SoftwareGraphics

如果发现内容上还有什么错误，别介意，太弱了。

前言

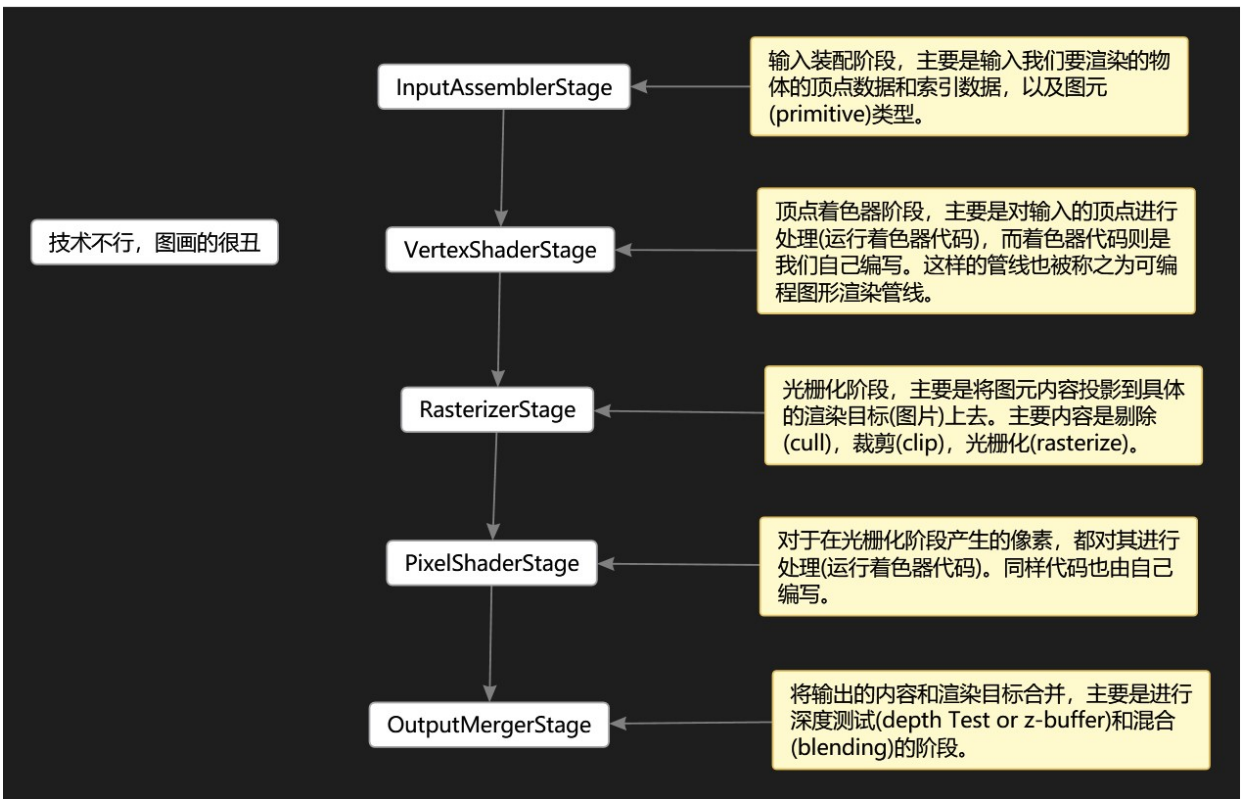
本篇文章内容主要是对于那些了解图形渲染管线(以下简称渲染管线)但是并不怎么熟悉渲染管线和一些管线中使用的算法的读者(找不到更好的词形容了)。主要内容是介绍如何实现一个简单的渲染管线(即实现软渲染)。

因此阅读之前建议对渲染管线以及着色器有一定的了解，以及有一定的代码能力。哦，不能忘记还要有一点点线代知识。

图形渲染管线

首先我们简单的介绍下什么是渲染管线。简单来说就是将输入数据渲染出来的过程，由于整个过程可以被分为几个阶段，我们就称之为渲染管线。

对于渲染管线来说，有如下阶段(只列出本文中会讨论的)。

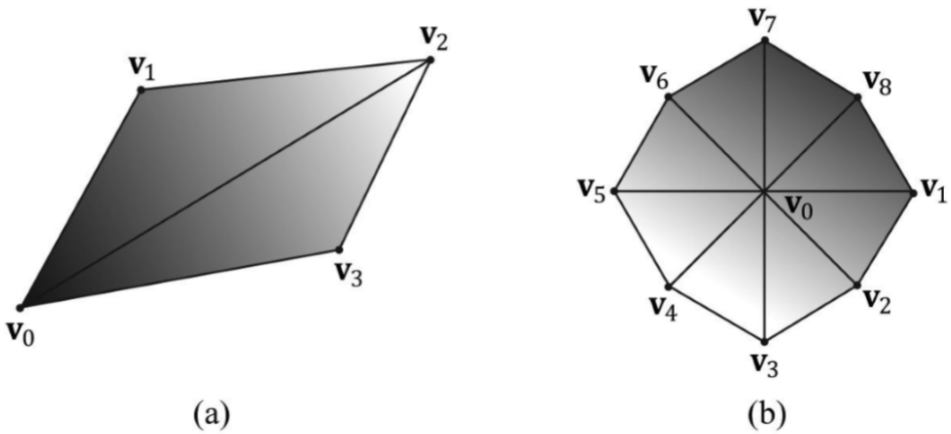


整个流程，图中其实很清楚了。但是有些内容还是稍微解释下。

什么是顶点数据，我们可以简单的认为就是在三维空间中的点的坐标，颜色，纹理坐标，法线等等数据。

什么是索引数据，可以这样认为，我们以数组的形式将一组顶点数据提交给渲染管线，那么索引即为顶点数组的索引，我们通过索引来构建我们渲染的物体，而不是顶点，这样可以节省下重复使用的顶点空间。

什么是图元，我们渲染的物体就是由图元构成的，而我们的索引数据构成的也是图元。通常我们使用的图元为三角形。以下就是使用三角形构成物体的例子。并且之后的内容我们都用三角形来表示图元(并且也只实现了三角形模式)。



什么叫做光栅化，我们的物体由三角形构成，我们只知道的是三角形的三个顶点，那么我们需要将三个顶点投射到一个单位大小的网格构成的二维平面上，这三个顶点在二维平面上仍然构成一个三角形，找出哪些网格在三角形内部的过程就叫做光栅化。

具体思路

首先要说明的是，由于只是简单的软渲染，所以有一些东西也没有去实现，以及性能上基本上很差。然后是关于一些算法，硬件并不是我们这样实现的，但是原理和想法是差不多的(因为要考虑GPU的特性)。

输入装配阶段

这个阶段的实现非常的简单，我们只需要提供一些函数来设置以及获取顶点数据，索引数据，图元类型即可。当然我们也需要记录当前我们使用的数据。

但是还是有一个小问题，即如何支持顶点属性。之前我们提到顶点数据除了点所在的位置，还可以有他的颜色，纹理坐标，法线。简单的一个想法是我们定义一个标准的顶点

结构，然后存储了顶点的坐标，颜色，纹理坐标，法线，深度值等信息，我们构建顶点数据的时候只需要填充我们想要使用的属性。我目前是这样做的。

不过图形API例如DirectX以及OpenGL并不是这样做的，它支持我们自定义顶点结构，但是在输入装配阶段还有一个输入布局属性。即用来描述我们输入的顶点属性的布局。这里我们就不讨论了。

顶点着色器阶段

这也是一个较为简单的阶段，一句话概括就是把我们会使用到的顶点找出来然后用着色器代码对其进行处理，然后构建好图元。

唯一一点可以提的是，我们完全可以只需要找出我们要使用的顶点数组范围，而不处理整个顶点数组。以及注意对于每个顶点进行着色器处理后，要记得对其进行透视除法。

并且因为后续需要，这里我记录了进行矩阵变换后的顶点的坐标，也记录了进行透视除法后的顶点的坐标。

光栅化阶段

这也是最为复杂的一个阶段，这一个阶段要进行的操作很多，例如剔除，裁剪，光栅化。

剔除

剔除，分为背面剔除和正面剔除或者不剔除。主要目的是将不符合要求的三角形去除。那么什么是不符合要求的三角形？我们是这样规定的，用三角形的顶点的排列顺序(顺逆时针)来定义三角形的这个面是正面还是背面。

三角形在空间中有两个面，你正看它他的排列顺序是一种，从背面看它排列顺序就是另外一种了。这样的定义主要是为了方便区分三角形的两个面。

并且不同的API定义正反两面的规则未必相同(标准不一样还有名词不一样真的难受，明明说的就是同一个东西还要思考思考)。这里我的定义是正面为顺时针，背面为逆时针。

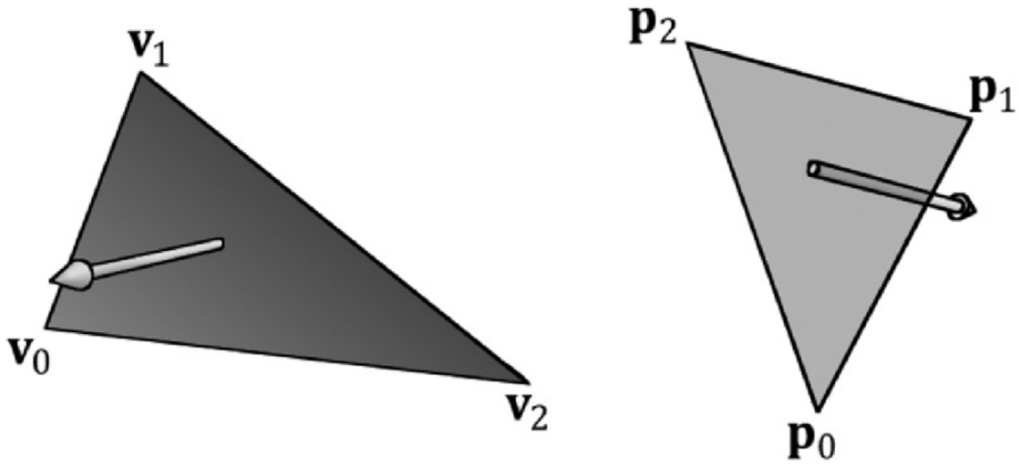


Figure 5.30. The left triangle is front-facing from our viewpoint, and the right triangle is back-facing from our viewpoint.

那么如何判断正面还是背面呢？如果直接在三维空间中讨论，那么我们不仅要考虑坐标顺序，还要考虑观察点或者说原点(摄像机坐标中)。

我们能够知道投影后的三角形是在同一个平面内的(透视除法后)，那么我们可以取出三角形顶点的$[X, Y]$ 分量。那么问题就变为了一个平面问题，我们只需要知道一个平面三角形它的顺逆时针而已。那么直接使用叉乘进行判断就好了。

至于剔除的目的，主要是用于优化性能的，对于很多物体来说，我们仅仅只能见到其一部分面，例如立方体我们最多同时看到3个面，那么那些没有必要的面，就完全没有必要进行渲染，因此我们可以设置剔除来避免渲染。

裁剪

裁剪，一句话来说就是将图元不在平截头体范围内的部分去掉。因为这部分我们是看不到的。

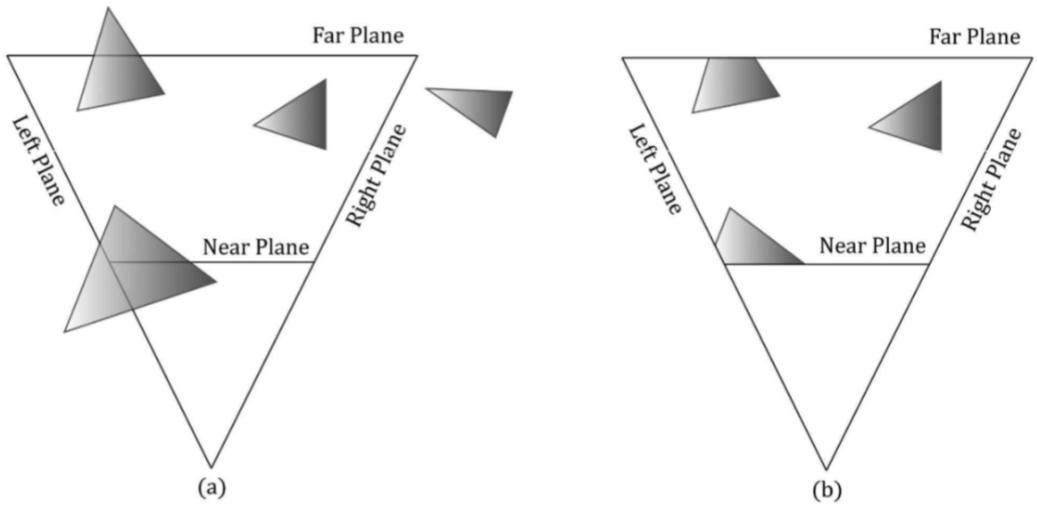


Figure 5.27. (a) Before clipping. (b) After clipping.

这里也终于有一个知道名字的算法了。叫做Sutherland-Hodgeman算法，用于裁剪的算法。

这个算法还是可以一句话来形容，枚举平截头体的每个边，然后用这个边把图元砍掉。

上面说的啥玩意啊。哪有那么简单。

这里还是详细介绍下算法流程，看不懂的可以搜索下。

首先我们还是要枚举每个平截头体的面(六个面)。然后对于每个面，我们*顺序考*虑由顶点构成的边。

- 对于一条边的起始顶点和结束顶点都在平截头体范围内的话，那么我们就将起始顶点加入到新的顶点数组去。
- 对于一条边的起始顶点和结束顶点都不在平截头体范围内的话，那么这两个顶点都不加入到新的顶点数组中去。
- 对于一条边的起始顶点在平截头体范围内，而结束顶点不在的话，那么我们就需要将起始顶点和这条边与平截头体的面相交位置的顶点加入到新的顶点数组中去。
- 对于一条边的起始顶点不在平截头体范围内，而结束顶点在的话，那么我们只需要将这条边和平截头体的面相交位置的顶点加入到新的顶点数组中去。

然后我们对新的顶点数组构成的图元继续考虑下一条平截头体的面，直到六个面都考虑完。

可能会有人疑问，这样的裁剪会产生在平截头体外的顶点，但是产生的点肯定是在其他面外的顶点(并且是目前没有考虑过的平截头体的面)，最后还是会被裁剪掉。

这个算法的思路其实不难理解，但是还有两个小问题比较麻烦解决。第一个是我们如何判断一个点在这个面的外面，第二个是我们如何裁剪边。

这个时候我们保存顶点的坐标就有用了(矩阵变换后和透视除法后的坐标)。透视除法结束后的点即在NDC空间中了。

我们考虑一个在NDC空间中的点$(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1)$ 。如果它在平截头体范围内的话。那么:

$$-1 \leq \frac{x}{w} \leq 1$$

$$-1 \leq \frac{y}{w} \leq 1$$

$$0 \leq \frac{z}{w} \leq 1$$

有了这些式子，那么我们判断一个点是否在平截头体内部就很简单了。

但是需要注意的是，这里并不能直接取判断其是否在$[-1, 1]$ 范围或者$[0, 1]$ 内，而是应该判断x 和y 是否在$[-w, w]$ 内，z 是否在$[0, w]$ 内。至于为什么，可以自己思考下。提示是关于w 的正负性。

而对于第二个问题，如何求边和面的交点，有了上面的经验，这个其实也很简单。这里我们以平截头体的左边的面为例子，来进行裁剪。

我们设边的起始点为s ，结束点为e ，设面和边的交点为$n = s + (e - s) * t$ 。我们知道n 在面上，那么n 变换到NDC空间中后，其x 坐标必然为-1 ，也就是说:

$$\frac{n.x}{n.w} = -1$$

$$\frac{s.x + (e.x - s.x) * t}{s.w + (e.w - s.w) * t} = -1$$

$$t = -\frac{s.x + s.w}{(e.x - s.x) + (e.w - s.w)}$$

可能有人会觉得要是除以0 怎么办，答案是不会出现这样的情况。可以思考下。提示是一条边中的两个端点中，最多只有一个在面上。

得到t 后，想要计算出n 则是非常简单的了。

$$n = s + (e - s) * t$$

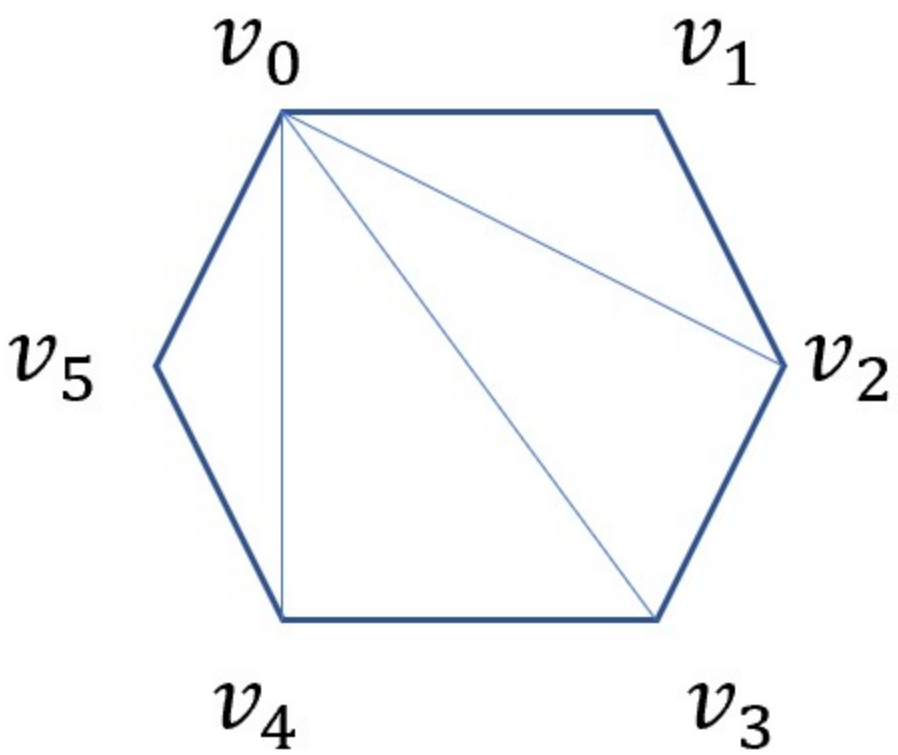
$$n = s - (e - s) * \frac{s.x + s.w}{(e.x - s.x) + (e.w - s.w)}$$

当然还要注意一个小细节，我觉得估计很多人可能会忘记，即记得对n 进行透视除法，通过插值计算出来的是不正确的，因此我们需要重新对其进行透视除法(毕竟泥有了除法操作)。

那么到这一步，裁剪也就完成了。

三角化

可能有些人已经发现，裁剪后的三角形未必还是三角形，因此我们还需要重新将图元变成三角形。这并不是一个很难的操作，你只需要保证加入的点是按照顺序排列的(顺逆时针)，那么固定一个点，就很容易将多边形分解成三角形了。并且为了方便，你还可以在这里将三角形全部变成逆时针的或者顺时针的，当然不改变也没关系，不过这样的话后面的内容可能要考虑顺逆时针反而平白无故增加工作量。

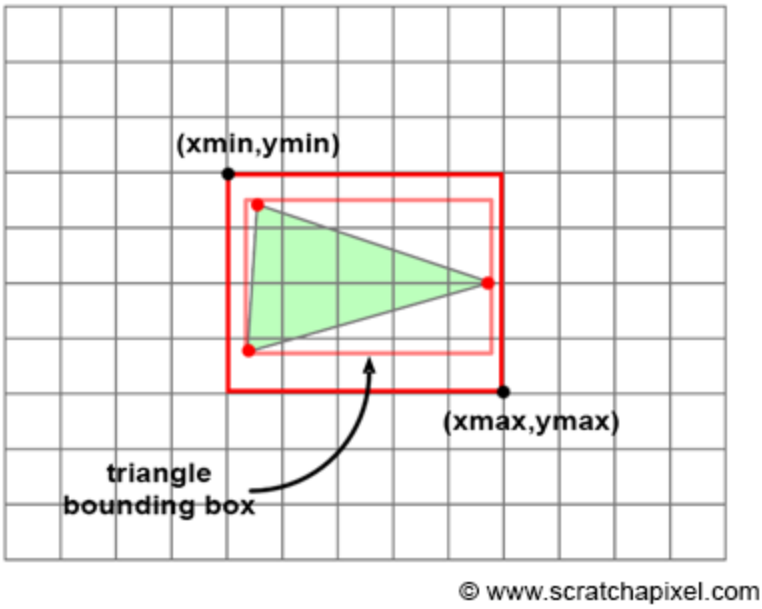


v_0 点就是我们固定的点，然后其余的点按照顺序可以分解成三角形。

光栅化

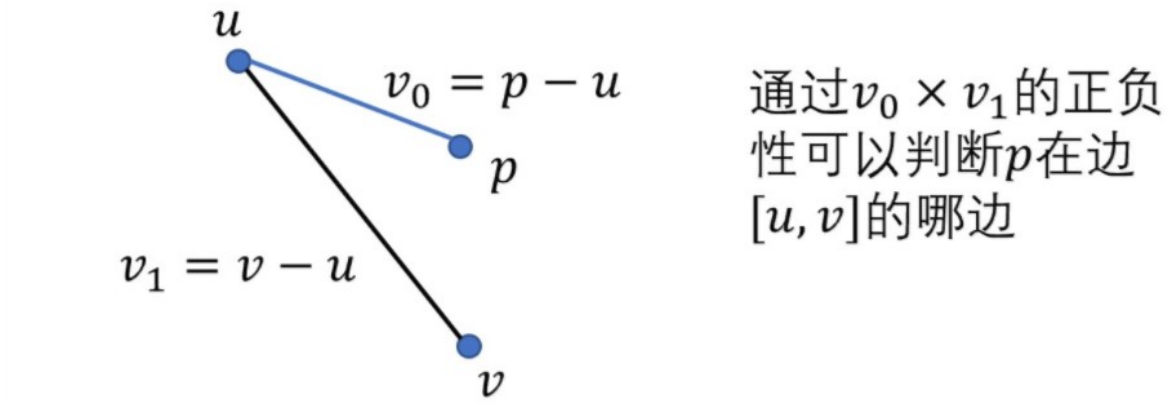
恕我直言，这应该就是最为复杂的部分了。不过基础思路还是一句话可以解决。对于每个图元，找到最小的包围盒，然后枚举包围盒中的像素，判断像素是否在包围盒内，如果在，那么就进行插值来获取这个像素对应的属性(例如坐标，颜色，纹理坐标，深度值等等)。

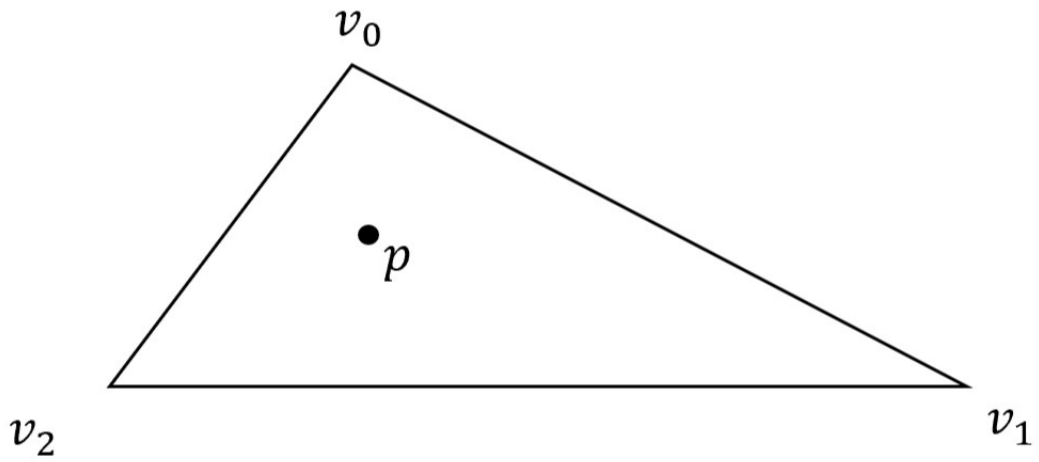
包围盒这里解释下，简单来说就是找一个平行于轴的矩形，然后其大小最小，但是可以把这个图元包含进来。其实就是找三角形顶点中的最小的XY和最大的XY(小声BB)。



那么现在还有两个需要解决的问题，一个是如何判断一个像素是否在三角形内，另外一个是如何进行插值。

第一个问题其实也很简单，和剔除的时候判断顺逆时针的方式差不多，都是使用叉乘来进行判断。考虑一条边$[u,v]$，我们有一个点p。





那么对于三角形来说，我们只需要判断三条边和p 的关系，就可以判断出p 是否在三角形内了。

最后一个问题，如何进行插值。我们来看这样的一个式子。

$$C_p = C_{v_0} * e_0 + C_{v_1} * e_1 + C_{v_2} * e_2$$

$$e_0 + e_1 + e_2 = 1$$

这是插值的形式(其中一个e 变为0的话，那么就是我们非常熟悉的形式了)，我们通过插值来计算出p 位置的颜色。其中只有e_0, e_1, e_2 是未知的，这也是我们需要求的。

从这里，我们可以想到一个东西，重心坐标。

$$(e_0 + e_1 \ldots + e_{n-1}) * p = e_0 * v_0 + e_1 * v_1 \ldots + e_{n-1} * v_{n-1}$$

而三角形的重心坐标又有另外一个称呼，面积坐标。简单来说：

$$e_0 = \frac{S_{\Delta v_1 v_2 p}}{S_{\Delta v_0 v_1 v_2}}$$

$$e_1 = \frac{S_{\Delta v_2 v_0 p}}{S_{\Delta v_0 v_1 v_2}}$$

$$e_2 = \frac{S_{\Delta v_0 v_1 p}}{S_{\Delta v_0 v_1 v_2}}$$

证明可以去翻wiki。

那么是不是我们就完成了光栅化？naive。还有啥问题啊，你别坑我哈。

可惜的是，确实还有问题。我们是对在NDC空间中的三角形进行光栅化，而我们的插值应该是对在3D空间中的三角形进行插值，而不是对NDC空间中那个进行透视除法后的三角形进行插值。

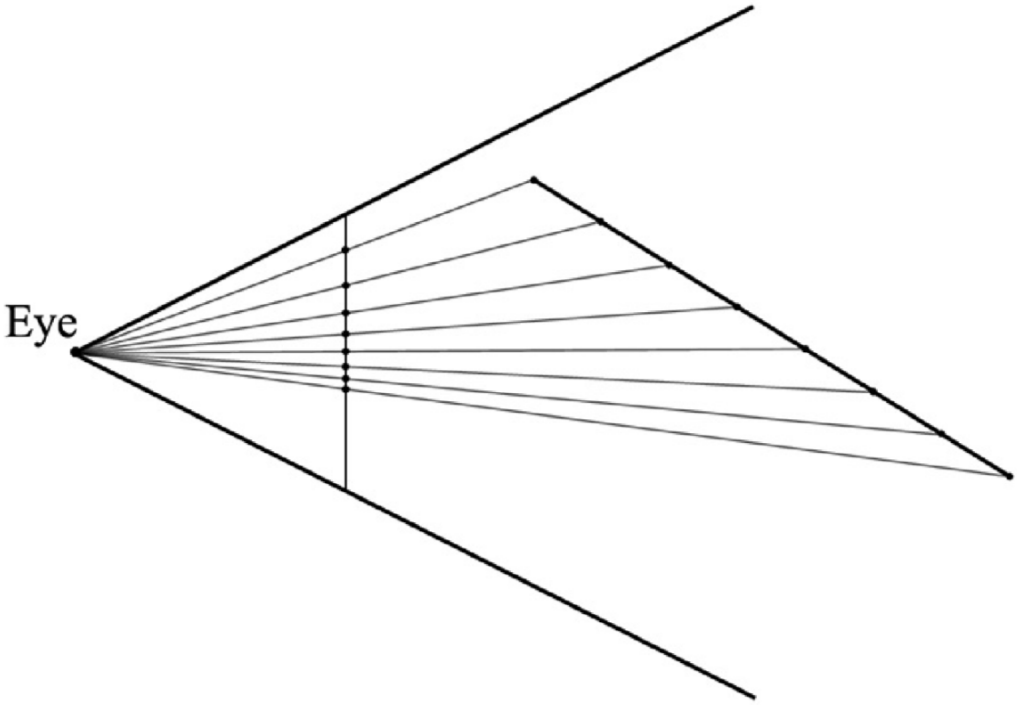
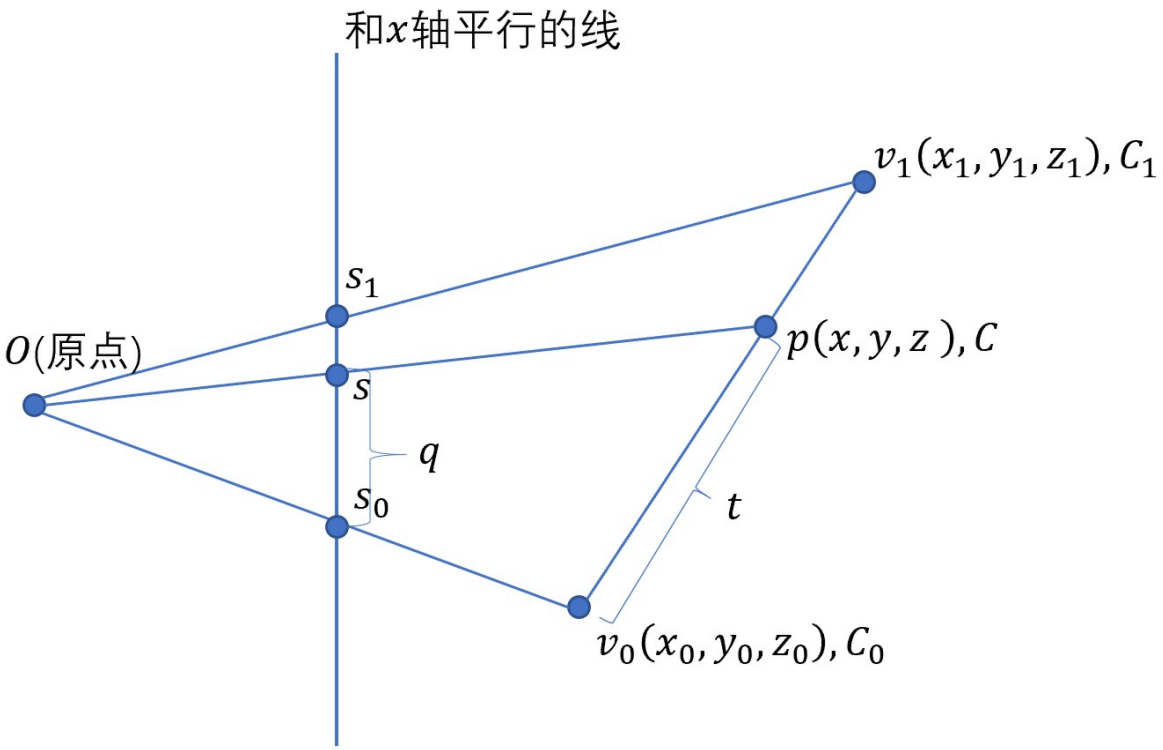


Figure 5.34. A 3D line is being projected onto the projection window (the projection is a 2D line in screen space). We see that taking uniform step sizes along the 3D line corresponds to taking non-uniform step sizes in 2D screen space. Therefore to do linear interpolation in 3D space, we need to do nonlinear interpolation in screen space.

这一步，我们叫做透视矫正插值(**Perspective Correct Interpolation**)。

至于思路，还是一句话。对于每个在三角形内的像素，其肯定能够找到对应的在3D空间中的三角形上的点，找到后，用其进行插值即可。但是实际上实现的话，我们没有必要这样做。因为直接去找到对应的点的代价相对较大。

我们考虑3D空间中线段上的点$p(x,y,z)$，然后考虑p 所在的线段的两个端点$v_0(x_0,y_0,z_0),v_1(x_1,y_1,z_1)$。我们设$p$ 点的属性为C，v_0 点的为C_0，v_1 点的为C_1。



那么我们可以得到这么一个式子:

$$\frac{z-z_0}{z_1-z_0}=\frac{C-C_0}{C_1-C_0}$$

C 是我们需要求的值，而我们只有C 和z 不知道。那么通常我们都会产生这样的一个想法，把z 用另外的变量代替，使得我们能够得到一个只关于C 的方程，然后就可以求解C。

很幸运的是，还真有这么一个性质可以用。那便是这样的一个式子。

$$\frac{1}{z}=(1-q)*\frac{1}{z_0}+q*\frac{1}{z_1}$$

倒数的插值是正确的，或许可以模仿这个式子，将C 也表达出来？然而并不能，证明过程将会解答(一大波式子预警)。

$$p=v_0+t*(v_1-v_0)$$

$$s=s_0+q*(s_1-s_0)$$

然后因为投影的缘故。

$$s = \frac{x}{z}, s_0 = \frac{x_0}{z_0}, s_1 = \frac{x_1}{z_1}$$

$$z = \frac{x}{s}, z_0 = \frac{x_0}{s_0}, s_1 = \frac{x_1}{s_1}$$

$$x = z * s, x_0 = z_0 * s_0, x_1 = z_1 * s_1$$

$$z = \frac{x_0 + t * (x_1 - x_0)}{s_0 + q * (s_1 - s_0)}$$

$$z = \frac{z_0 * s_0 + t * (z_1 * s_1 - z_0 * s_0)}{s_0 + q * (s_1 - s_0)}$$

由于

$$z = z_0 + t * (z_1 - z_0)$$

$$z_0 + t * (z_1 - z_0) = \frac{z_0 * s_0 + t * (z_1 * s_1 - z_0 * s_0)}{s_0 + q * (s_1 - s_0)}$$

化简可以得到(具体不写出来了，想办法把s 消掉，得到t 和q 的关系)

$$t * (q * z_0 + (1 - q) * z_1) = q * z_0$$

$$t = \frac{q * z_0}{q * z_0 + (1 - q) * z_1}$$

然后带回去，就可以消掉t

$$z = z_0 + t * (z_1 - z_0)$$

$$z = z_0 + \frac{q * z_0 * (z_1 - z_0)}{q * z_0 + (1 - q) * z_1}$$

化简后得到

$$z = \frac{1}{(1 - q) * \frac{1}{z_0} + q * \frac{1}{z_1}}$$

$$\frac{1}{z} = (1 - q) * \frac{1}{z_0} + q * \frac{1}{z_1}$$

继续回到关于z 和C 的式子

$$\frac{z - z_0}{z_1 - z_0} = \frac{C - C_0}{C_1 - C_0}$$

把z带进去

$$\frac{\frac{1}{(1-q)*\frac{1}{z_0}+q*\frac{1}{z_1}}-z_0}{z_1-z_0}=\frac{C-C_0}{C_1-C_0}$$

考虑将左边的式子化简(将分子先化简，然后凑出一个z_1-z_0，就可以将分母消掉)，得到

$$\frac{\frac{1}{(1-q)*\frac{1}{z_0}+q*\frac{1}{z_1}}-z_0}{z_1-z_0}=\frac{q*z_0}{q*z_0+(1-q)*z_1}$$

带回去

$$\frac{q*z_0}{q*z_0+(1-q)*z_1}=\frac{C-C_0}{C_1-C_0}$$

化简后，得到

$$C=\frac{(1-q)*C_0*z_1+q*C_1*z_0}{(1-q)*z_1+q*z_0}$$

为了方便插值，我们给分子和分母乘以一个$\frac{1}{z_0*z_1}$

$$C=\frac{\frac{1}{z_0*z_1}*((1-q)*C_0*z_1+q*C_1*z_0)}{\frac{1}{z_0*z_1}*((1-q)*z_1+q*z_0)}$$

$$C=\frac{(1-q)*\frac{C_0}{z_0}+q*\frac{C_1}{z_1}}{(1-q)*\frac{1}{z_0}+q*\frac{1}{z_1}}$$

$$C=z*((1-q)*\frac{C_0}{z_0}+q*\frac{C_1}{z_1})$$

根据这个形式，最后的C则是

$$C=z*(e_0*\frac{C_0}{z_0}+e_1*\frac{C_1}{z_1}+e_2*\frac{C_2}{z_2})$$

$$e_0+e_1+e_2=1$$

$$z=e_0*\frac{1}{z_0}+e_1*\frac{1}{z_1}+e_2*\frac{1}{z_2}$$

可能有人想到，既然知道了z，那么完全就可以找到在3D空间中对应的点的坐标了，那么为什么不直接对3D空间中的三角形进行插值。可以是可以，但是同样也要算e，并且

三角形的面积没有那么好求。而这个式子的e 则是平面上的三角形上的，因此面积直接叉积就可以了。

那么总结下过程，首先剔除，然后裁剪，然后重新将图元三角形化。然后对每个图元进行光栅化。对每个图元首先找到最小的包围盒，然后扫描整个包围盒的像素点，先判断是否在三角形内部，在的话先算出e 然后进行插值算出这个像素点的属性。

输出合并阶段

这个阶段还是比较简单的，只要实现一下深度测试和混合就好了。虽然样例代码中还没有做，但是并不难。只要在将像素输出到我们渲染的目标(图片或者屏幕啥的)的时候，判断一下像素的深度值来决定是否将像素写入，混合的话就是将这个像素位置的旧数据和新数据根据公式计算出新的像素信息即可。

代码

地址:https://github.com/LinkClinton/SoftwareGraphics

点击标题可以传送。这里还是不放什么和代码有关的东西了。

使用C#写的，不知道会不会有阅读障碍。注释有一点是英语的，可能因为太菜读不通加油脑补吧。当然也可以直接怼我这句话不通。

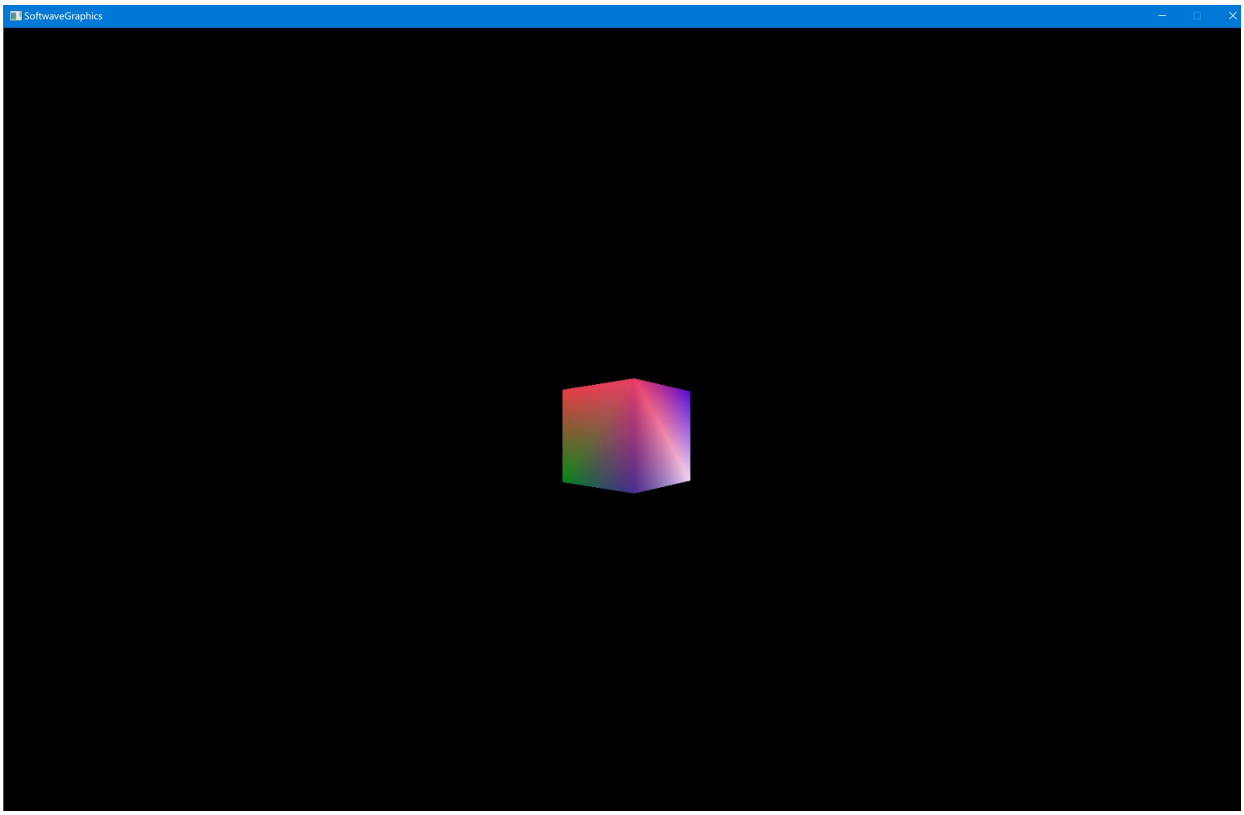
实现技巧

理论到实现并不总是那么容易的，相反还可能会比理论难。这里虽然实现还是没有那么难，但是作为过来人还是有一点小建议的。

可以考虑一个 `DrawCall` 类，用于存放渲染过程中要经过多个阶段的数据。主要是方便每个阶段传递数据。

同时也可以考虑一个 `UnitProperty` 类型，或者说统一的一个顶点属性类型。

演示效果，帧数感人。

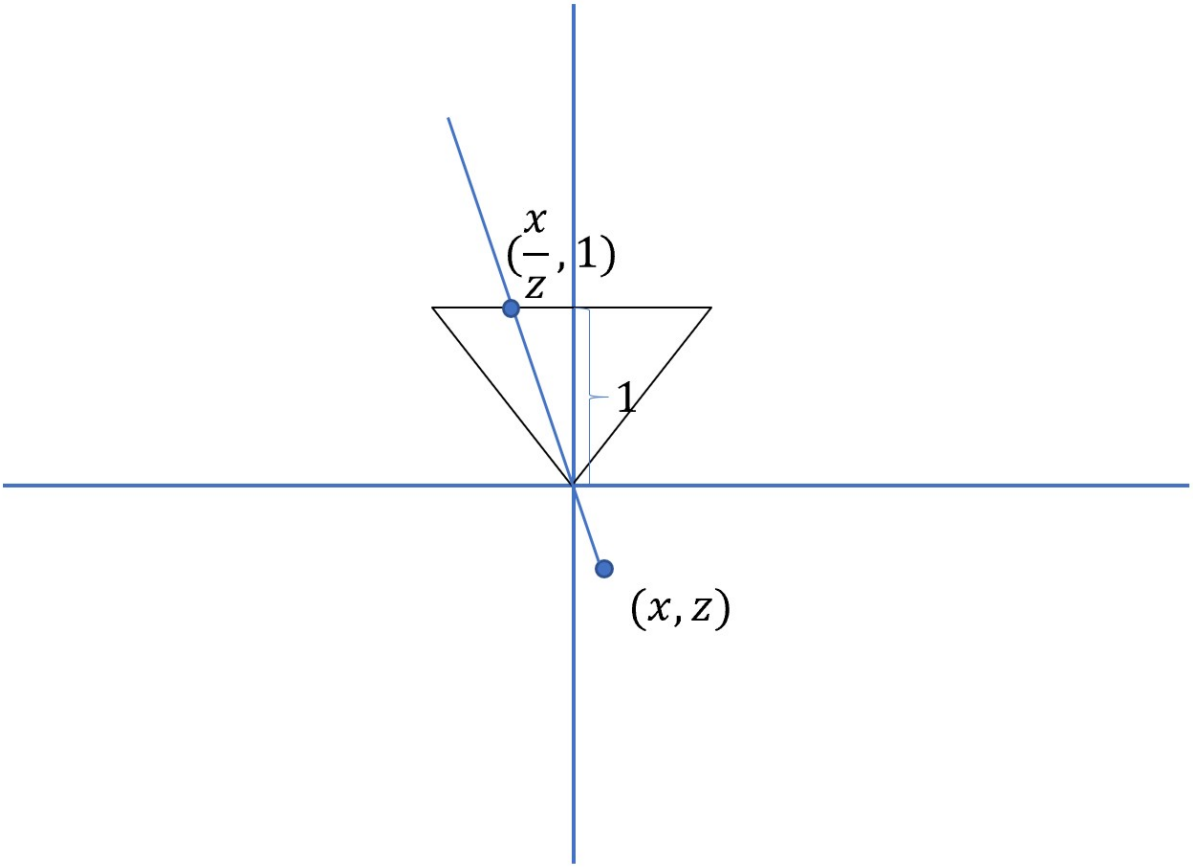


一些证明

关于w 的

但是需要注意的是，这里并不能直接取判断其是否在$[-1,1]$ 范围或者$[0,1]$ 内，而是应该判断x 和y 是否在$[-w,w]$ 内，z 是否在$[0,w]$ 内。至于为什么，可以自己思考下。提示是关于w 的正负性。

注意我们的问题是什么，其实就是判断是否在面的某一边，而不是判断投影后的内容是否在面的某一边。这样就会导致如下错误。点不在面的右边，但是投影后在，这样的话，不应该算它在面的右边。



关于除以0 的

可能有人会觉得要是除以0 怎么办，答案是不会出现这样的情况。可以思考下。提示是一条边中的两个端点中，最多只有一个在面上。

考虑出现裁边的情况，只有在一个点在平截头体内部(包括面上)，一个在外部才可能出现。而对于$(e.x - s.x) + (e.w - s.w)$ 来说，我们将其整理下，得到

$$(e.x + e.w) - (s.x + s.w)$$

因为其中一个在外面，一个在里面，那么也就是括号内的值有一个为非负，有一个为负。那么，显然他们没有办法相减为0 .

结语

感谢阅读。

引用

- 非彩色图片出自*Introduction to 3D Game Programming with Directx 12*
- 其中一种包围盒的图片出自: https://www.scratchapixel.com
- 安利一个网站 : https://www.scratchapixel.com