SoftwareGraphics

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的读者。主要内容是介绍一下在整个渲染管线中会使用的算法，以及如何实现它。

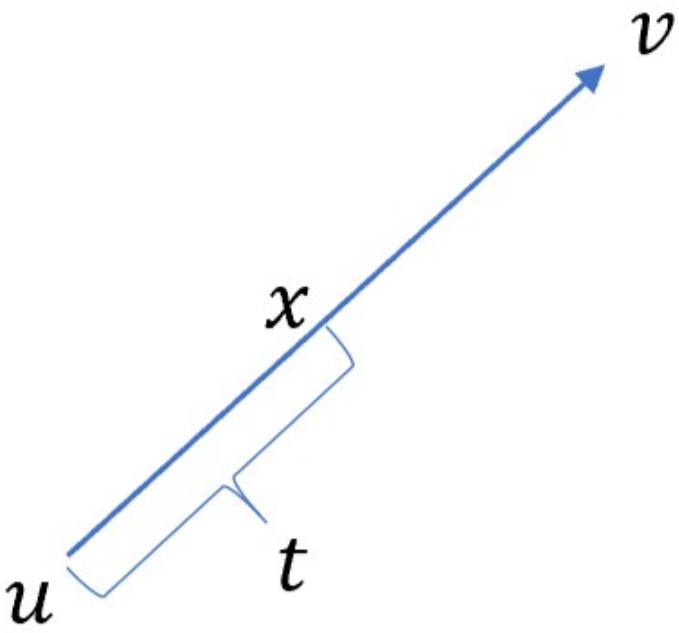
因此阅读之前建议对渲染管线以及着色器有一定的了解，以及有一定的代码能力，还有一小点的线代知识。

当然，如果不了解有也没有关系，我会尽可能的介绍。

前置技能介绍

主要是介绍一些简单的前置内容，以便于对图形渲染管线不了解的读者阅读。

插值



这是一个很简单的概念，请先参见图片和式子。

$$x = u + (v - u) * t$$

$$x = (1 - t) * u + v * t$$

这两个式子是等价的，在已知u, v, t 的情况下我们可以计算出x 的值。并且是可以推广到更高维度。

这里给出一个例子。

$$u = (5.0, 4.0)$$

$$v = (10.0, 8.0)$$

$$t = 0.6$$

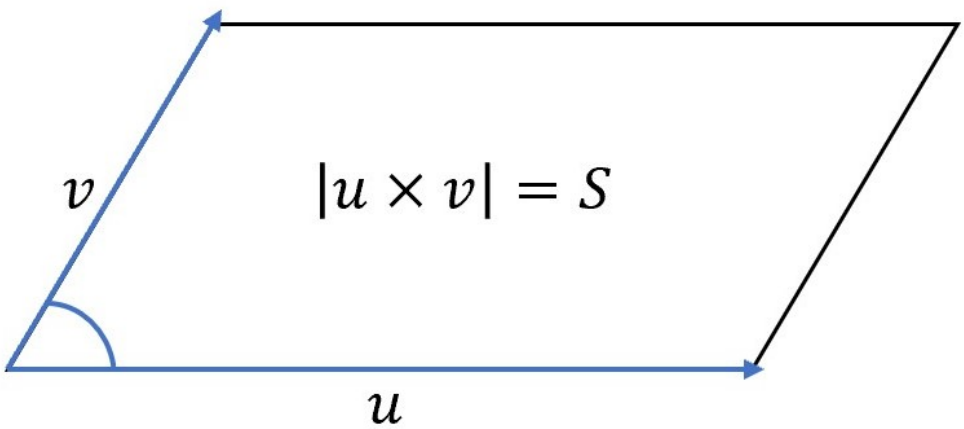
那么

$$x = (1 - t) * u + v * t$$

$$x = (8.0, 6.4)$$

叉乘

这里只考虑二维向量的叉乘，其几何意义是这两个向量构成的平行四边形的面积。稍微有一点不同的是，运算顺序的不同，其结果的正负性不同。



图片中，$u \times v = S$ 而$v \times u = -S$ 。其定义式为:

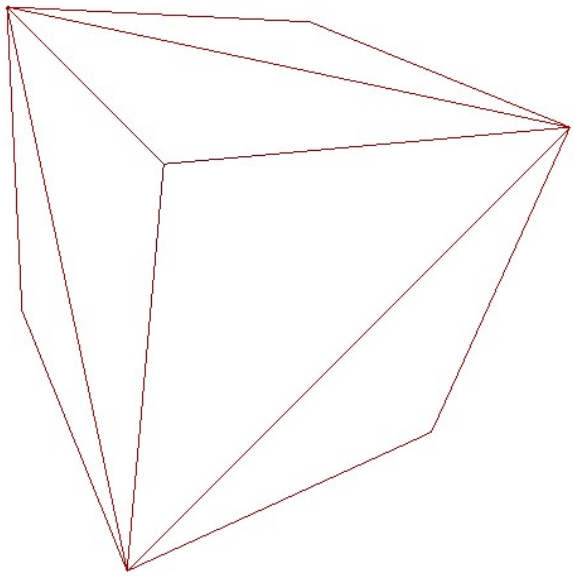
$$u \times v = u.x * v.y - v.x * u.y = |u| * |v| * \sin(u, v)$$

有兴趣的可以自己去证明下。实在找不到合适的方法表达u, v 的夹角了。

图元

在介绍什么是图元之前，我们先介绍下什么是顶点。所谓的顶点，就是空间中的点，是构成图元的基本类型。而图元，我们可以认为就是点和边的集合，由点和边构成的图形。那么显然，三角形属于图元的一种，这也是我们需要主要讨论的类型。在之后的内容中，如果没有特别说明，我们一般都是直接用三角形代替了图元。

那么图元的作用是什么？它的作用是构成我们渲染的立体图形。例如一个立方体，我们知道他有8个顶点，6个面。那么如果想要构成一个立方体，我们便会使用图元来构建立方体的每个面，即两个三角形构成一个面，然后6个面构成一个立方体。而对于更复杂的物体，通常我们也是使用三角形来构成面。然后用面构成体。例如这样的图形。



三角形在空间中是有两个面的，为了区分这两个面，我们定义了正反面概念。

- 正面：三角形顶点的排列顺序为顺时针。
- 反面：三角形顶点的排列顺序为逆时针。

这里不同的API对于正反面的定义不同，需要注意。

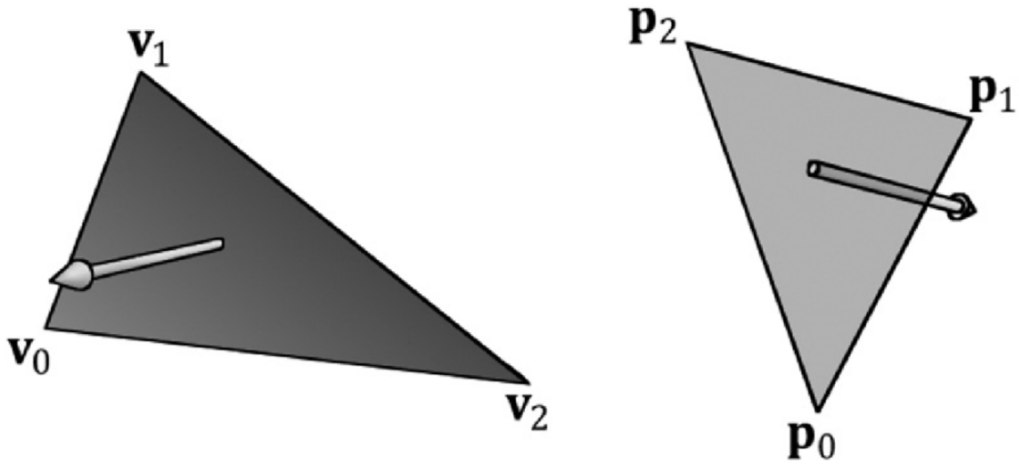
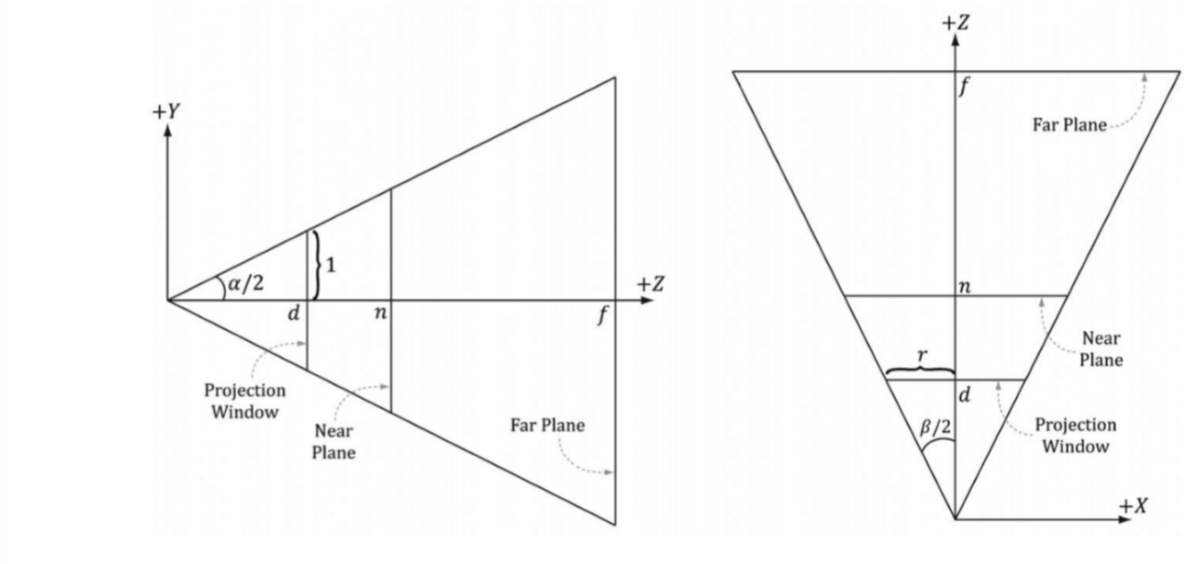


Figure 5.30. The left triangle is front-facing from our viewpoint, and the right triangle is back-facing from our viewpoint.

左边的三角形是正面，右边的是反面。因为v_0, v_1, v_2 的顺序排列是顺时针，而p_0, p_1, p_2 的排列是逆时针。

投影

所谓投影就是将我们可以看到的物体投影到2D平面内。

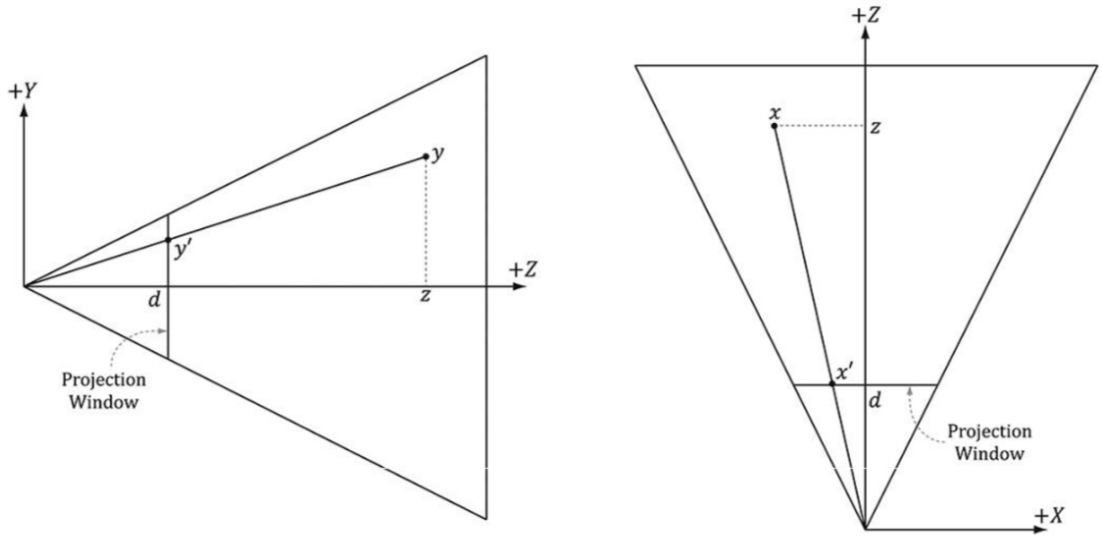


上图是空间中的一个截面，d 是我们所要投影的平面，n 和f 两个平面以及和三角形的两条边构成的范围则是我们的可视范围。

当然这只是一个截图，在空间中，这个三角形其实是一个四棱锥，而我们的可视范围其实是一个棱台一样的东西，我们称之为**平截头体**。

在可视范围内的点，最后都会被投影到d 所在的面内。这里为什么我不说平面，目的是要和无限大的面区分一下，被投影的面是一个高度为2 ，宽度为$2r$ 的面。

r 是什么？r 是我们最后要呈现物体的图片或者窗口的长度和宽度的比值。也就是说，其实d 这个面是我们的图片或者窗口按照比例缩放后的样子。至于为什么要按照比例缩放，可以思考下[1] 。



$$\frac{x'}{d}=\frac{x}{z}\Rightarrow x'=\frac{xd}{z}=\frac{x\cot(\alpha/2)}{z}=\frac{x}{z\tan(\alpha/2)}$$

$$\frac{y'}{d}=\frac{y}{z}\Rightarrow y'=\frac{yd}{z}=\frac{y\cot(\alpha/2)}{z}=\frac{y}{z\tan(\alpha/2)}$$

为了方便处理，我们投影的时候，其实还是会对d 这个平面进行一下缩放，将d 平面缩放成一个宽度为2 高度也为2 的面。同样也可以思考下为什么[2] 。

因此，我们的x' 还需要处理r 才能得到我们最后想要的x 坐标。

最后，我们使用一个矩阵来帮助我们进行运算。

$$(x,y,z,1)\times\begin{bmatrix}\frac{1}{r\tan(\alpha/2)} & 0 & 0 & 0\\ 0 & \frac{1}{\tan(\alpha/2)} & 0 & 0\\ 0 & 0 & A & 1\\ 0 & 0 & B & 0\end{bmatrix}=$$

$$(\frac{x}{r\tan(\alpha/2)},\frac{y}{\tan(\alpha/2)},Az+B,z)$$

细心的会发现，我们少除以了一个z，并且我们发现坐标的w分量就是我们要除以的z，因此最后我们要对坐标除以一个w才算完成转换，这一步我们叫做透视除法。而对于z分量，由于这不是我们讨论的重点，我们只需要知道最后满足如下性质:

$$0 \leq A + \frac{B}{z} \leq 1$$

$$A + \frac{B}{n} = 0$$

$$A + \frac{B}{f} = 1$$

也就是说，对于在平截投影内部的点，经过矩阵变换后，其分量满足这样的式子：

$$-w \leq x \leq w$$

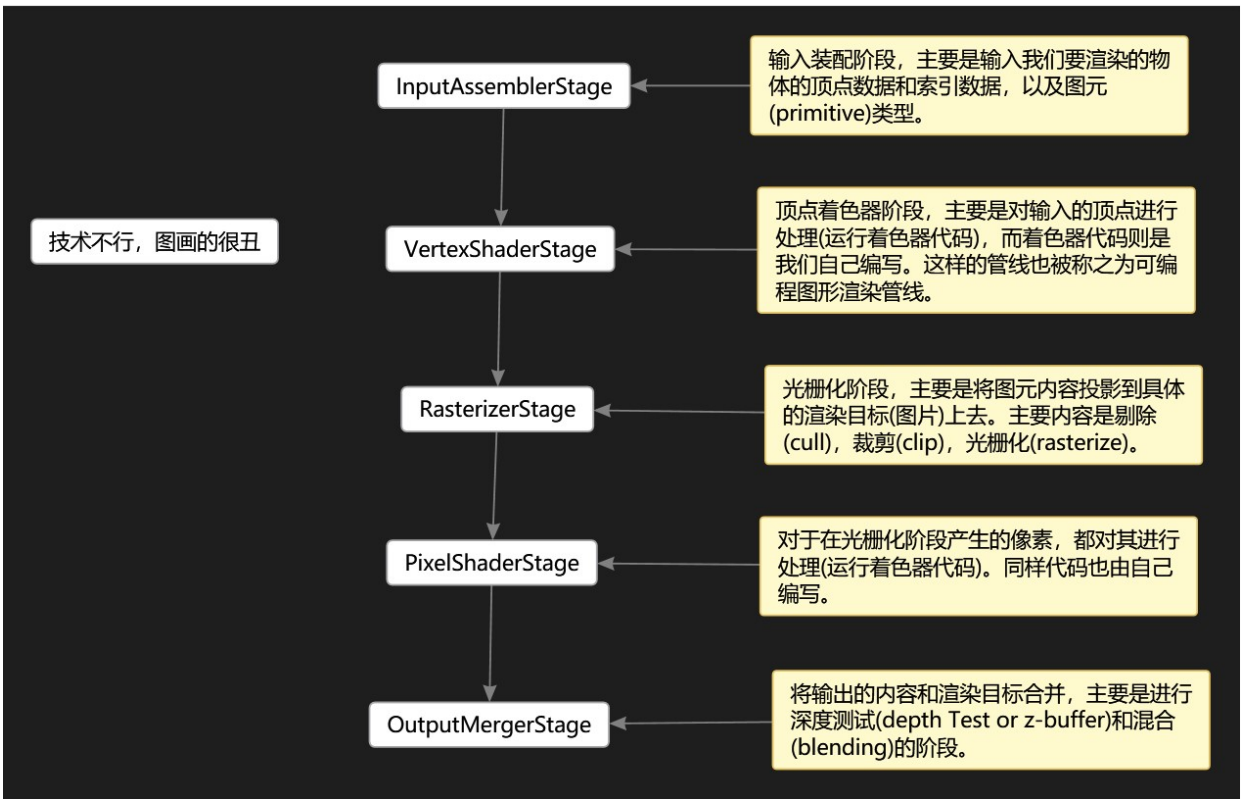
$$-w \leq y \leq w$$

$$0 \leq z \leq w$$

着色器

着色器其实是一个很简单的东西，我们只需要知道其就是一段在GPU上运行的程序即可。

图形渲染管线



这里列出了一个简单的管线流程以及管线中每个阶段的作用。目前的版本中，我们将管线分为了五个阶段，分别是输入装配阶段，顶点着色器阶段，光栅化阶段，像素着色器阶段，输出合并阶段。接下来依次简单介绍下各个阶段。

输入装配阶段，这个阶段主要是用来设置我们要渲染的物体的数据的。例如物体的顶点数据，以及图元数据，以及构成物体的图元类型(并不是只能用三角形构成物体，但是我们通常使用三角形，因此之后将会使用三角形代替图元)。

顶点着色器阶段，就如同名字一样，这一个阶段主要是关于顶点着色器的。在这个阶段，我们将会为物体的每个顶点运行一遍顶点着色器代码，换言之就是将顶点作为参数运行着色器代码。着色器代码是由我们自己编写的，因此我们可以在这个阶段对顶点做任何我们想做的事情，例如将顶点进行平移操作。

光栅化阶段，物体在空间中投影到2D平面(由像素构成)后，将会覆盖一部分区域，而光栅化就是找出哪些像素被投影后的图形覆盖。

像素着色器阶段，和顶点着色器阶段差不多，但是对象不再是顶点，而是在光栅化阶段找出的像素。

输出合并阶段，我们在光栅化阶段得到了一些像素，这些像素在像素着色器阶段进行处理后，将会实际输出到图片中或者屏幕中去。而在实际输出之前，我们还需要对这些像素进行深度测试(depth-test or z-buffer)以及混合(blending)。由于这部分本文不会涉及，因此不再展开。

实现原理

这是一个简单的软渲染实现，目前也并没有去很完整的去完善一些功能，因此有一些功能并没有支持，即便只是需要改动一下一点代码就能支持。使用的算法方面，只能作为一个参考，因为硬件的特性，GPU未必就是按照这样的算法进行的，但是原理是一致的。

输入装配阶段

对于物体的顶点，单纯的三维坐标可能并不满足。例如我们想要在物体的表面贴上图片的话，我们还需要为顶点加上纹理坐标。例如我们想要进行光照的话，就要为顶点附加上法线。自然而然的，我们可以为定义一个这样的顶点类型，既包含三维坐标，纹理坐标，法线等等数据。

但是有一些时候，可能我们用不到纹理坐标或者法线。那么这将是一种浪费，浪费空间去存储用不着的数据，浪费时间去处理用不着的数据。为了避免这样的浪费，我们可以在设置顶点数据的时候，告诉管线我们的顶点数据的组成，例如我们的顶点数据是由一

个三维坐标加上一个纹理坐标构成，那么我们的管线处理的时候，就只会处理和这些数据有关的数据，而不会去处理法线。

顶点着色器阶段

这一阶段，我们可以先对物体的每个顶点运行一遍着色器代码。然后将三角形(图元)构建出来。

虽然简单，但是这里还有一些可以注意的小地方。

- 记住每个顶点运行完一遍着色器代码后，需要进行透视除法。
- 没必要完全处理所有的顶点，只需要将这次渲染中涉及到的顶点进行处理。

光栅化阶段介绍

光栅化阶段是整个流程的核心，基本上大部分运算都在这个阶段进行。光栅化阶段和像素着色器阶段的效率对整个流程的效率影响很大，因此一个快速的进行光栅化是优化效率的一个好思路。

而在这一个阶段，所需要进行的处理也是相当多的。大致可以分为这几个：剔除，裁剪，光栅化。

剔除

对于一些封闭的物体，例如立方体，其内部的三角形我们是看不见的。而我们在处理的过程中，我们仍然会处理这些内部的三角形。为了避免这样的情况，我们就需要找出哪些三角形是内部的，然后将其移除，这样我们就能够减少我们需要处理的三角形数量。这样的过程我们叫做剔除。

由于一个一个去寻找这个三角形是否能够看见可能带来的损耗会比收益更大，因此我们的剔除是根据三角形的正反面决定的。我们可以选择剔除正面的三角形，也可以选择剔除反面，或者都不剔除。

也就是说，在剔除过程中，我们只需要对每个三角形判断一下正反面即可。那么如何判断正反面？在空间中判断可能会比较麻烦，因此我们可以考虑在投影后的平面内考虑。

我们知道，三角形的三个点投影到平面后也是对应的三个点，我们只需要考虑这三个点的排列顺序是顺逆时针，就可以判断出这个三角形到底是正面的还是反面的。而平面内，判断点排列顺序的顺逆的方法，参见之前讲的叉乘即可解决。

裁剪

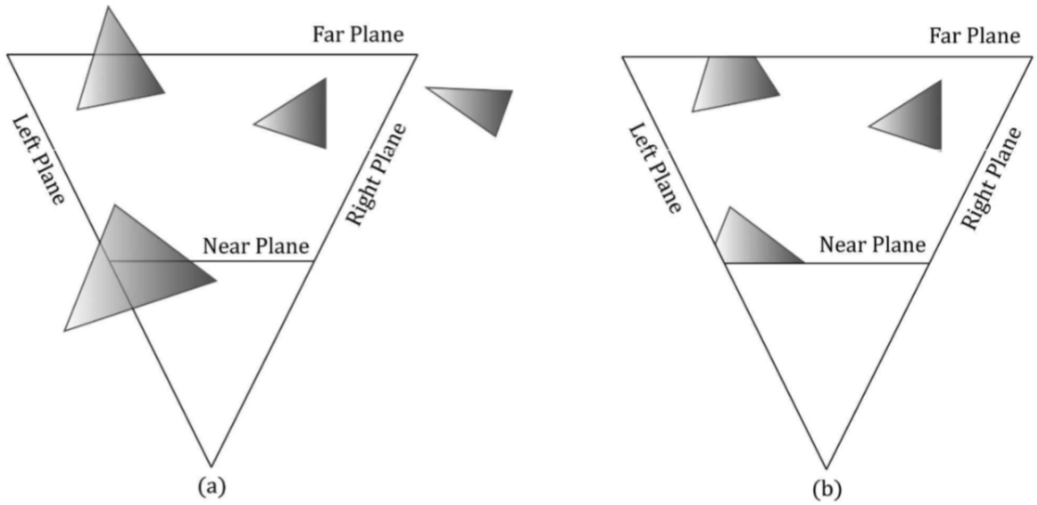


Figure 5.27. (a) Before clipping. (b) After clipping.

有些时候，我们并没有办法看到一整个物体，而是只能看到一部分物体，那么如果去处理整个物体显然是没有必要的。因此我们可以先将我们的物体进行裁剪，将不可视部分去除，只留下可视部分。图中的那个梯形就是我们的可视范围，左边是没有进行裁剪的情况，右边是进行裁剪后留下的物体。

Sutherland-Hodgeman算法，一种裁减算法。其主要思路是这样的：

枚举平截头体的每个面，然后判断三角形是否和面相交，如果相交，那么就将多出的部分截掉。

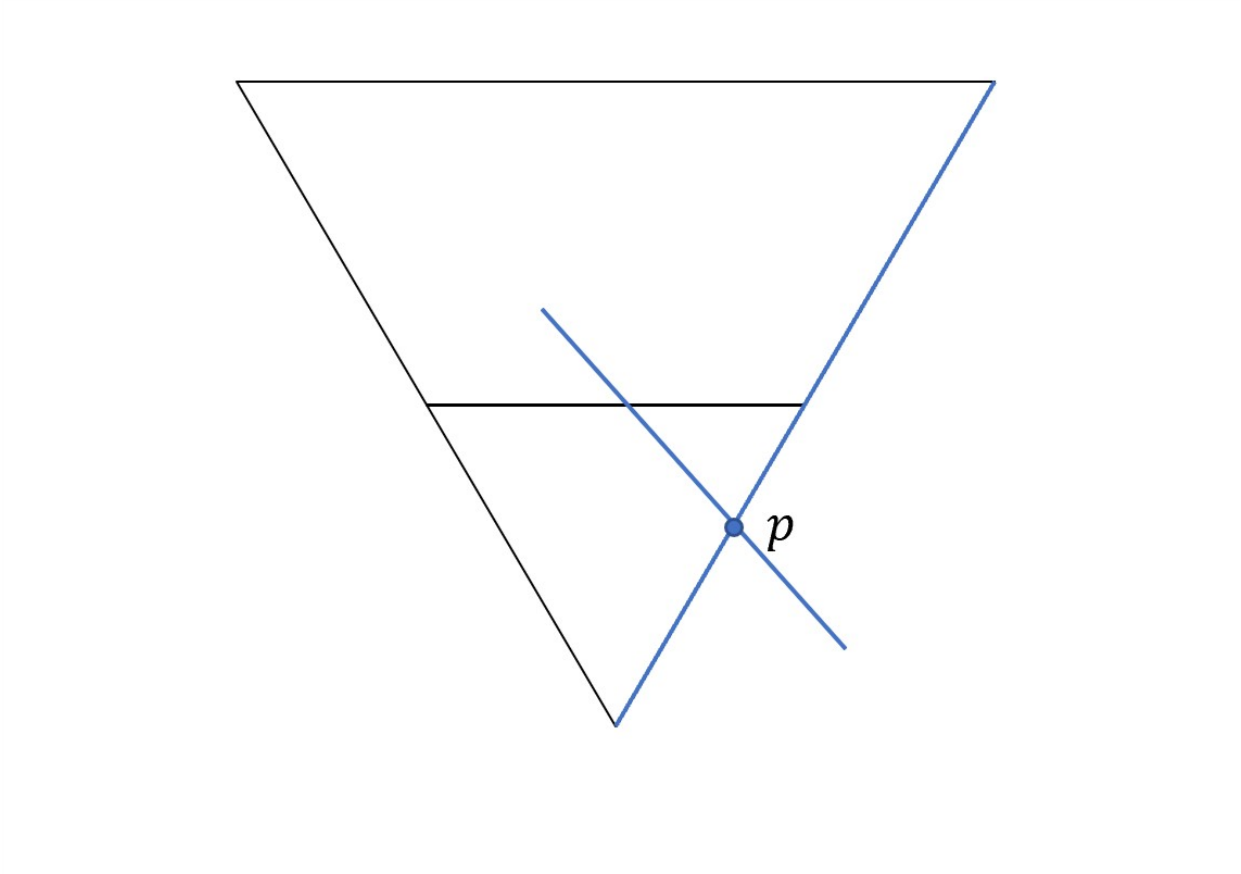
而对于我们枚举的面，我们依次枚举三角形的边：

- 对于三角形的一条边，如果起始顶点和结束顶点都在平截头体内部的话，那么我们就将起始顶点加入到新的顶点数组里面去。
- 对于三角形的一条边，如果起始顶点和结束顶点都不在平截头体内部的话，那么我们什么也不做。
- 对于三角形的一条边，如果起始顶点在平截头体内部，而结束顶点不在的话，那么我们就将起始顶点和这条边和面的交点加入到新的顶点数组中去。
- 对于三角形的一条边，如果起始顶点不在平截头体内部，而结束顶点在内部的话，那么我们就将边和面的交点加入到新的顶点数组中去。

那么新的顶点数组就将构成新的图元，我们用这新的图元去代替原本的图元，在枚举下一个面的时候使用。最后结束的时候剩下的图元，就是被裁剪后的图元。

这里有几个问题需要解决，或者说解释。

- 裁剪出的未必是三角形。
- 边和面的交点可能还是在平截头体外(下图中点p 即为例子)。
- 如何判断点是否在平截头体内部以及如何找出面和边的交点。



第一个问题在图中就可以找出一个例子出来，这个待会我们会有解决方法。而第二个问题，能够保证最后的图元没有在平截头体外面的部分，至于为什么可以思考下，在文章最后会留有解释[3] 。

如何判断点在平截头体内，这其实是一个非常简单的问题。在讲述投影的时候，我们就已经解决了。但是需要注意的是，我们不能去判断**经过透视除法**后的点的坐标范围是否满足这样的式子:

$$-1 \leq x \leq 1$$

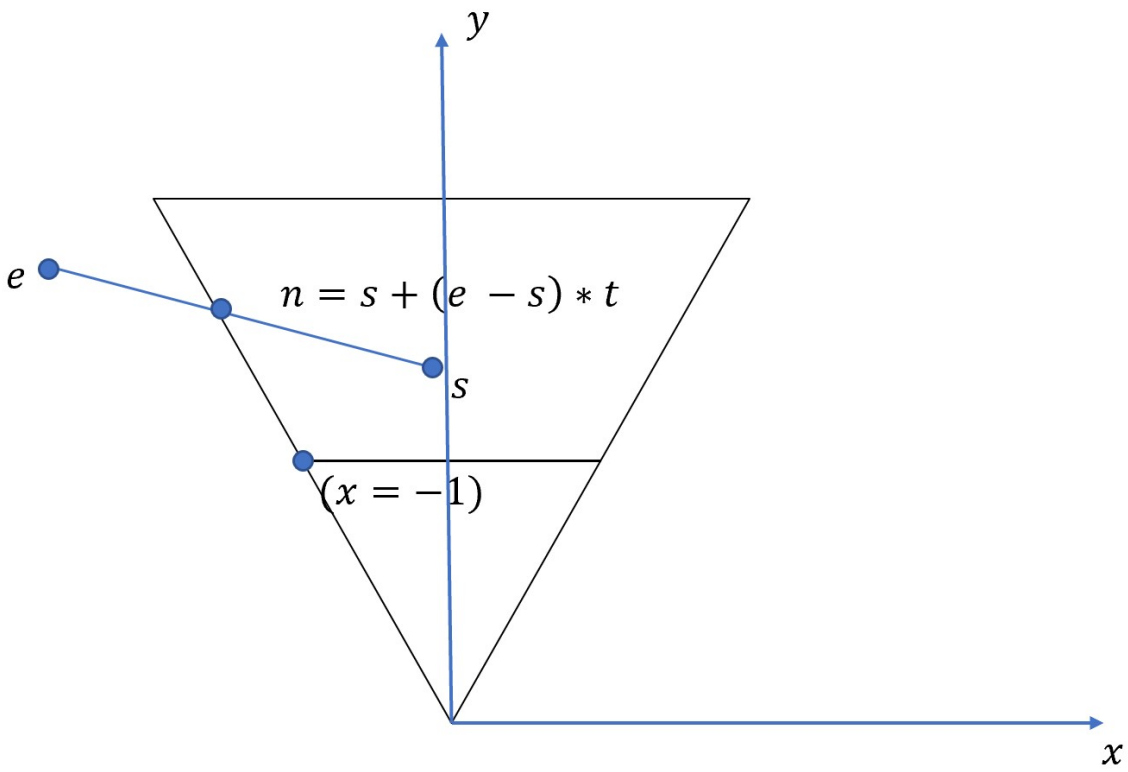
$$-1 \leq y \leq 1$$

$$0 \leq z \leq 1$$

可以思考下，文章末尾解释为什么[4] 。

如何找出边和面的交点?

边和面的交点还是在平截头体内部，但是有一点不同的是，他们是在面上。这意味着他们的投影刚好就是临界值。这里我们以平截头体的左边的面为例子。



严谨来说图中的$x=-w$，但是这里为了表达清楚，我们使用的是$x=-1$。可以思考下为什么，不建议思考太久[5]。

我们设边的起始顶点为s，结束点为e，那么边和面的交点$n=s+(e-s)*t$，n在左边的面上，那么n进行**透视除法**后的点的x分量必然为-1。也就是说:

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

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

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

那么我们就能够计算出t，从而计算出n了。

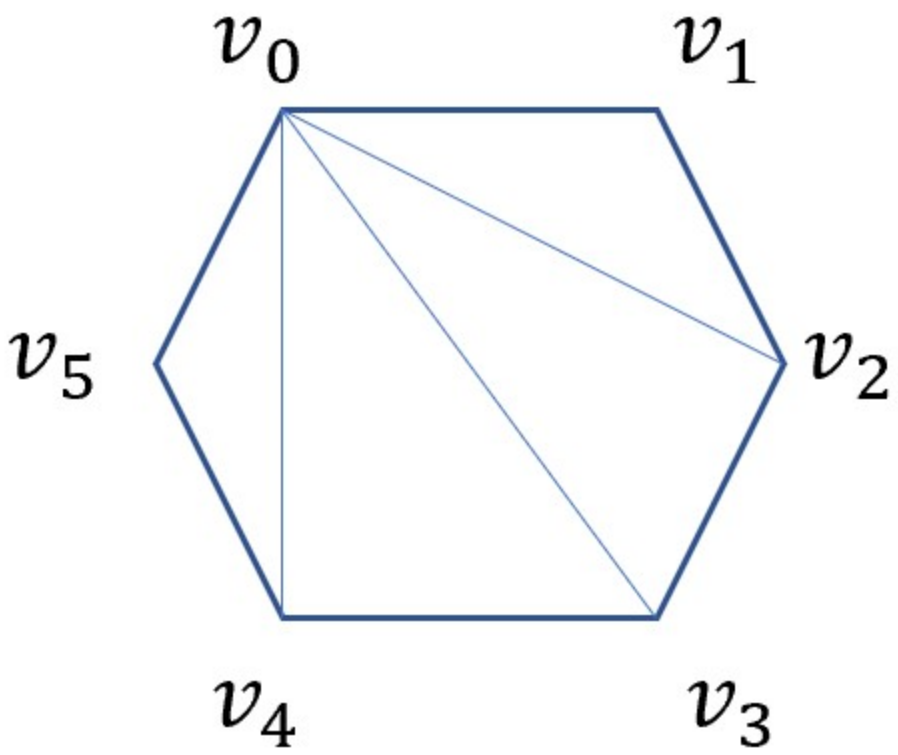
$$n=s+(e-s)*t$$

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

但是这里有一个小细节需要注意下，当我们计算出n之后，不要太高兴了，以至于忘记我们还要对n进行透视除法。除此之外，还有一个问题，除以0的情况，即为什么$(e.x-s.x)+(e.w-s.w)\neq 0$，思考一下[6]。

三角剖分

重新回到我们的第一个问题，裁剪出来的未必是三角形。因此我们要重新将图元三角形化，这个操作非常的简单。

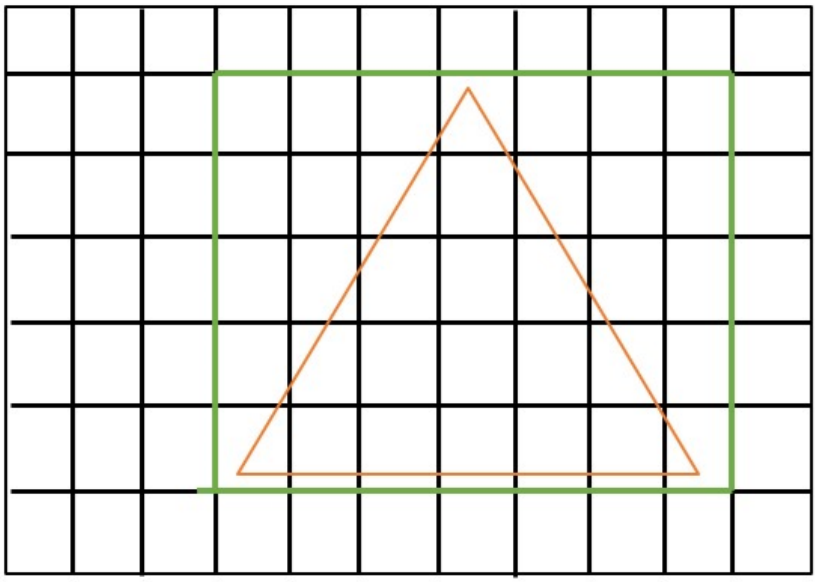


选定一个基准点v_0，然后按照图片的方式剖分一下就好了。在这里，我们还可以把三角形的顶点排列顺序统一下，可以全部将其改为顺时针也可以全部将其改为逆时针，这样的话可以减少后面过程中对三角形顺逆时针的讨论。

光栅化

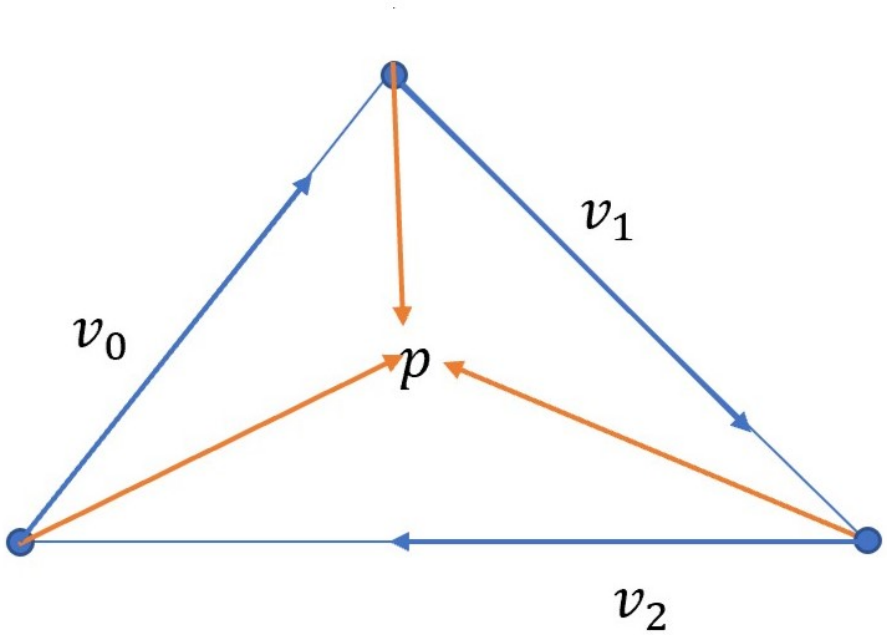
经过上述流程后，我们得到了一组在平面上的三角形。在之前介绍光栅化的时候，提到物体会占据一部分平面内的区域，这些三角形所占的区域就是物体投影后所占的区域。由于物体投影后的区域是不规则的，因此我们将其拆散开来，只讨论三角形的情况，最后将得到的像素输出即可。

这样做，对于三角形重叠部分的像素来说，他们被计算了两次，我们输出的时候也会输出两个同样位置的像素，这个时候我们就需要考虑使用深度测试来解决像素的先后问题。



对于每个三角形，我们可以先找出一个最小的能够把三角形包围的包围盒。即图中绿色的盒子。然后我们对于绿色的盒子里面的像素，依次枚举，判断**像素的中心**是否在三角形内部，如果在，那么我们就认为这个像素被三角形覆盖，如果不在，那么我们就认为没有被覆盖。

对于在三角形内部的点来说，当点在三角形的三条边的某一侧的时候，就成立。



图中p 点和三角形顶点构成的向量分别在v_0, v_1, v_2 的某右侧。很显然也就是说，在v_0, v_1, v_2 顺时针排列的时候，只要p 点和三角形顶点构成的向量分别在v_0, v_1, v_2 的右

边，那么p 点就在三角形内部。当然，如果v_0, v_1, v_2 是逆时针排列的话，那么p 点和三角形顶点构成的向量就需要分别在v_0, v_1, v_2 的左边。至于如何判断两个向量的相对位置的问题，参考叉乘即可解决。

之前我们提到过，顶点不止只有位置坐标，还可以有一些其他的属性。同样的，对于我们生成的像素，也可以有这些属性。而这些属性则是由顶点的属性生成。如何生成？我们使用插值来通过顶点的属性来计算出像素的属性。

$$C_p = C_{v_0} * \lambda_0 + C_{v_1} * \lambda_1 + C_{v_2} * \lambda_2$$

$$\lambda_0 + \lambda_1 + \lambda_2 = 1$$

我们用这个式子来计算出某个像素的属性。其中C_v 表示的是向量起点的那个顶点的属性。λ 则目前不知道，但是他们的和为1 。

这其实就是一个插值的过程，不过他并不是单个元素而已。当其中一个λ 为0 的时候，就和我们介绍的插值的形式是一样的了。

那么我们要如何求解$\lambda_0, \lambda_1, \lambda_2$ 。我们可以联想到一个东西，重心坐标。

$$(\lambda_0 + \lambda_1 \ldots + \lambda_{n-1}) * p = \lambda_0 * v_0 + \lambda_1 * v_1 \ldots + \lambda_{n-1} * v_{n-1}$$

而三角形的重心坐标又叫做面积坐标，简单来说:

$$\lambda_0 = \frac{S_{\Delta v_1 v_2 p}}{S_{\Delta v_0 v_1 v_2}}$$

$$\lambda_1 = \frac{S_{\Delta v_2 v_0 p}}{S_{\Delta v_0 v_1 v_2}}$$

$$\lambda_2 = \frac{S_{\Delta v_0 v_1 p}}{S_{\Delta v_0 v_1 v_2}}$$

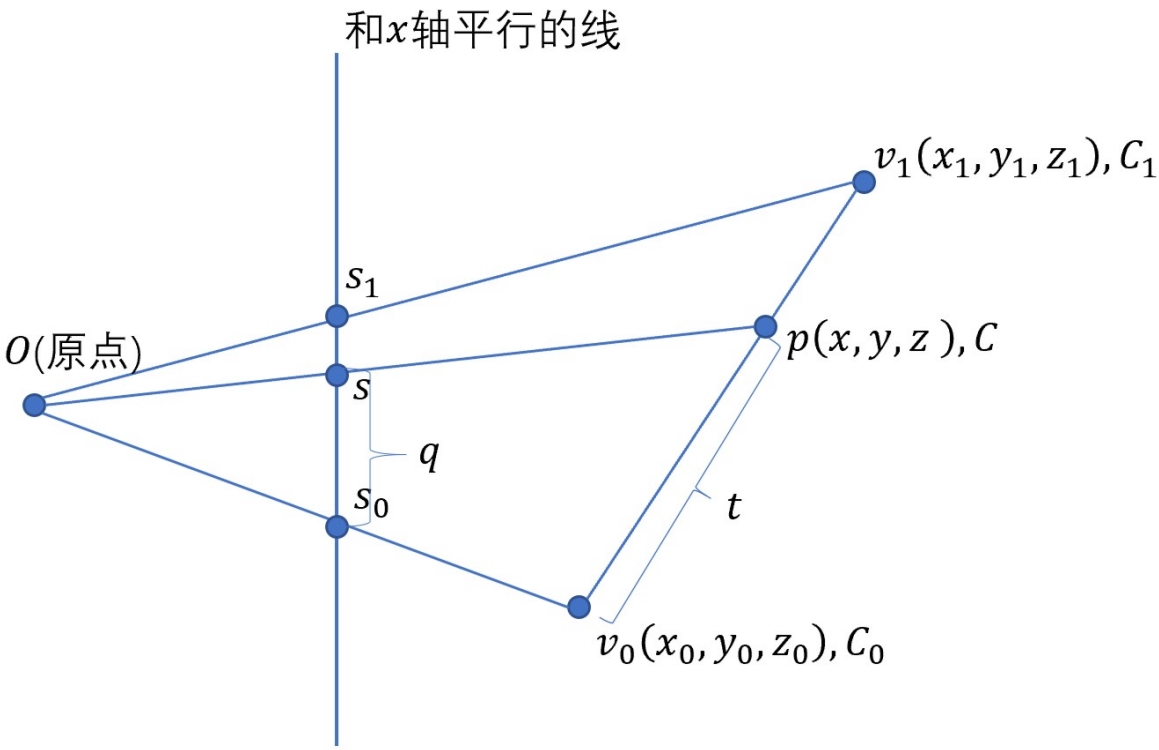
证明可以去翻阅维基百科。

目前我们提到过两种三角形，第一种是在空间中被投影的三角形。另外一种则是投影到平面后的三角形。那么我们要对哪种三角形去进行插值计算呢？答案是第一种，被投影的三角形。可以思考下为什么，文章末有解释[7] 。

对于任意一个在投影后的三角形内部的像素，我们肯定能够在被投影的三角形中找到一个位置和这个像素对应。那么我们可以找到这个对应的位置，然后使用上面的公式计算出λ 就可以计算出这个像素的属性了。

但是实际上我们没必要这么做，即便找到对应的位置后，还是需要计算面积求出λ ，而空间中的三角形的面积并没有平面内的那么好求。

我们设空间中线段上的点$p(x,y,z)$，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$p$ 的属性为C，v_0 的属性为C_0，v_1 的属性为C_1。



那么我们可以得到这个式子:

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

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。

刚好有一个性质我们可以用到。

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

通过这两个式子我们可以求解C。

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

那么我们最后的C 则可以依据这个式子扩展。

$$C=z*(\lambda_0*\frac{C_0}{z_0}+\lambda_1*\frac{C_1}{z_1}+\lambda_2*\frac{C_2}{z_2})$$

$$\lambda_0 + \lambda_1 + \lambda_2 = 1$$

$$z = \lambda_0 * \frac{1}{z_0} + \lambda_1 * \frac{1}{z_1} + \lambda_2 * \frac{1}{z_2}$$

推导过程在文章末会写下来[8] 。

最后总结下这个阶段的流程，首先进行剔除，然后裁剪，结束后将图元重新三角化。然后对每个三角形进行光栅化操作，找出三角形的包围盒，扫描包围盒内部的每个像素是否在三角形内部，在的话计算出λ ，然后依据式子进行插值计算出结果。

输出合并阶段

输出合并阶段并不是我们这篇文章的重点，因此不会花过多的笔墨去描述。

如果对输出合并阶段较为了解的话，其实现是非常简单的，混合和深度测试等等功能，基本上只是需要注意一下架构上的设计。例如深度测试，需要支持提供设置深度缓冲来存储每个像素的深度值，并且提供接口支持更换深度比较函数，来决定到底是深度值大的最后渲染到图片或者窗口，还是深度值小的。

而对于本文章的程序来说，目前并没有对输出合并阶段有什么实现，包括深度测试以及混合都没有实现，有兴趣的可以自己去实现下。

代码

请点击这里: https://github.com/LinkClinton/SoftwareGraphics

整个软渲染由C#编写，主要是算法和框架的设计，因此可能对一些细节优化以及细节处理可能还是存在一些问题。如果发现问题，欢迎提出来。

代码内部有一部分注释，如果语法有问题，欢迎提出来。

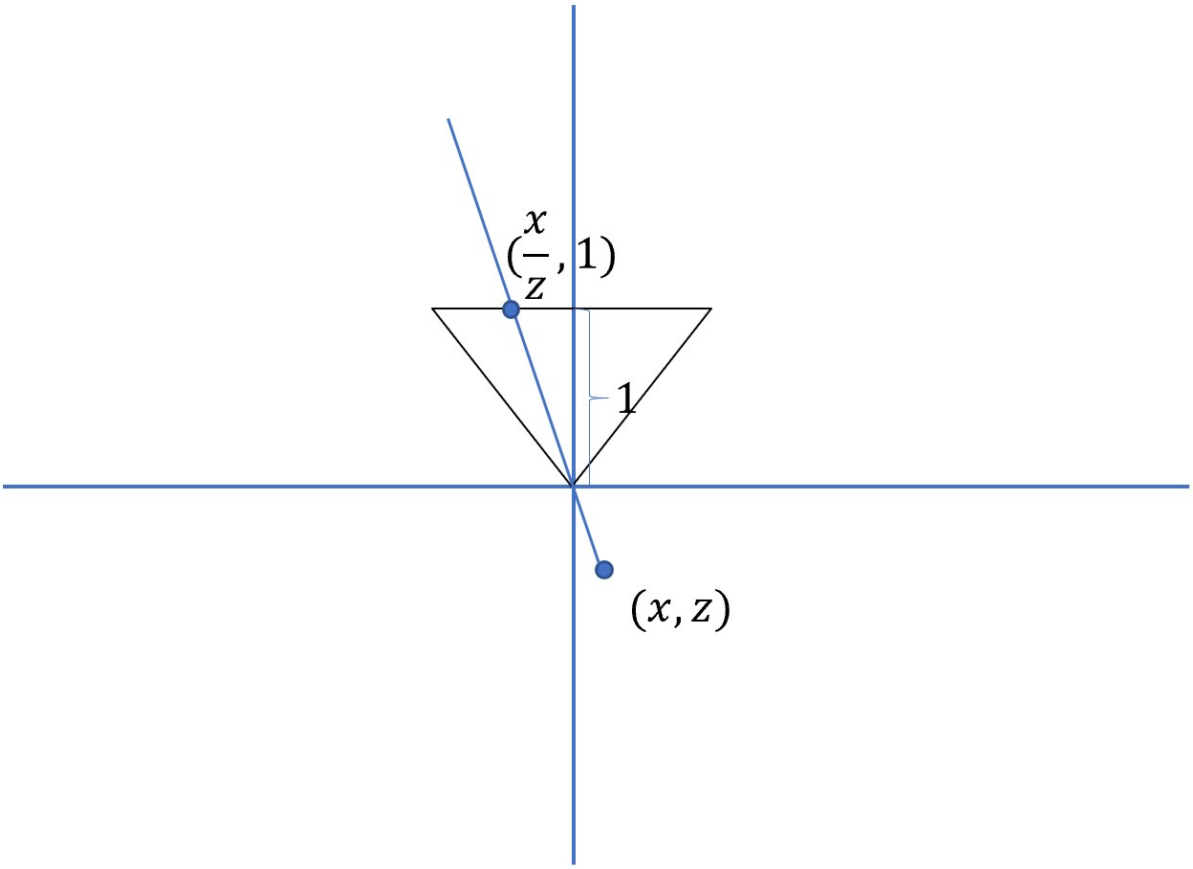
文章解释

$[1]$: 按比例缩放的目的是解决当物体渲染到图片或者窗口后因为比例不同导致渲染结果被缩放的问题。例如如果不按照比例缩放，那么一个正方形可能就会被渲染成长方形。正方形在投影平面内是正方形，但是当光栅化的时候，投影平面的比例和图片比例不同，那么正方形就会被缩放成长方形。

$[2]$: 在$[1]$ 中我们提到要按照比例缩放，但是这样的话，就代表我们必须知道这个比例是多少。为了方便处理，我们可以先缩放一下，然后最后光栅化的时候又缩放，最后两者互相抵消，那么我们就在后续处理中的时候不需要关系缩放的比例问题了。最后经过透

视除法的顶点所在的坐标系我们就叫做标准化设备坐标系，在这个坐标系中，我们不需要再去考虑任何和最后渲染的图片或者窗口有关的问题。

[3] : 裁剪边新产生的点虽然可能在视截体外部，但是绝对不会在已经经过裁剪处理的面外(面外指的是没有视截体在的那一边)，因为产生这样的情况的话，必须得有一个点在已经经过裁剪处理的面外。我们对每个面进行裁剪处理，每次都能够保证面外的点被裁剪掉，且产生的点即便在视截体外面也是没有经过裁剪的面外，那么因此最后一个面被裁剪后，就不会存在点在面外了。



[4] : 我们并不能直接判断是否在$[-1, 1]$ 范围或者$[0, 1]$ 范围内，而是应该对于x, y 分量，判断是否在$[-w, w]$ 范围内，对于z 判断是否在$[0, w]$ 范围内。参见图中的例子，应该能够想出来。

[5] : 绘图错误。

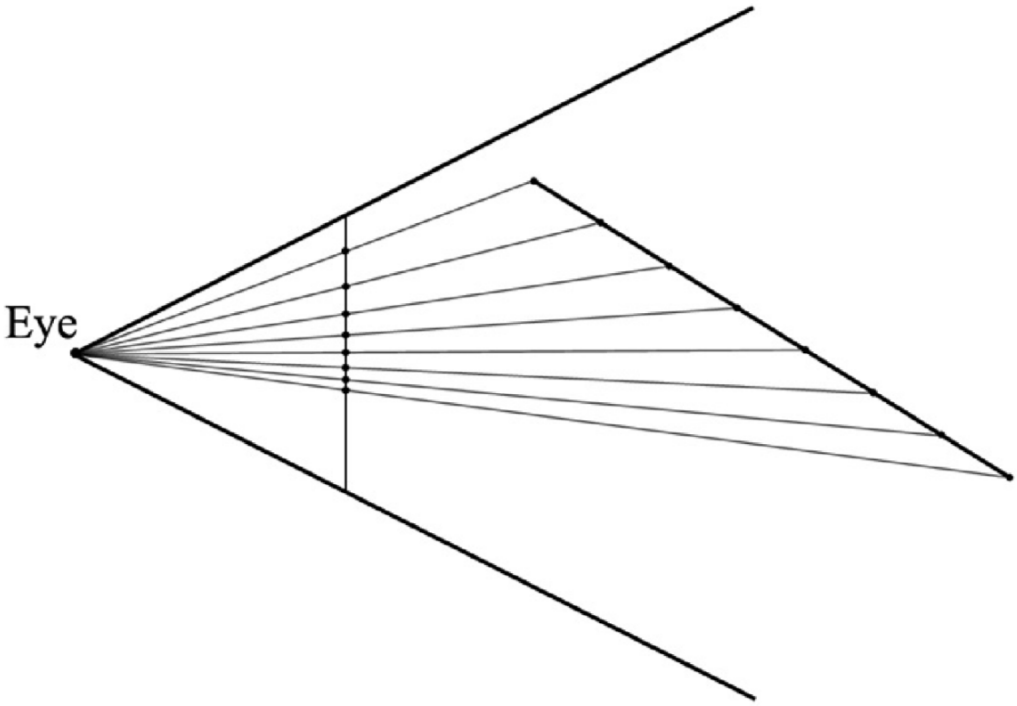


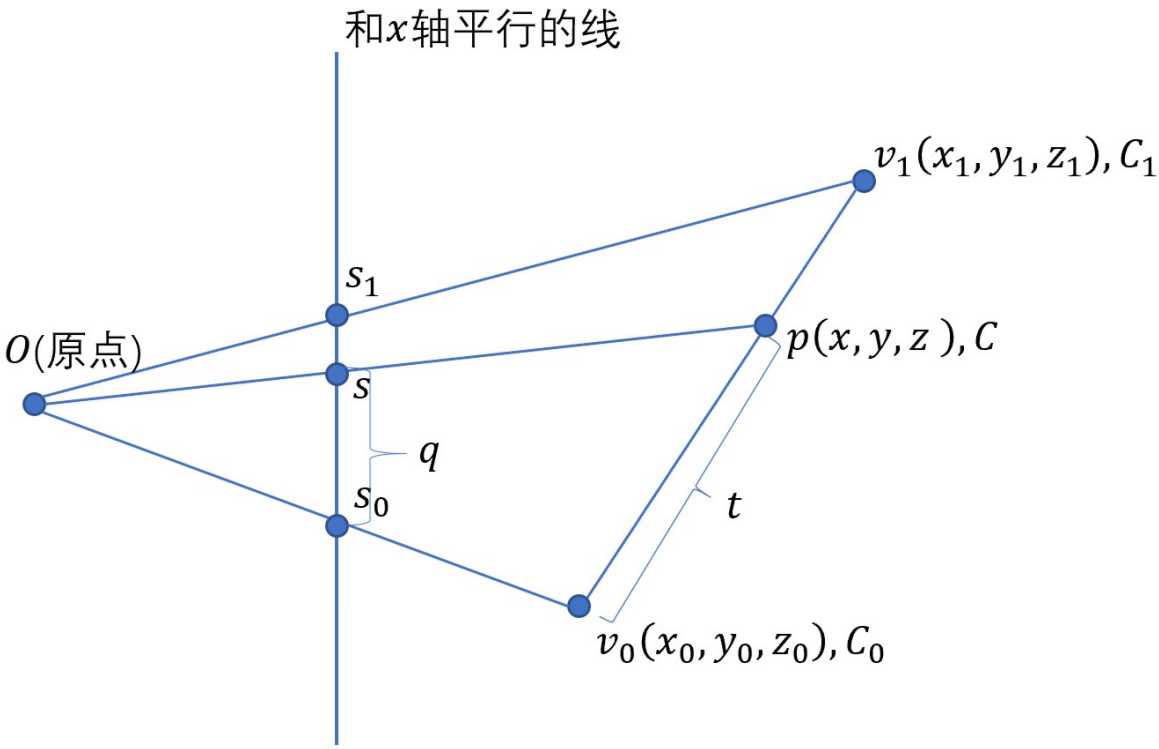
Figure 5.34. A 3D line is being projected onto the projection window (the projection is a 2D line in screen space). We see that taking uniform step sizes along the 3D line corresponds to taking non-uniform step sizes in 2D screen space. Therefore to do linear interpolation in 3D space, we need to do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

[6] : 考虑出现裁剪边的情况，只有一个点在平截头体内部(包括面上)，另外一个外部这样的情况。那么对于$(e.x - s.x) + (e.w - s.w)$ 来说，我们整理下:

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

由于其中一个点在外部，一个点在内部，那么肯定有一部分为非负数，有一部分为负数。所以两者相减不可能为0 。

[7] : 投影后的三角形是经过透视除法运算的三角形，你没有办法去对一个经过除法运算的式子去进行插值，那样会导致错误的结果。因此我们得去对空间中没有经过透视除法运算的原三角形进行插值。换种解释方法，参见图片。



[8] : 下面是式子推导。

首先我们证明:

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

插值:

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

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

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

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

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

那么就证明了。接下来我们来考虑下面的式子:

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

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$\frac{1}{z_0*z_1}$

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

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

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

引用

- 非彩色图片出自Introduction to 3D Game Programming with Directx 12这本书，其余图片出自Power Point。
- 推荐一个网站: www.scratchapixel.com

结语

感谢阅读，如果有疑问欢迎邮件。

E-mail: LinkClinton@outlook.com