶点的纹理坐标值，它是一直从模型坐标带过来或者是通过 API 自动生成的，始终没有进行过处理。

如下图所示通过线性插值来进行扫描线的转换：



在第一层循环的时候， 通过左右两边的直线方程以及当前的 y，计算出左边线段的 x 和右边线段的 x，左边线段的 s、t 和右边线段的 s、t。 然后计算出 s 和 t 针对于的 x 变化量。

第二层循环就是实际绘制扫描线,绘制的同时根据纹理坐标变化量更新 s 和 t，然后把 s 和 t 所指向的纹理值赋给当前的插值像素点

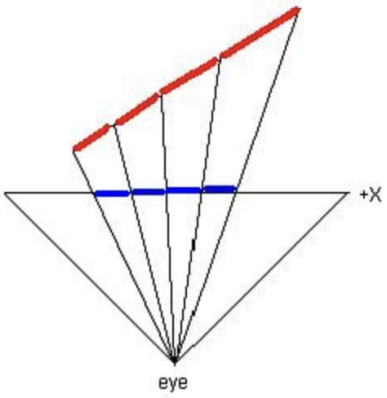
这个过程使用了最简单的替换（replace）纹理混合方式，直接把纹理颜色替换到帧缓冲中当前的位置。而且在替换之前没有作任何的片元操作,比如深度测试、蜡板测试、Alpha 测试以及混合等等。也没有考虑离散像素的填充规则，是一个最简单的光栅化框架。

当然，纹理坐标和屏幕坐标是线性的假设是简单且错误的，因此，接下来我们对纹理映射进行了改进。

c、改进

依据上面的算法计算 sleft、sright 以及 tleft、tright 的时候，做了关于 y 的线性插值。这表明在 y 方向上，纹理坐标 s 和 t 的变化和 y 的变化是按照线性、均匀的方式处理的。另外，纹理坐标 s 和 t 的扫描线步长 sstep 和 tstep 的计算，是根据扫描线的长度平均分配纹理变化量，也是按照线性、均匀的方式处理的。

但是，投影平面上的线性关系，还原到空间中便发生了改变：



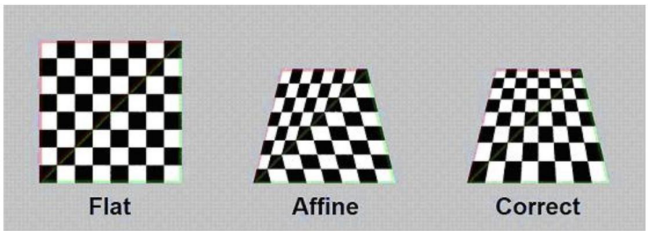
这张图是相机空间的一张俯视图。一个多边形通过透视投影的方式变换到了投影平面上,图中红色的是空间中的多边形,蓝色的是变换到投影平面之后的多边形。

在投影平面上的蓝色线段被表示成若干个相等的单位步长线段，相当于我们在上面的算法中递增扫描线位置的步骤——“++x”。

投影面上单位步长的线段所对应的投影之前的红色线段的长度不是相等的，从左到右所对应的长度依次递增。而实际上，我们的纹理坐标是定义在红色的多边形上的，因此纹理坐标的增量应该是和红色线段的步长对应的。

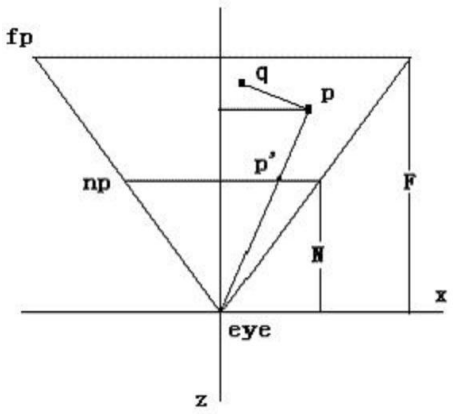
所以我们可以知道：投影平面上的 x、y 和纹理坐标 s、t 不是线性关系。

仿射纹理映射会导致下图所示的错误渲染：



可以看到，左边是多边形和投影平面平行时候的渲染。右边两个多边形和投影平面倾斜一定角度，中间的仿射纹理映射出现了渲染错误：纹理发生了扭曲了，这是直接对纹理坐标使用线性插值的结果，而最右边是使用带透视校正的透视纹理映射的效果。

d、透视纹理映射



上图是在相机空间的俯视图，eye 是眼睛的位置（原点）。np 和 fp 分别是近、远裁剪平面，N 和 F 分别是 z=0 到两个裁剪平面的距离。

pq 是一个三角形 pqr 在 xy 平面上的两个点，p 的坐标为（x，y，z），p’ 是 p 投影之后的点，坐标为(x’，y’，z’)，则有

$$(1)\begin{cases} x' = -N\dfrac{x}{z} \Rightarrow x = -\dfrac{x'z}{N} \\ y' = -N\dfrac{y}{z} \Rightarrow y = -\dfrac{y'z}{N} \end{cases}$$

在相机空间中，三角形 pqr 是一个平面，因此它内部的每一条边上的 x 和 z，以及 y 和 z 都是线性关系

$$(2)\begin{cases} x = Az + B \\ y = Az + B \end{cases}$$

所以，我们可以得到：

$$-\frac{x'z}{N} = Az + B \Rightarrow$$
$$-\frac{x'}{N} = A + \frac{B}{z} \Rightarrow$$
$$x' = -N\frac{B}{z} - AN \Rightarrow$$
$$(3)\begin{cases} x' = C\frac{1}{z} + D \Rightarrow \frac{1}{z} = Ax' + B \\ y' = E\frac{1}{z} + F \Rightarrow \frac{1}{z} = Ay' + B \end{cases}$$

从上面的式子，我们可以看到：x’和 1/z 是线性关系，y’和 1/z 也是线形关系。因此我们可以在投影面上通过 x’和 y’对 1/z 进行线性插值。再通过上面的（1）式计算出原始的 x 和 y，然后在 3D 空间中通过 x 和 y 计算出 s 和 t（x、y 和 s、t 都是在 3D 空间中的三角形上定义的，是线性关系）。这样就找到了投影面上一个点所对应的纹理坐标的正确值了。

以下可以对上述算法再进行一些优化，以减少计算次数：

在空间中，x、y 和 s、t 都是线性的，所以有关系：

$$(4)\begin{cases} x = As + B \\ x = At + B \\ y = As + B \\ y = At + B \end{cases}$$

把（4）带入（1），有

$$As + B = -\frac{x'z}{N} \Rightarrow$$
$$A\frac{s}{z} + B\frac{1}{z} = -\frac{x'}{N}$$

把（3）带入上式的中间项，可以得到：

$$A\frac{s}{z}+B(Cx'+D)=-\frac{x'}{N}\Rightarrow$$

$$\frac{s}{z}=Ex'+F\Rightarrow$$

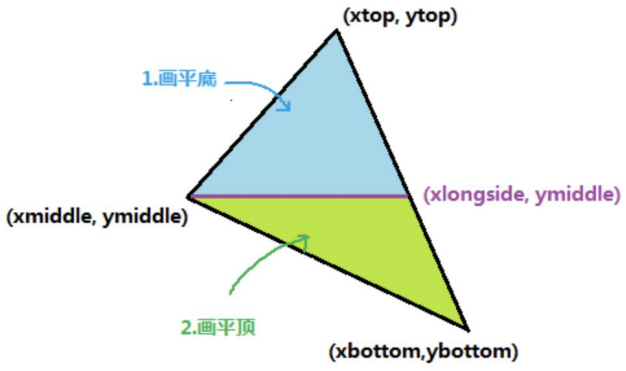
$$(5)\begin{cases}\frac{s}{z}=Ax'+B\\ \frac{t}{z}=Ax'+B\\ \frac{s}{z}=Ay'+B\\ \frac{t}{z}=Ay'+B\end{cases}$$

可以看到：s/z、t/z 和 x’、y’也是线性关系。且已知 1/z 和 x’、y’是线性关系。所以我们可以对 1/z 关于 x’、y’插值得到 1/z’，然后对 s/z、t/z 关于 x’、y’进行插值得到 s’/z’、t’/z’，然后用 s’/z’和 t’/z’分别除以 1/z’，就得到了插值 s’和 t’。

具体的代码实现如下：

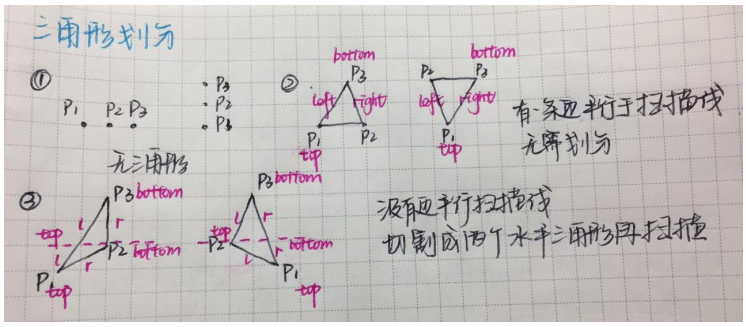
```
void vertex_rhw_init(vertex_t *v)//若点不在同一深度，颜色/纹理的渐变就需要把z方向给考虑进去
{
    float rhw=1.0f/v->pos.w;
    v->rhw=rhw;
    v->tc.u*=rhw;//纹理坐标同除点的w
    v->tc.v*=rhw;
    v->color.r*=rhw;//RGB颜色坐标同除点的w
    v->color.g*=rhw;
    v->color.b*=rhw;
}
```

e、任意三角形的光栅化



如上图，对于任意的一个三角形，先求出一个特殊点，之后画一个平底三角形，再画一个平顶三角形。

考虑下图中所示的三种情况：



先对三点进行排序：假设 p1 为最低点，p2 为次低点,p3 为最高点。

通过斜率判断所要进行光栅化的三角形哪种形态，主要用到了下面的代码：

```
k=(p3->pos.y-p1->pos.y)/(p2->pos.y-p1->pos.y);
x=p1->pos.x+(p2->pos.x-p1->pos.x)*k;
```

```
int trapezoid_init_triangle(trapezoid_t *trap,const vertex_t *p1,const vertex_t *p2,const vertex_t *p3);
```

f、获取扫描线与顶点的两个交点

在这一步，我们通过线性插值计算了出交点的顶点位置，纹理坐标，色彩 RGB，rhw。

g、根据两个端点初始化计算扫描线的起点和步长

在这里，我们对起点和步长进行了如下定义：

起点：最左边的那个点的整数型

步长：横坐标 x++后，顶点位置，纹理坐标，色彩 RGB，rhw 的变化量

h、画扫描线

从扫描线的起点开始向右绘制直到结束，算法逻辑如下：

```
begin:
    if 当前位置的深度比存储了的更近（通过 rhw 判断）
        更新深度缓存 zbuffer
        if 颜色模式
            framebuffer 添加颜色信息（0xffffff）
        if 纹理模式
            framebuffer 添加纹理信息
end;
```

颜色的存储是:0xffffff，前两个 ff 为 R，中 间两个 ff 为 G，最后两个 ff 为 B

7、添加纹理

首先，我们需要读取 bmp 文件。

bmp 文件的文件头，保存了包括文件格式标识、颜色数、图像大小、压缩方式等信息，为 54 字节。大小一项在宽:ox0012 和 ox0016

54 字节以后，如果是 16 色或 256 色 bmp，则还有一个颜色表，但 24 位色 bmp 没有这个。24 位色 bmp 文件中，每三个字节表示一个像素颜色。bmp 文件中采用的是 BGR(和 RGB 反过来了)。像素数据量并不完全等于图像的高度乘以宽度乘以每一位像素的字节数，而是可能略大于这个值。原因是对齐，每一行像素数据的长度若不是 4 的倍数，则填充一些数据使它是 4 的倍数。所以在计算分配空间的时候要把这个算上，否则可能导致分配的内存空间长度不足，造成越界。具体的修正代码如下：

```
int filepos=54;
for(int i=0;i<iwidth;i++)
{
    for(int j=0;j<iheight;j++)
    {
        fseek(pfile,filepos,SEEK_SET);
        fread(&B,1,1,pfile);
        fseek(pfile,filepos+1,SEEK_SET);
        fread(&G,1,1,pfile);
        fseek(pfile,filepos+2,SEEK_SET);
        fread(&R,1,1,pfile);
        filepos+=3;

        int r=(int)R;
        int g=(int)G;
        int b=(int)B;
        t[i][j]=(r<<16)|(g<<8)|b;

    }
}
```

之后，把要渲染的像素写到 framebuff 中，再将 framebuff 渲染到屏幕上就完成了所有步骤。

三、光照

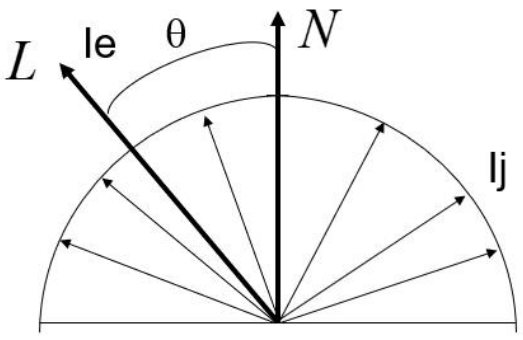
1、基本介绍

物体的颜色是它反射的光的颜色。光是白色的，但它的白色是由许许多多不同颜色不同频率的光混在一起呈现的。实现光照，就要去模拟物体的反射光，让物体有明暗关系，有亮面有暗面。

2、漫反射

当光照射到物体上面，物体会反射自身的颜色，这是最基本的反射，会根据光照方向产生明暗关系。

如下图，取单独一个切面来进行分析：

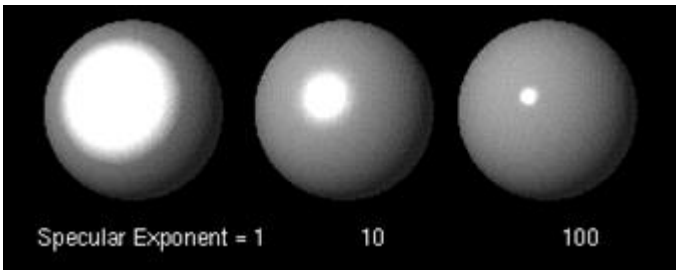


N 是物体的法线，L 是入射光的方向向量，le 是入射光的颜色，则反射光的颜色强度 lj 满足：lj = le Kd cos(θ) 或者 lj = le Kd (L • N)。其中，Kd 是物体本身的颜色。光照方向向量和法线都是标准化之后的。颜色向量的 RGB 范围都是 0~1。

物体反射的颜色，是物体本来的颜色和照射物体的光的颜色的混合。物体本身颜色和光的颜色的混合比例通过 θ 来确定，当 θ 为 0，即光正对着使劲照物体，那物体就是很亮的光的颜色。当光正好和物体平行擦着过去，那物体就没颜色了，黑色的。光对物体越正，物体就越亮，否则越暗。

3、高光

光滑的物体会有一小部分特别亮的地方，趋近于镜面反射。



如上图所示，中间圆形白色就是高光。同一个小球，左上偏亮右下偏暗，这样的明暗关系是漫反射。高光颜色强度公式为：

$$lj(V) = le\ Ks\ (V \bullet R)^{spec}$$

高光和视角有关系，V 是视角向量。Ks 是物体镜面反射的颜色，一般是白色。R 是反射光的方向向量，即 L 入射光关于法线 N 的对称向量。spec 是反射强度。

镜面反射（高光）的颜色是入射光的颜色和镜面反射光的颜色的叠加。镜面反射光的强度取决于视角以及 spec。

当 V 和 R 平行时，即反射的光直射眼睛，会特别亮。到 V 和 R 垂直时，光照全都照不进眼睛，就没有镜面反射（高光）。

4、环境光

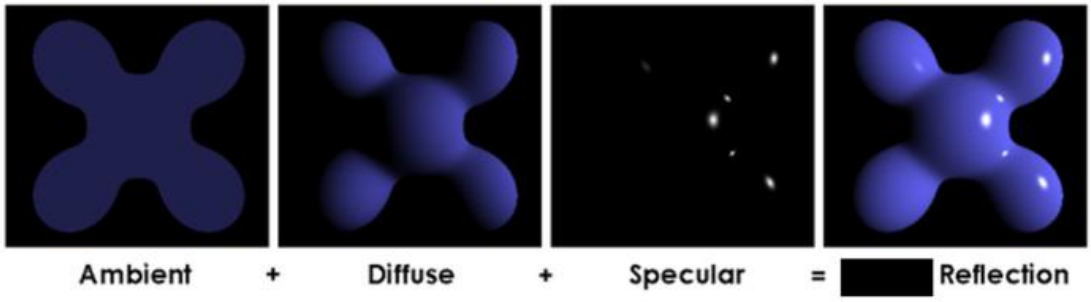
环境光颜色计算公式为：lj = la Ka。la 是环境光的颜色，Ka 是物体的环境光系数，一般而言 Ka = Kd = 物体本身的颜色，只有高光的 Ks 不是物体本身的颜色，而是偏白色。

5、光照计算公式

综合上述 3 中反射，一个物体在光照下拥有明暗关系之后的颜色公式如下：

$$\text{Color C} = \sum_{lights} \text{specular + diffuse + ambient components}$$
$$= (Ks \sum [le\ (R \bullet E)^s]) + (Kd \sum [le\ (N \bullet L)]) + (Ka\ la)$$

即所有光源的高光计算结果+所有光源的漫反射计算结果+环境光颜色计算结果。场景中的环境光只有一个，而发光光源可能有好几个，所以高光和漫反射要对每个光源进行计算，然后相加，具体情况如下图所示：



6、小结

计算光照的时候是在相机坐标系空间下。所以 E（视角向量）永远都是（0，0，-1），（我们往正前方看其实是往 Z=1 的地方看，它的反向量就是 -1）。因为光照计算涉及方向向量 L, E, N, R，而从 image 变换到 perspective 空间会扭曲图形损失信息，所以我们需要在仿射空间（即 model， world， image 3 个 space）进行变换。

在 image 空间下，法线等方向向量都要变换到 image space，但是，法线的变换是只有旋转，没有位移和缩放的。因为法线如果位移，即在法线向量上加一个位移向量，两个向量如果不平行，那么和向量的方向会和原法线不一致。

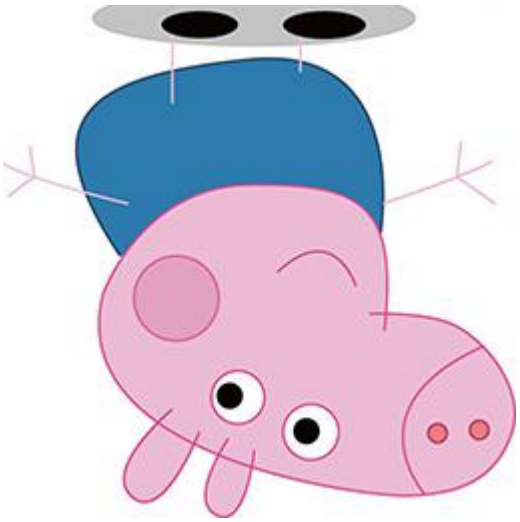
当 E 和 L 不在一边的时候，即我们在看物体正面，光从物体背面照过来，我们是什么都看不见的，是黑色，这种情况直接跳过不再计算。当 E 和 L 在同一边，但法线 N 在另一边的时候，算出来是负的，这时要把法线取反再进行计算。

计算结果要保证最后的颜色 RGB 在 0~1 范围内。

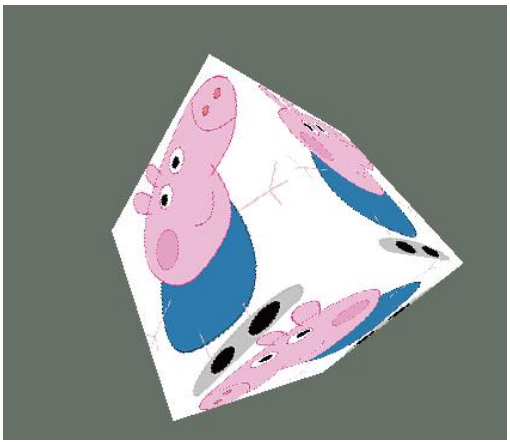
四、实验结果

C++实验：

使用以下 bmp 图片进行实验：

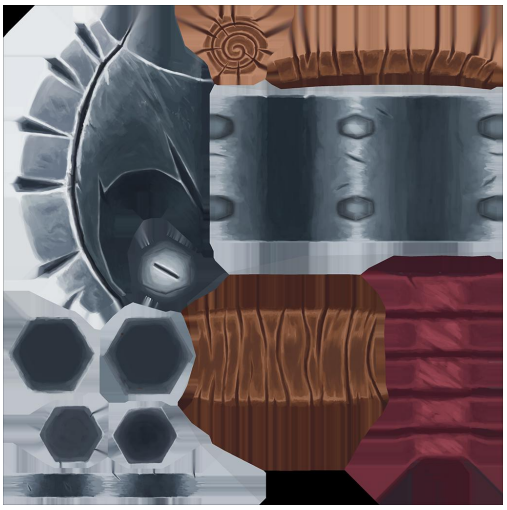


最终得到的实验结果如下：



python 实验：

使用以下图片进行渲染：



最终得到的实验结果如下：

五、代码结构及启动方式

C++版本：

render_cpp 文件夹下打开 render.sln 直接运行。

math.cpp

1. 如果 x 超出了边界，就选边界；如果 x 没有超出边界，就选 x；在判断颜色值的时候用到

```
int CMID(int x,int min,int max)
```

2. 向量相加 z=x+y

```
void vector_add(vector_t *z,const vector_t *x,const vector_t *y)
```

3. 向量相减 z=x-y

```
void vector_sub(vector_t *z,const vector_t *x,const vector_t *y)
```

4. 向量数量积

```
float vector_dotproduct(const vector_t *x,const vector_t *y)
```

5. 向量积

```
void vector_crossproduct(vector_t *z,const vector_t *x,const vector_t *y)
```

6. 计算插值：t∈[0,1]

```
float interp(float x1,float y1,float t)
```

7. 矢量插值：t∈[0,1]

```
void vector_interp(vector_t *z,const vector_t *x,const vector_t *y,float t)
```

8. 矢量归一化

```
void vector_normalize(vector_t *v)
```

9. 矩阵相加 c=a+b

```
void matrix_add(matrix_t *c,const matrix_t *a,const matrix_t *b)
```

10. 矩阵相减 c=a-b

```
void matrix_sub(matrix_t *c,const matrix_t *a,const matrix_t *b)
```

11. 矩阵相乘 c=a*b

```
void matrix_mul(matrix_t *c,const matrix_t *a,const matrix_t *b)
```

12. 矩阵缩放 c=a*i

```
void matrix_scale(matrix_t *c,const matrix_t *a,float i);
```

13. 向量和矩阵相乘 y=x*m

```
void matrix_apply(vector_t *y,const vector_t *x,const matrix_t *m);
```

14. 重置为单位矩阵

```
void matrix_set_identity(matrix_t *m);
```

15. 重置为零矩阵

```
void matrix_set_zero(matrix_t *m);
```

16. 平移变换的平移矩阵

```
void matrix_set_translate(matrix_t *m,float x,float y,float z);
```

17. 缩放变换的缩放矩阵

```
void matrix_set_scale(matrix_t *m,float x,float y,float z);
```

18. 旋转变换的旋转矩阵

```
    void matrix_set_rotate(matrix_t *m,float x,float y,float z,float
theta);
```
19. 摄像头设置
```
    void  matrix_set_lookat(matrix_t  *m,const  vector_t  *eye,const
vector_t *at,const vector_t *up);
```
20. 透视投影矩阵设置
```
    void    matrix_set_perspective(matrix_t    *m,float    fovy,float
aspect,float zn,float zf);
```

transform.cpp

1. 矩阵更新，计算：transfrom=world*view*projection
```
    void transform_update(transform_t *ts);
```
2. 初始化，设置屏幕宽高和变换矩阵
```
    void transform_init(transform_t *ts, int width, int height);
```
3. 将点坐标进行坐标变换
```
    void  transform_apply(const  transform_t  *ts,  vector_t  *y,  const
vector_t *x);
```
4. 检查齐次坐标同 CVV 的边界用于视锥体的裁剪
```
    int transform_check_cvv(const vector_t *v);
```
5. 归一化，得到屏幕坐标
```
    void transform_homogenize(const transform_t *ts, vector_t *y, const
vector_t *x);
```

geometry.cpp

1. 若点不在同一深度，颜色/纹理的渐变就需要把 z 方向给考虑进去
```
    void vertex_rhw_init(vertex_t *v)
```
2. 顶点插值
```
    void vertex_interp(vertex_t *y,const vertex_t *x1,const vertex_t
*x2,float t)
```
3. 顶点加法
```
    void vertex_add(vertex_t *y,const vertex_t *x)
```
4. 顶点除 width，用以得到相应点的纹理与颜色
```
    void vertex_division(vertex_t *y,const vertex_t *x1,const vertex_t
*x2,float w)
```
5. 根据左右两边的端点，初始化计算扫描线的起点和步长
```
    void trapezoid_init_scan_line(const trapezoid_t *trap,scanline_t
*scanline,int y)
```
6. 按照 Y 坐标计算出左右两边纵坐标等于 y 的顶点
   获取扫描线与顶点的两个交点
```
    void trapezoid_edge_interp(trapezoid_t *trap,float y)
```
7. 将三角形分割成可以扫描的水平三角形
```
    int  trapezoid_init_triangle(trapezoid_t  *trap,const  vertex_t
*p1,const vertex_t *p2,const vertex_t *p3)
```

renderer.cpp 渲染实现

1. 绘制扫描线(扫描线填充算法)
   void device_draw_scanline(device_t *device,scanline_t *scanline)
2. 主渲染函数
   void device_render_trap(device_t *device,trapezoid_t *trap)
3. 画原始三角形
   void device_draw_primitive(device_t *device,const vertex_t *v1,const vertex_t *v2,const vertex_t *v3)

device.cpp 渲染设备

1. 设备初始化，fb 为外部帧缓存，非 NULL 将引用外部缓存帧缓存（每行对齐 4 字节）
   void device_init(device_t *device,int width,int height,void *fb)
2. 清空 framebuffer 和 zbuffer
   void device_clear(device_t *device,int mode)
3. 设置当前纹理
   void device_set_texture(device_t *device,void *bits,long pitch,int w,int h)
4. 删除设备
   void device_destroy(device_t *device)
5. 画点
   void device_pixel(device_t *device,int x,int y,IUINT32 color)
6. 画线段
   void device_draw_line(device_t *device,int x1,int y1,int x2,int y2,IUINT32 c)
7. 根据坐标读取纹理
   IUINT32 device_texture_read(const device_t *device,float u,float v)

loadbmp.cpp 加载图片

1. 加载 bmp 图像
   void loadbmp(const char *szfilename,IUINT32 t[256][256])

light.cpp 光照

1. 设置光线颜色
   void set_light_color(light_t *light,float r,float g,float b)
2. 设置光照位置
   void set_light_pos(light_t *light,float x,float y,float z)
3. 全局环境光
   void global_ambient_light(device_t *device,light_t *light)
4. 漫反射光
   void diffuse_reflection_light(device_t *device,light_t *light)
5. 开启光照
   void open_light(device_t *device,light_t *light)

python 版本：

render_python 文件夹：

在此处使用命令行，依次输入如下命令：

```
$ pip install -r requirements.txt
$ python setup.py build_ext --inplace && python test_render.py
```

六、参考资料

1、深入探索透视纹理映射

https://blog.csdn.net/u012419410/article/details/41989501

2、韦易笑老师的 mini3D

http://www.skywind.me/blog/archives/1498

3、视图矩阵（View Matrix）的推导

https://www.web-tinker.com/article/20177.html

4、投射矩阵(Projection Martrix)的推导

https://www.web-tinker.com/article/20157.html

5、OpenGL 学习脚印：OpenGL 坐标变换

http://www.360doc.com/content/14/1028/10/19175681_420522107.shtml