3D 图形学知识汇编

前　　言

最近我在学习计算机图形学的时候大量的上网翻书查阅资料，但是从网上和部分书籍里面看到的知识有些零散，所以我作此书，将这些知识点串联起来。我的演示代码只是提供一个参考，希望读者通过阅读本文自己实现相应功能。本书主要是介绍 3D 渲染的流程和原理，所以有几乎没有对代码进行优化。如果要实现可用的高性能的渲染器，代码优化是必不可少的。

目录

第一部分：预备知识 1
第 1 章 在屏幕上绘制图案 2
1.1 像素 2
1.2 像素的抽象表示 2
1.3 通过控制不同像素的亮度来组合成不同图案 3
1.4 单独控制每个像素的亮度 4
1.5 Windows 和 Linux 上的程序实现 4
1.5.1 使用 EasyX 实现 windows 下的像素绘制 4
1.5.2 使用 FrameBuffer 实现 Linux 下的像素绘制 5
1.6 CMake 的简单使用 5
第 2 章 直线和三角形的绘制 6
2.1 线段的绘制 6
2.2 三角形的绘制 7
第 3 章 重心坐标插值 13
3.1 插值 13
3.2 重心坐标插值 14
3.2.1 一维重心坐标插值 14
3.2.2 二维重心坐标插值 14
3.2.3 重心坐标插值的规律 16
3.2.4 重心坐标插值在栅格化程序的优化 17
3.2.5 多属性的插值 19
3.3 二维纹理 21
第二部分：进阶知识 22
第 4 章 齐次坐标和透视投影 23
4.1 齐次坐标 23
4.2 透视投影 23
4.2.1 二维透视投影 23
4.2.2 三维透视投影 25
4.2.3 用齐次坐标表示透视投影 26
第 5 章 透视校正插值 27
5.1 空间三角形的绘制 27
5.2 绘制两个三角形拼接成一个正方形 28
5.3 仿射变换导致的错误 28

5.4 透视校正插值 29
5.5 附录 34
5.5.1 附录一 34
5.5.2 附录二 36
5.5.3 附录三 37
5.5.4 附录四 38
第 6 章 深度测试 39
6.1 画家算法 39
6.2 Z-Buffer 算法 40
第 7 章 裁剪 42
7.1 一些裁剪的基本知识 42
7.2 深度值的另外一种表示 43
7.3 近平面裁剪的必要性 46
7.4 裁剪的实现 46
7.5 附录 50
7.5.1 附录一 50
第 8 章 标准视锥体和设备坐标 51
8.1 设备坐标 51
8.2 设备坐标的标准化 53
8.3 将多边形裁剪到视锥体内部 55
8.4 一个核心功能完备的渲染器出炉了 58
第三部分：渲染器的功能扩展 60
第 9 章 渲染器功能升级 61
9.1 OpenGL 渲染管线简介 61
9.2 改造我们的渲染器 61
9.3 三角形剔除 63
9.4 屏幕刷新 65
9.4.1 上一帧残留数据清理 65
9.4.2 双缓冲 66
第 10 章 矩阵 68
10.1 透视投影矩阵 68
10.2 缩放矩阵 68
10.3 平移矩阵 69
10.4 旋转矩阵 69

10.5 坐标变换的一些技巧 70
10.6 完成旋转动画 70
第 11 章 走进图形学的大门 72
11.1 漫反射光照模型 72
11.2 纹理生成 73
11.3 模型读取 73
11.4 最后一个练习题 73

第一部分：预备知识

在这最初的第一部分中，我们先学习 2D 图像绘制的一些相关知识，我们将会学习最基础最核心的知识：在屏幕上绘制图案。只有能在屏幕上绘制图案，我们才能在 3D 的世界遨游。

第1章在屏幕上绘制图案

1.1 像素

有时候我们在使用屏幕的时候不小心滴了一滴水到屏幕上面，我们透过水滴就可以看到屏幕是由一个个的像素组成的，这时候小水滴就相当于一个放大镜，我们可以看到屏幕的一个局部大概是这样子的：

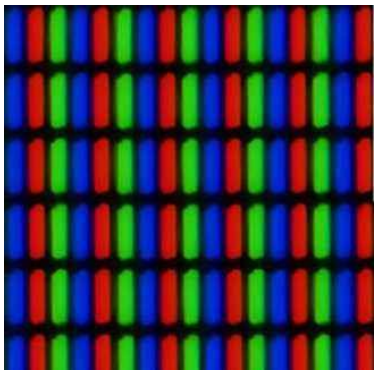


图 1-1 一个彩色显示器的局部放大

我们可以看到一个个类似小灯一样的东西，我们通常把红绿蓝三种小灯各包含一个的单元叫做一个像素，通过控制每个像素中每个小灯(实际上 LED 屏幕才是真正的小灯，其他类型的屏幕不一定是小灯，比如 LCD 屏幕是通过液晶的偏转来控制 RGB 中每个颜色的明暗，CRT 显示器则是通过电子束的强度来控制明暗，但是原理都是通过明暗来表示颜色)的亮度，就可以在屏幕上显示五彩缤纷的图案。

1.2 像素的抽象表示

实际上在真实的屏幕上，像素的排列方法各种各样，除了下面图片中的几种方式外还有其他不同的方法。

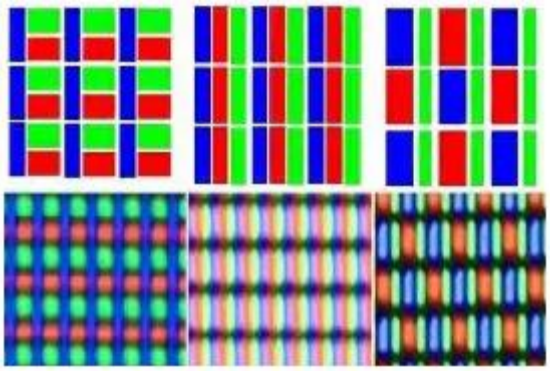


图 1-2 上：像素排列示意图 下：屏幕实拍图

在不同屏幕上，各个小灯的排列方式是不同的，我们可以把这些不同屏幕抽象成下面这种抽象屏幕(以后本书中简称屏幕)：

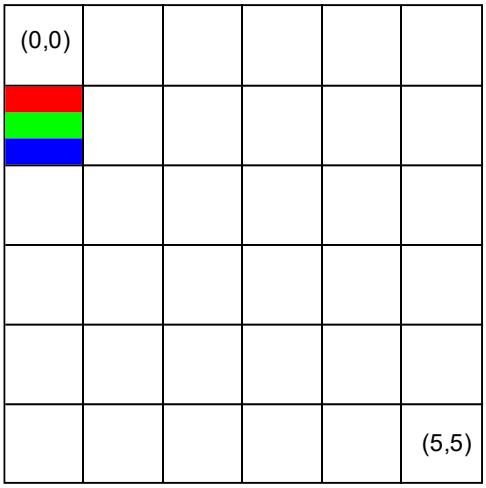


图 1-3 抽象出来的屏幕

上图中表示的是一个 6*6 大小的屏幕，每个格子代表一个像素，每个像素包含一个由红绿蓝三个小灯组成的单元(如图 1-3 中的第(0,1)像素所示)。这样可以通过控制每个像素 RGB(光的三原色为红绿蓝三种光)的亮度组合成不同的颜色。通常在屏幕上我们使用左上角作为屏幕坐标原点，向右为 X 轴正方向，向下为 Y 轴正方向。这里可能有的同学会好奇：为什么不以左下角作为原点，向上作为 Y 轴正方向呢？这是因为以前还在使用 CRT 显示器的时候，这种显示器使用电子来轰击屏幕上面的荧光粉来发光产生亮度。这些电子是从屏幕后面的电子枪中发射出来的，这也是为什么这种显示器必须要有一个大大的屁股，这后面放的就是电子枪。从电子枪中发射出来的这些电子被两组偏转线圈控制着按照一定的规律轰击屏幕，如右图 1-4 所示：电子先横向从左到右扫描一次，然后短暂关闭并移动到下一行的最左边，再次从左到右扫描，这就是为什么屏幕坐标以左上角为原点，向右为 X 轴正方向，向下为 Y 轴正方向的原因了，并且横向的一次扫描轨迹称之为扫描线。同时类似于 720P 指的就是指屏幕有 720 条扫描线，其中 P 是 Progressive 的缩写。

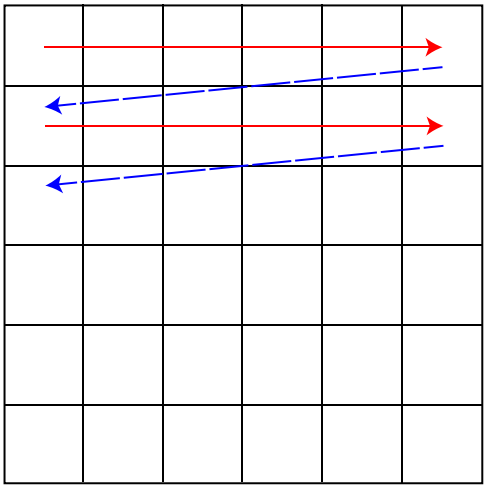


图 1-4 扫描线示意图

1.3 通过控制不同像素的亮度来组合成不同图案

我们可以知道在屏幕上面显示的图片和文字等图案都是有一个个像素组成的，我们只需要控制每个像素的明暗程度就可以组成不同的图案。

如图 1-5 所示，我们通过控制有些像素发光而另外一些像素不发光可以显示出一个图案，实际中的屏幕往往不会有图中的方格参考线。

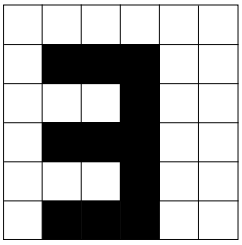


图 1-5 数字 3 在屏幕上面的表示

1.4 单独控制每个像素的亮度

单独控制每个像素是图形显示最基本的方法，我们在用手机、电脑等工具看图片、视频或者文字的时候，也是通过软件或者硬件来计算每个像素应该呈现出什么颜色(对于彩色显示器来说一般是通过控制 RGB 亮度)来组合成一幅完整的图案，计算机图形系统中通常都会提供画点的 API 来供程序绘制图案。

在 1987 年 IBM 提出了一种叫做 VGA(Video Graphics Array)的显示标准，VGA 支持 APA 图像模式和字符模式两种显示模式。一般情况下 VGA 控制器的显存会被映射到内存地址从 0xA0000 到 0xBFFFF 的这段空间中，我们可以通过设置 VGA 控制器的某些寄存器使其工作在不同的模式，其中 0xA0000 到 0xBFFFF 这段空间被用于图形显示模式，0xB0000 到 0xB7777 这段空间被用于单色字符显示模式，0xB8000 到 0xBFFFF 被用于彩色字符模式。VGA 控制器工作于图形显示模式的时候，会将屏幕上面的每一个像素和显存对应，我们修改相应地址的内存内容，则变化会被反应到屏幕上。

在 windows 中因为对物理地址和虚拟地址进行了映射，所以用户编写的程序不一定能修改 0xA0000 处的内存，并且可能 0xA0000 处的虚拟地址不一定对应着 0xA0000 的物理地址，所以我们不能在 windows 系统中直接修改这段内存来实现指定某个像素的颜色，在 windows 中通常需要借用一些系统或者显卡驱动的 API 才能实现对某个像素的修改。Linux 下面有一个叫做帧缓冲设备(FrameBuffer)的东西，这个设备是 VGA 显存的一个映射，直接对这个设备进行读写可以达到修改显存的效果。但是现在的显卡通常会有更复杂的显示方法来支持新的功能，想要操作某个像素需要更加复杂的方法，GPU 制造商通过设备驱动程序的方式暴露出 API 给用户使用，所以我们可以使用一些 API 来减少实现画点的功能的工作量。在 Windows 下我们可以使用 GDI、EasyX 等相关库，而在 Linux 下我们可以使 FrameBuffer 或者是一些其他库来对屏幕上面某个像素实现操作，而且还有一些其他库比如 SDL、QT 等库甚至可以跨平台使用，但是他们的使用相对来说较为复杂。

1.5 Windows 和 Linux 上的程序实现

为了减少我们的代码工作量，我们了 EasyX 来在 windows 实现屏幕的画点功能，EasyX 还有一些其他功能，但是在本文中我们暂时不使用，而在 Linux 我们将会使用 FrameBuffer 实现，并且我们将画点的功能封装起来方便后面使用。相关代码在 chapter1/Windows 和 chapter2/Linux 中。

1.5.1 使用 EasyX 实现 windows 下的像素绘制

我们可以通过 EasyX 的官网 https://easyx.cn 下载库帮助我们实现在 Windows 下的程序开发。EasyX 是针对 C++ 的图形库，可以帮助 C++初学者快速上手图形和游戏编程，EasyX 的安装很简单，只要照着安装程序点击下一步即可。

我们先实现一个名为 GraphicsDevice 的类，这个类用于完成图形设备的基本管理和画点功能。我们在构造函数完成绘图窗口的创建，在析构函数中完成绘图窗口的关闭回收，同时实现了画点函数 SetPixel，在这个函数中，我们对 Y 坐标进行了一个转换，使得屏幕的原点从左上角变为了左下角，这样更符合我们的日常使用习惯。

1.5.2 使用 FrameBuffer 实现 Linux 下的像素绘制

在 Linux 程序中，我们同样构造一个 Graphics 类完成绘图设备的管理和画点功能，只是具体实现相比 Windows 略有不同。需要注意的是，非 root 用户没有权限打开 Framebuffer 设备，并且需要按下 Ctrl+Alt+F1~F7 切换到字符终端才能看到更新 FrameBuffer 带来的改变。

1.6 CMake 的简单使用

我们的代码只有图形设备创建、回收和画点是与平台相关的，后续的各种算法都是和平台无关的代码，所以这次我们使用 CMake 来管理项目代码，这样在不同平台下可以自动把对应的文件包含进来，我们就不用每个代码都要写两份了，我们把 GraphicsDevice 的不同实现放在 platform\linux 和 platform\windows 下面。安装 VS 之后会自带 CMake，我们假设源码放在 A 目录，我们想在 B 目录生成工程，则只需要在 B 目录执行 CMake A 即可，如：源码放在 C:\ code\src\chapter1 中，我们想要在 C:\myproject 中生成项目，我们只需要按如下操作即可：

CD C:\myproject

cmake C:\ code\src\chapter1

这样就可以在 C:\myproject 创建项目了,如果是在 Windows 下，则会生成.sln 文件，双击这个文件即可打开解决方案，我们可以发现这个解决方案包含了三个项目：

图 1-6 解决方案示意

其中 ALL_BUILD 和 ZERO_CHECK 这两个项目是 cmake 自动生成的，我们暂时用不上，所以我们可以在我们自己的项目上，如 DrawPixel 上面右击鼠标，然后选择设为启动项目即可使用 VS 调试我们的程序。在 Linux 上面使用也是同理，先使用 cmake 构建项目，然后会在目标目录生成相关 makefile，然后执行 make 即可生成可执行文件 DrawPixel。

第2章 直线和三角形的绘制

在屏幕上显示的图形总是能被分割成有限的顶点、线段或者三角形，以点、线、面作为图形的基础可以完成丰富多彩的呈现效果。点的绘制在上一章中我们已经详细介绍过了，本章将会简单的介绍线段和三角形的绘制，我们把和图形学相关的算法放在lib/GraphicsLibrary.h 中，这个文件定义了一些数据结构和算法，具体的实现放在同目录下的GraphicsLibrary.cpp 中。

2.1 线段的绘制

线段是直线上两点之间有限的部分，所以我们想要绘制线段需要知道线段的两个端点P1、P2。直线的绘制算法有很多种，这里我只简单的介绍一下 DDA 算法。DDA 算法是 Digital Differential Analyzer 的简称，通过直线中 x 或者 y 的增量来绘制直线的一种算法。

在图 2-1 中，我们在屏幕上绘制一个从 P1(0,1)到 P2(3,4)的线段，我们可以先计算线段的 Y 值在 X 方向上的增量，即每当 X 值变化为 1 时，Y 值的变化量ΔY，然后在我们绘制下一个点时，令 X'=X+1，Y'=Y+ΔY。很明显，Y 在 X 方向的增量为该线段的斜率$dy=\frac{p2_y-p1_y}{p2_x-p1_x}=\frac{4-1}{3-0}=1$。我们先绘制起点(0,1)，则下一点 X=1 时,Y=1+1=2，依次绘制到终点(3,4)完成本条线段。上述思想在编程上的实现为：

```
DrawLine(P1,P2)
{
    dy=斜率
    起点=X 值小的点
    x=起点 x
    y=起点 y
    for(从起点的 x 到终点的每个 x，x 依次加 1)
    {
        绘制(x,y)
        y=y+dy
    }
}
```

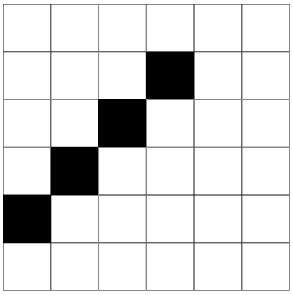


图 2-1 线段绘制示意图

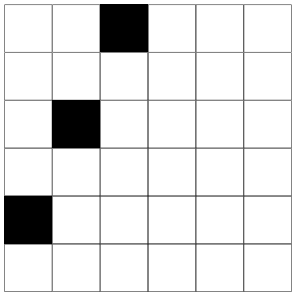


图 2-2 断裂的线段

但是如果我们使用上述代码绘制从点(0,1)到点(2,5)的线段，就会发现，在斜率 k>1 或者 k<-1 时，会出现线段断裂的现象，如图 2-2。这是因为斜率大于 1 或者小于 1 时，在 X 增加 1 的时候，Y 值的变化量超过了 1，就会和上一个点不再连贯。所以我们可以改进一下上面的代码，不再单独使用 dy 来控制 y 的增量，当线段满足$|k|\le 1$时，我们仍然使用上述算法，但是当$|k|\ge 1$时，我们计算出 X 在 Y 方向上的增量 dx，令 X'=X+dx，Y'=Y+1，这样就可以解决线段断裂的问题。

```
DrawLine(P1,P2)
{
    if(斜率绝对值小于等一 1)
    {
        dy=斜率
        起点=X 值小的点
        x=起点 x
        y=起点 y
        for(从起点的 x 到终点的每个 x，x 依次加 1)
        {
            绘制(x,y)
            y=y+dy
        }
    }
    else
    {
        dx=1/斜率
        起点=y 值小的点
        x=起点 x
        y=起点 y
        for(从起点的 y 到终点的每个 y，y 依次加 1)
        {
            绘制(x,y)
            x=x+dx
        }
    }
}
```

代码见 lib\GraphicsLibrary.cpp 中的 DrawLine 函数。

2.2 三角形的绘制

在图形学中，有许多通用多边形绘制方法，但是我们的目的是编写一个 3D 渲染器，只需要绘制三角形即可，所以这里我们只编写一个绘制三角形的函数，在绘制三角形的时候，我们借鉴多边形扫描线填充算法的思想，并将其简化。

我们沿着扫描线序号增加的方向绘制三角形，即使得 Y 值每次递增 1，在每行扫描线上面绘制三角形的一部分。先将三角形的三个顶点按照 Y 值从小到大排序为P_1、P_2、P_3，然后在P_1P_3这条边上面找到一点$P_{'}$，使得$P_{'}$的 Y 值等于P_2的 Y 值，然后再使用P_2、$P_{'}$的 X 值对P_2、$P_{'}$进行排序，使得$P_2.x < P_{'}.x$，这样任意一个三角形都能被上述方法分割成两个类似

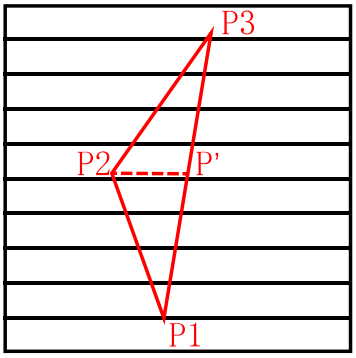


图 2-3 三角形分割

的三角形。在排序之后△ $P_1P_2P_,$和△ $P_2P_,P_3$的绘制方法如下:

```
//下半边三角形的绘制
dx_left=1/P1P2斜率
dx_right=1/P1P,斜率
sx=P1.x//扫描线起点 X 值
ex=P1.x//扫描线终点 X 值
for(从P1.y到P2.y的每个 y)
{
    //sx 到 ex 相当于扫描线的一部分
    for(从 sx 到 ex 的每个 x)
    {
        绘制(x,y)
    }
    sx=sx+dx_left
    ex=ex+dx_right
}
```

```
//上半边三角形的绘制
dx_left =1/P2P3斜率
dx_right =1/P,P3斜率
sx=P2.x//扫描线起点 X 值
ex=P,.x//扫描线终点 X 值
for(从P2.y到P3.y的每个 y)
{
    //sx 到 ex 相当于扫描线的一部分
    for(从 sx 到 ex 的每个 x)
    {
        绘制(x,y)
    }
    sx=sx+dx1
    ex=ex+dx2
}
```

上述两个三角形的绘制代码大部分都是相同的，我们可以简单的合并一下部分代码:

```
DXleft=[1/P1P2斜率, 1/P2P3斜率]//存放左边两条边的斜率倒数
DXright=[1/P1P,斜率, 1/P,P3斜率]//存放右边两条边的斜率倒数
Start_x=[P1.x, P2.x]
End_x=[P1.x, P,.x]
Start_y=[P1.y, P2.y]
End_y=[P2.y, P3.y]
for(int i=0;i<2;i++)
{
   dx_left= DXleft[i]
   dx_right= DXright[i]
   sx=Start_x[i]//扫描线起点 X 值
   ex=End_x[i]//扫描线终点 X 值
   sy=Start_y[i]//三角形起始扫描线
   ey=End_y[i]//三角形结束扫描线
   for(从 sy 到 ey 的每个 y)
   {
      //sx 到 ex 相当于扫描线的一部分
      for(从 sx 到 ex 的每个 x)
      {
            绘制(x,y)
      }
         sx=sx+dx_left
         ex=ex+dx_right
   }
}
```

按照上面的方法，我们已经可以完成三角形的绘制，但是有一点就是如果三角形的面积为 0 的话，我们是不应该绘制这个三角形的，所以我们还需要增加对三角形面积的判断。知三角形的顶点坐标 P1、P2、P3，我们可以用用两条边作为向量得到$\overrightarrow{P_1P_2} \times \overrightarrow{P_1P_3}$，向量叉乘结果的模长即为三角形面积的两倍。因为叉乘的两个向量是三维向量，所以我们构造一个坐标系，以原先屏幕的 x、y 轴作为坐标轴，垂直于屏幕向外作为 z 轴(右手坐标系)，这样原先的坐标点(x,y)就会变成(x,y,0),得到如下计算公示:

$$S = \overrightarrow{P_1P_2} \times \overrightarrow{P_1P_3} = \begin{vmatrix} i & j & k \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} \frac{P1.x*P2.y+P2.x*P3.y+P3.x*P1.y-P1.x*P3.y-P2.x*P1.y-P3.x*P2.y}{2}$$

在上式中$a = \overrightarrow{P_1P_2}$，$b = \overrightarrow{P_1P_3}$，这个公式计算出来的结果可以为负数，因此这个面积也叫做有向面积，在后面说三角形剔除的时候我会简单的介绍有向面积。因为我们只需要判断面积是否等于 0，所以我们不需要下面的分母 2，这样可以减少一点计算。所以最终判断条件为:

```
if((P1.x * P2.y + P2.x * P3.y + P3.x * P1.y − P1.x * P3.y − P2.x * P1.y − P3.x * P2.y)==0)
{
    放弃本三角形的绘制
}
```

同时因为计算机表达数字总会有误差，浮点数的精度总是有限的，所以我们不能仅仅用面积==0 作为判断依据，如 452.625 + 186.37499999999997 + 239.625 - 452.625 - 186.37499999999997 - 239.625，我们用肉眼观察都能算出结果等于 0，因为前面三个加数和后面三个减数一样，然而用计算机计算却可能输出 2.84217e-14。所以当一个浮点数的值和数字 A 之间的差值小于某个值时，我们可以认为这个浮点数等于 A，上述代码还需要修改为：

```
if(abs((P1.x * P2.y + P2.x * P3.y + P3.x * P1.y − P1.x * P3.y − P2.x * P1.y − P3.x * P2.y))<ε)
{
    放弃本三角形的绘制
}
```

根据 IEEE 的标准，double 的有效精度为 1e-15，所以当我们使用 double 表示面积的时候，上面代码的ε取ε=1e-15。

上述的代码在大部分情况下都能正确的绘制出一个三角形，但是在一些特殊情况下可能会绘制超出三角形范围的像素，如下图 2-4 所示：

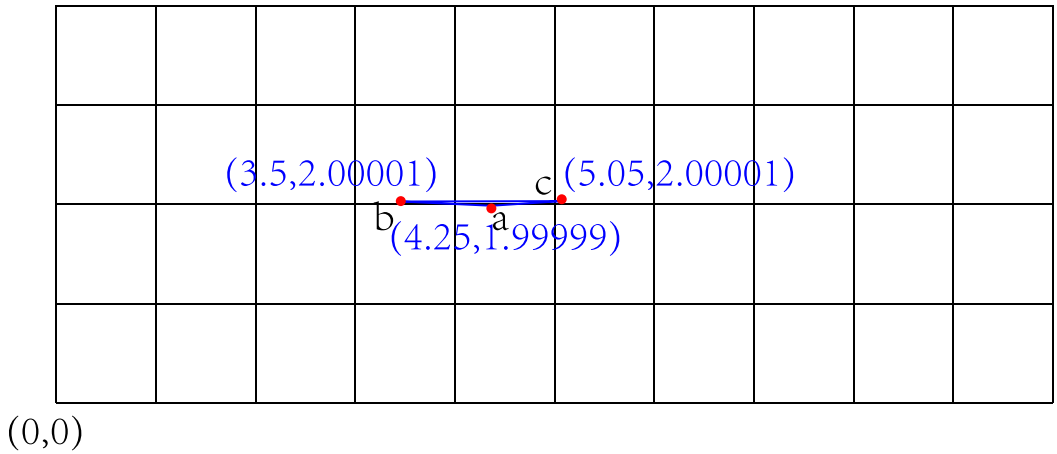


图 2-4 某条边斜率特别小的情况

如果三角形的某条边斜率特别小，那么计算出来的 dx 就会特别大，如上图中的 ab 边 dx=-37500，而 ac 边则达到了 dx=40000，那么我们在绘制三角形第一条扫描线还比较正常，当即将绘制下一条扫描线时，出现了

$$\begin{cases} sx = sx - 37500 = 4.25 - 37500 = -37{,}495.75 \\ ex = ex + 40000 = 4.25 + 40000 = 40004.25 \end{cases}$$

这时候很明显 sx 和 ex 远远超出了三角形范围，甚至超出了屏幕范围。对于上图来说，我们希望 sx 不小于 3.5，ex 不大于 5.05，所以我们还要在上面的代码进行修改,将 sx 和 ex 限定在三角形范围内,绿色部分为新增代码。

```
DXleft=[1/P1P2斜率, 1/P2P3斜率]//存放左边两条边的斜率倒数
DXright=[1/P1P,斜率, 1/P,P3斜率]//存放右边两条边的斜率倒数
Start_x=[P1.x, P2.x]
End_x=[P1.x, P,.x]
Start_y=[P1.y, P2.y]
End_y=[P2.y, P3.y]
edge_left_ex = [ P2.X, P3.X ]; //记录每条边结束的 x 值
edge_right_ex = [ P_.X, P3.X ];
for(int i=0;i<2;i++)
{
   dx_left= DXleft[i]
   dx_right= DXright[i]
   sx=Start_x[i]//扫描线起点 X 值
   ex=End_x[i]//扫描线终点 X 值
   sy=Start_y[i]//三角形起始扫描线
   ey=End_y[i]//三角形结束扫描线
   for(从 sy 到 ey 的每个 y)
   {
      if (dx_left<0)//sx 随着 y 增大而减小
      {
            sx = max(sx,edge_left_ex[i]);//将 sx 限定为不小于终点 x
      }
      else if(dx_left >0)//sx 随着 y 增大而增大
      {
            sx =min(sx,edge_left_ex[i]);//将 sx 限定为不大于终点 x
      }
      if (dx_right < 0)//ex 随着 y 增大而减小
      {
            ex = max(ex, edge_right_ex[i]);//将 ex 限定为不小于终点 x
      }
      else if (dx_right > 0)//ex 随着 y 增大而增大
      {
            ex = min(ex,edge_right_ex[i]);//将 ex 限定为不大于终点 x
      }
      //sx 到 ex 相当于扫描线的一部分
      for(从 sx 到 ex 的每个 x)
      {
            绘制(x,y)
      }
         sx=sx+dx_left
         ex=ex+dx_right
   }
}
```

至此，三角形的绘制工作已经完成，相关代码见 lib\GraphicsLibrary.cpp 中的 DrawTriangle2D 函数。

第3章 重心坐标插值

3.1 插值

当我们已知一些坐标和该坐标记录的数据值，然后对未记录数据值的坐标进行推算，这个过程就叫做插值。这样说可能有点绕口，所以同学们请看下图 3-1：

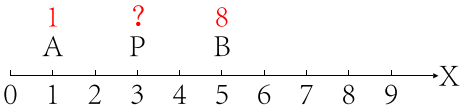


图 3-1

假设我们在一维数轴上有两点 A 和 B，已知点 A 的亮度值为 1，B 的亮度值为 8，那么 P 的亮度值为多少？所以这时候我们需要一个函数用于标记属性值 V 和坐标 X 的关系。假设 V 和 X 是线性变化的，即 V 关于 X 的函数为：$V = aX + b$这种一次多项式，则我们很容易得到一张关系图如下图 3-2 所示：

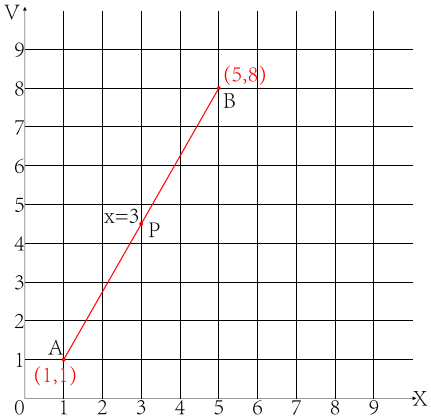


图 3-2 线性插值函数

我们可以很容易的计算出点 P 的 V 值为：

$$V_p = V_a + (X_p - X_a) \times \frac{V_b - V_a}{X_b - X_a} = 1 + (3 - 1) \times \frac{8 - 1}{5 - 1} = 4.5$$

这种根据 AB 点的属性值来预测 P 点属性值的做法就叫做插值，并且我们选则的插值函数为一次多项式插值函数，所以也叫做线性插值。同理，我们可以使用抛物线、对数等函数进行插值，则相应的插值方法也叫做抛物线插值和对数插值。下图 3-3 示意的就是使用二次函数插值的结果。

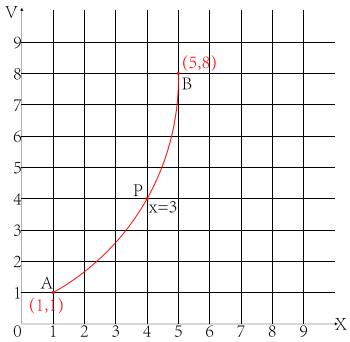


图 3-3 其他类型的插值函数

3.2 重心坐标插值

3.2.1 一维重心坐标插值

我们先介绍一下一维情况下的重心坐标，如下图 3-4 所示：

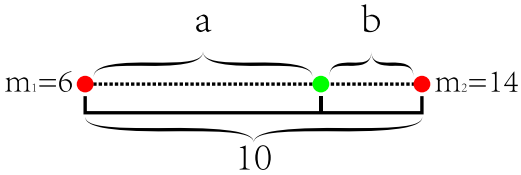


图 3-4 重心坐标

假设现在已知在一维空间有两个质点，两质点距离为 10，左侧质点的质量为 6，右侧质点的质量为 14，现在用一根无质量的轻杆将其连接，求这个部件的重心位置，像不像一个物理题。我们可以很轻松的解答出这个问题，因为两质点的质量之比为 3/7，杆长为 10，所以绿色点重心的位置应该在距离左边质点 7 的位置，也就是在上图 3-4 中，a=7，b=3。我们也容易发现，重心的位置和质量比值相关，即在上图 3-4 中存在如下的关系:$\frac{a}{b}=\frac{m_2}{m_1}$，其中通过杆长 l 和 a 或者 b 可以确定重心的坐标。同样，假如我们确定了重心的坐标和总质量 M，也可以求出m_1和m_2，假设现在已知:$\frac{a}{b}=\frac{7}{3}$，总质量 M=20，则可以计算出 $m_1=6$、$m_2=14$。

假设我们令总质量 M=1，令两质点的质量为他们的权值，我们就可以得到一个重心坐标 P 属性值的插值公式：$V_p=W_1\times V_1+W_2\times V_2$。如下图 3-5 所示：

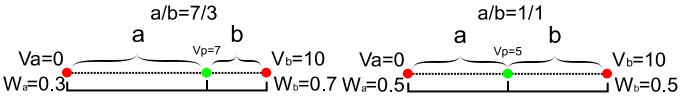


图 3-5 重心坐标插值

设左右两点有属性值 V 分别为 1、10，令总质量 M=1。在图 3-5 左中，点 P 的位置为 a/b=7/3，则得到$m_1=0.3$，$m_2=0.7$，$V_p=0.3\times 0+0.7\times 10=7$，。在右图中则有$m_1=0.5, m_2=0.5, V_p=0.5\times 0+0.5\times 10=5$。

如果我们现在知道点P_1、P_2和 P 的坐标，知道了V_1、V_2，则V_p重心坐标插值的结果为：

$$V_p=\frac{P_2-P}{P_2-P_1}*V_1+\frac{P-P_1}{P_2-P_1}*V_2$$

在上述公式中$\frac{P_2-P}{P_2-P_1}$为P_1的权值w_1，$\frac{P-P_1}{P_2-P_1}$为P_2的权值w_2，并且$w_1+w_2=1$，这个公式不仅仅能对P_1P_2内部的点 P 进行插值，并且当 P 不在P_1P_2内部时同样能够插值。

3.2.2 二维重心坐标插值

二维情况下的重心坐标和一维类似，如下图 3-6：

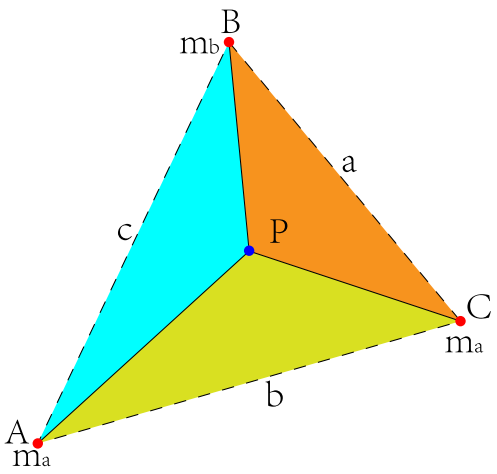


图 3-6 三角形的重心

假设在二维平面上有三个质点 ABC，被三个无质量的轻杆 abc 连接成一个三角形，abc 为顶点 ABC 的对边，其质量分别为m_a、m_b、m_c，现在求其重心 P。我们同样可以很快得到结果：我们设边 a 和 P 围成的三角形为△a，同样的有△b 和△c，我们得到如下等式

$$\frac{S_a}{S}=\frac{m_a}{M}$$

$$\frac{S_b}{S}=\frac{m_b}{M}$$

$$\frac{S_c}{S}=\frac{m_c}{M}$$

其中S_a、S_b、S_c分别为△a、△b、△c 的面积，S 为△ABC 的面积，$M=m_a+m_b+m_c$，我们回到类似一维重心插值的讨论，假设我们已知重心 P 的坐标，令 M=1,则三个小三角形的面积比就是三个顶点的权重比。即：

$$W_a=\frac{S_a}{S}$$

$$W_b=\frac{S_b}{S}$$

$$W_c=\frac{S_c}{S}$$

并且小三角形△a、△b、△c 的面积同样可以取负值，我们以顶点 A 和△a 来举例说明，如下图 3-7：

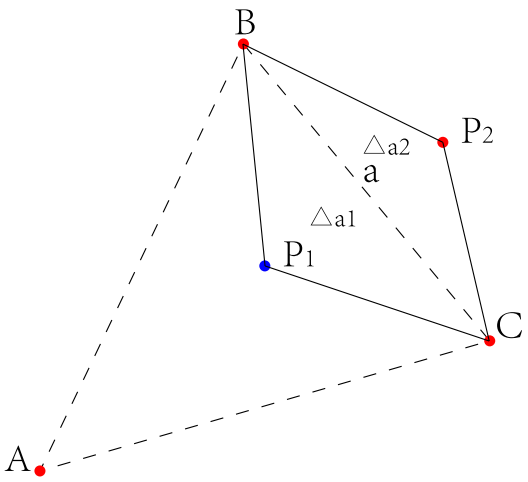


图 3-7 有向面积

我们可以认为三角形的面积是有方向的，假设有顶点 A 和待插值点 P,则当 P 和 A 在边 a 的同一侧的时候$S_a > 0$，当 P 和 A 在边 a 的异侧时$S_a < 0$。上图中的边 a 和 P1 围成的△a1 面积为正，△a2 面积为负。

3.2.3 重心坐标插值的规律

我们通过在一维和二维的重心坐标插值研究可以发现，当代插值点 P 在一条直线上运动时，顶点的权重和 P 的位移距离线性相关，如下图 3-8 所示:

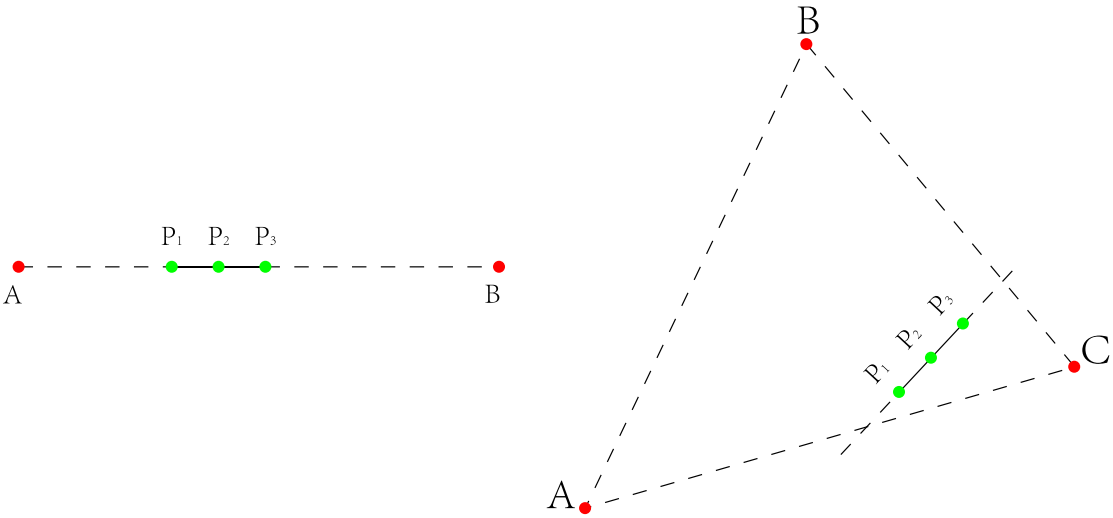


图 3-8 重心坐标插值的线性变化

在上面左图和右图中,设 A 点对于P_1、P_2、P_3的权值分别为W_{p1}、W_{p2}、W_{p3}，如果有$P_2 - P_1 = P_3 - P_2$，则有$W_{p2} - W_{p1} = W_{p3} - W_{p2}$。因为小三角形的面积变化和 P 点在直线上的位置也是线性相关的，所以上述结论对于三角形中的 B 点或者 C 点同样成立，并且如果设三个点的属性值分别为V_{p1}、V_{p2}、V_{p3}，同样也有$V_{p2} - V_{p1} = V_{p3} - V_{p2}$。也就是说如果一个变量在一条直线上是线性均匀变化的，我们就能使用重心坐标插值对其进行计算，而且在直线上插值的时候，重心坐标插值和线性插值是等价的。以上规律可以推导到高维空间中，如在三维空间中，用四个不共面的顶点确定一个三棱锥，用一个点 P 将三棱锥分割成四个小三棱锥，则顶点对面的小三棱锥体积比就是权值之比。

3.2.4 重心坐标插值在栅格化程序的优化

计算三角形面积花费的运算量比较大，而重心坐标插值的结果在直线上是线性变化的，所以我们可以将重心坐标插值按照扫描线的填充顺序进行插值，如下图 3-9 所示:

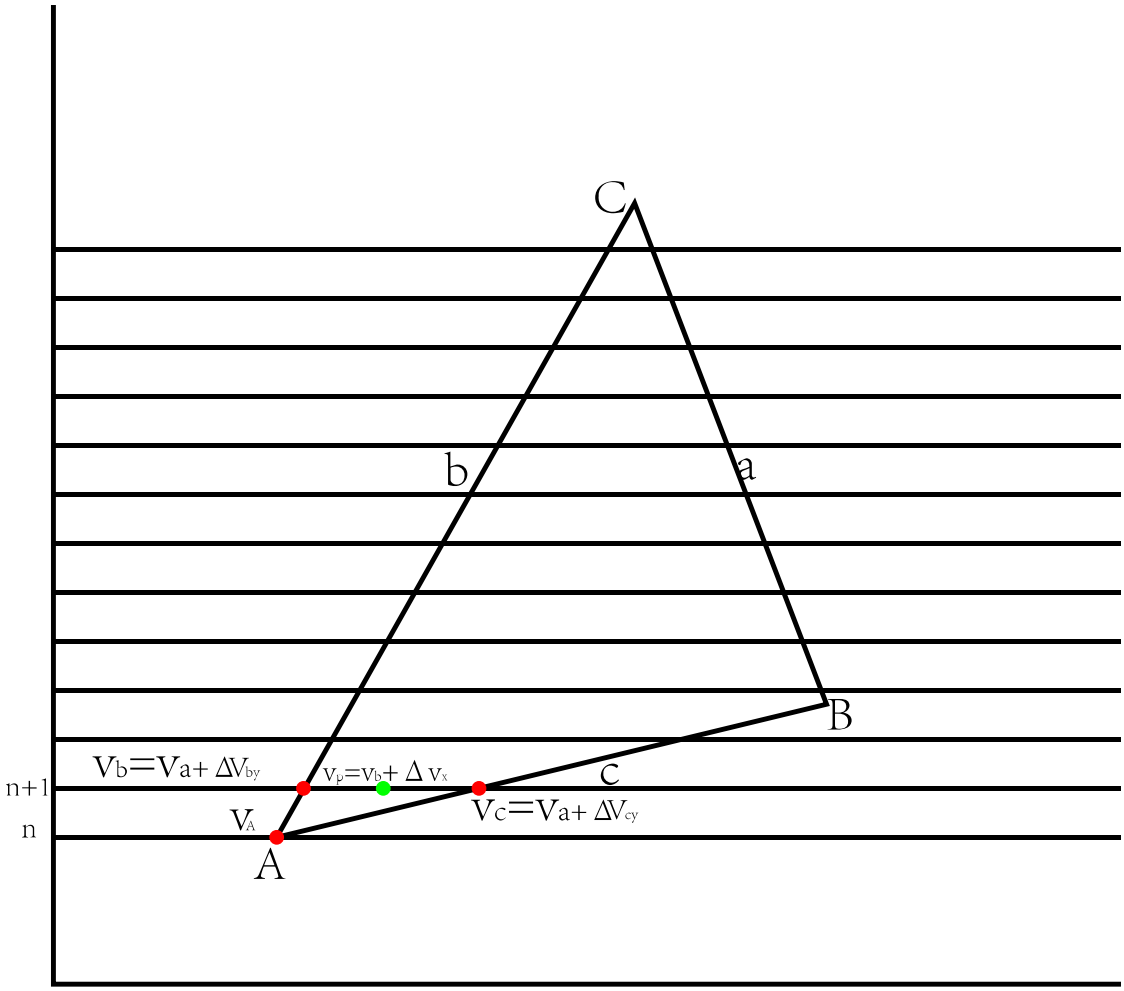


图 3-9 重心坐标插值的优化

我们在逐扫描线填充三角形的时候，先计算出属性值在三条边在 Y 方向上的变化率ΔV_{ay}、ΔV_{by}、ΔV_{cy}，此时得到在 n+1 行扫描线的时候,b 边上的属性值为$V_A + \Delta V_{by}$，c 边上的属性值为$V_A + \Delta V_{cy}$，然后再在同一行扫描线利用V_b和V_c计算出绿色点的属性值 V 在 X 方向的变化率ΔV_x。我们只需要将绘制三角形的函数稍作添加，即可在填充像素的同时计算出该像素重心坐标插值的结果，下面代码中黄色部分就是我们添加的主要代码。

```
/*
在顶点排序求P,的时候使用线性插值计算P,的属性值
*/

DXleft=[1/P1P2斜率, 1/P2P3斜率]//存放左边两条边的斜率倒数
DXright=[1/P1P,斜率, 1/P,P3斜率]//存放右边两条边的斜率倒数
DVyleft=[P1P2边属性值在 Y 方向的变化量，P2P3边属性值在 Y 方向的变化量]
DVyright=[P1P,边属性值在 Y 方向的变化量，P,P3边属性值在 Y 方向的变化量]
Start_x=[P1.x, P2.x]
Start_v=[P1.v, P2.v]
End_x=[P1.x, P,.x]
End_v=[P1.v, P,.v]
Start_y=[P1.y, P2.y]
End_y=[P2.y, P3.y]
for(int i=0;i<2;i++)
{
   dx_left= DXleft[i]
   dx_right= DXright[i]
   dv_left= DVyleft[i]
   dv_right = DVyright [i]
   sx=Start_x[i]//扫描线起点 X 值
   sv=Start_v[i]//扫描线起点属性值
   ex=End_x[i]//扫描线终点 X 值
   ev= End_v[i]//扫描线终点属性值
   sy=Start_y[i]//三角形起始扫描线
   ey=End_y[i]//三角形结束扫描线
   for(从 sy 到 ey 的每个 y)
   {
      //sx 到 ex 相当于扫描线的一部分
      v=sv
      dvx=(ev-sv)/(ex-sx)//计算属性值 v 在 x 方向上的增量
      for(从 sx 到 ex 的每个 x)
      {
            绘制(x,y,v)
            v=v+dvx
      }
         sx=sx+dx_left
         sv= sv+dv_left
         ex=ex+dx_right
         ev=ev+dv_right
   }
}
```

为了让顶点能够附带属性值，我们给 Point2 这个类新增了一个属性 V，这次我们把属

性值用作像素的灰度值，我们可以看到一个从白色到黑色的渐变三角形。相关代码见 chapter3\SingleInterpolation 和 lib\GraphicsLibrary.cpp 中的 SingleInterpolationIn2D 函数。

3.2.5 多属性的插值

因为我们往往不仅仅只需要计算一个属性，而是需要同时对多个属性插值，所以我们将前一节的代码稍作修改即可。我们给 Point2 新增一个属性为 ValueArray，表示顶点的多个属性，然后将前一节的黄色部分代码修改如下，下面的代码中将修改处标为绿色，我们可以发现这份代码和使用单属性值的代码没有太大区别，只是在计算属性值的时候变成了对集合的处理，也就是原来我们只处理一个值，而现在对集合中的每个属性值都使用同样的方法计算，相关代码见 chapter3\MultipleInterpolation 和 lib\GraphicsLibrary.cpp 中的 MultipleInterpolationArrayIn2D 函数，这次我们使用重心坐标插值绘制了一个渐变色三角形,并且我们把绘制像素的函数定义成一个回调函数 fragmentShader，简称 FS，这样就可以由调用者自己决定将插值之后的属性值如何变成像素颜色，我们在 chapter3\MultipleInterpolation\main.cpp 中给 MultipleInterpolation 传递了一个 lambda 表达式，这个函数把传递进去顶点属性值作为像素的 RGB 值。

```
/*
在顶点排序求P,的时候对属性值集合 V 使用线性插值计算
*/

DXleft=[1/P1P2斜率, 1/P2P3斜率]//存放左边两条边的斜率倒数
DXright=[1/P1P,斜率, 1/P,P3斜率]//存放右边两条边的斜率倒数
DVylefts=[P1P2边属性值集合在 Y 方向的变化量, P2P3边属性值集合在 Y 方向的变化量]
DVyrights=[P1P,边属性值集合在 Y 方向的变化量, P,P3边属性值集合在 Y 方向的变化量]
Start_x=[P1.x, P2.x]
Start_vs=[P1.vaulearray, P2.vaulearray]
End_x=[P1.x, P,.x]
End_vs=[P1.vaulearray, P,.vaulearray]
Start_y=[P1.y, P2.y]
End_y=[P2.y, P3.y]
for(int i=0;i<2;i++)
{
   dx_left= DXleft[i]
   dx_right= DXright[i]
   dv_lefts= DVylefts[i]
   dv_rights = DVyrights[i]
   sx=Start_x[i]//扫描线起点 X 值
   svs=Start_v[i]//扫描线起点属性值
   ex=End_x[i]//扫描线终点 X 值
   evs= End _v[i]//扫描线终点属性值
   sy=Start_y[i]//三角形起始扫描线
   ey=End_y[i]//三角形结束扫描线
   for(从 sy 到 ey 的每个 y)
   {
      //sx 到 ex 相当于扫描线的一部分
      vs=svs
      dvxs=(evs-svs)/(exs-sxs)//计算每个属性值 v 在 x 方向上的增量
      for(从 sx 到 ex 的每个 x)
      {
            绘制(x,y,v)
            vs=vs+dvxs
      }
         sx=sx+dx_left
         svs= svs+dv_lefts
         ex=ex+dx_right
         evs=evs+dv_rights
   }
}
```

3.3 二维纹理

如果我们使用一张二维图片作为纹理包裹在一个物体上，这张图片就叫做纹理，“一个图像（纹理）被贴(映射)到场景中的一个简单形体上，就像印花贴到一个平面上一样”-引用自维基百科：材质贴图。

通常我们将纹理的左下角定为纹理的原点，横向向左为 u 轴增长方向，纵向为 v 轴增长方向构成一个 uv 直角坐标系，这个坐标系的 u 轴等于 xy 直角坐标系的 x 轴,v 轴等于 xy 直角坐标系的 y 轴，uv 的取值都是[0,1]，即当 u=1 时，表示 x=width，v=1 时,y=height。我们可以通过纹理坐标取得该点的颜色值，这个颜色称之为纹素。

我们可以通过如下的公式将 uv 坐标转换成纹理原始图片的 xy 坐标：

$$x = width * u$$

$$y = height * v$$

这次我们还使用 MultipleInterpolation 这个函数，只是在 chapter3/texture/main.cpp 中对 FragmentShader 函数的内容做了些许修改，将一张图片的一部分映射到绘制的三角形上。我们在 chapter3\texture\main.cpp 中创建了一个 Texture 类，里面直接使用数组创建一个如下 2*2 大小的图片：

```
int img[2][2] = {
        {RGB(255,0,0),RGB(0,255,0)},
        {RGB(0,0,255),RGB(255,123,172)}
};
```

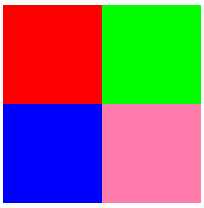


图 3-10 用作纹理的图片

定义了一个函数 texture2D 用于获取纹素颜色，这个函数输入 uv 坐标，返回指定坐标的纹素颜色。定义了一个 FS 函数，用于根据每个像素的属性值(这里我们传入的参数为顶点的纹理坐标)获取纹素值。然后在 main 函数中给三角形的三个顶点赋上纹理坐标属性，调用 MultipleInterpolationArrayIn2D 函数绘制绘制出如下图：

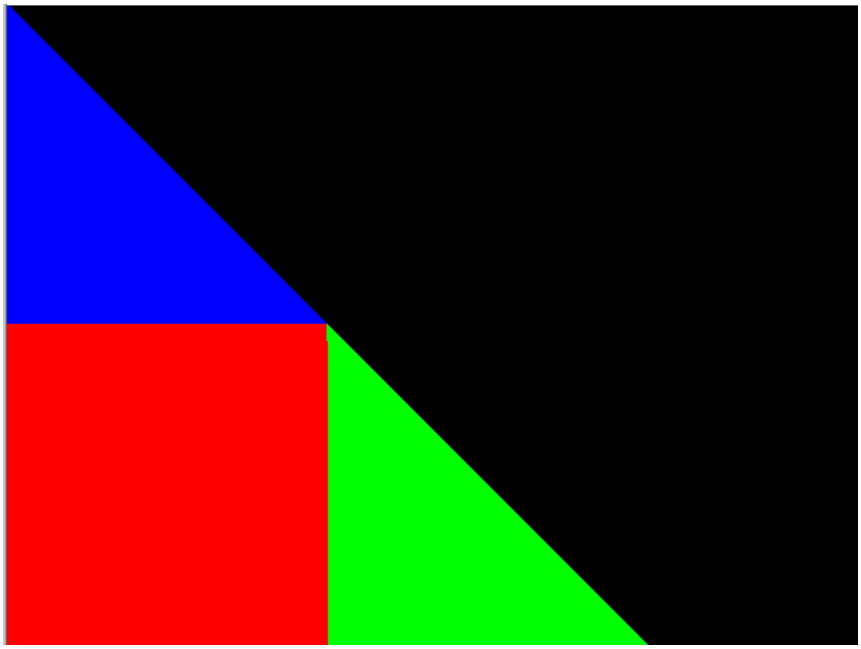


图 3-11 程序运行结果

第二部分：进阶知识

经过第一部分的学习，相信读者们已经对二维图像的绘制有了一个认识，在第二部分中，我们将会进入 3D 的世界，尽情的畅游吧。

第4章 齐次坐标和透视投影

4.1 齐次坐标

转眼间我们已经进入第二部分的学习了，前面我们所使用的坐标都是类似于一维(x),二维(x,y),三维(x,y,z)，用一个分量表示该点在各个坐标轴上面的分量。如果我们给每个坐标额外增加一个分量ω，把上述的坐标变成(x,ω), (x,y,ω), (x,y,z,ω)，并且对ω的作用作出如下解释：$(x_a) = (\omega x_a, \omega)$，也就是有下面这种类似的表达：因为有$2 = \frac{2}{1} = \frac{4}{2} = \frac{6}{3}$，所以(2)=(2,1)=(4,2)=(3,6)，$(\omega x_a, \omega)$叫做(x_a)的齐次坐标，对于二维、三维坐标同样的有$(x_a, y_a) = (\omega x_a, \omega y_a, \omega)$和$(x_a, y_a, z_a) = (\omega x_a, \omega y_a, \omega z_a, \omega)$，等号后面的坐标都是前面坐标的齐次坐标，齐次坐标是用一个 N+1 维的坐标来表示 N 维坐标。

齐次坐标可以方便的使用矩阵对坐标进行变换，同时齐次坐标的ω分量还可以记录额外的信息，如在进行投影运算的时候可以记录原始点的深度信息。这些好处可以让我们忍受多写一个分量带来的麻烦。

另外，齐次坐标还可以区分点和向量，如在一个二维坐标系中坐标(x,y)可以表示一个二维点和二维向量，如果我们用其次坐标表示成(x,y,ω)时，如果ω=0，即(x,y,0)，则表示这个一个向量，因为$y' = \frac{y}{\omega} = \frac{y}{0} = \infty$，$x' = \frac{x}{\omega} = \frac{x}{0} = \infty$,这个点表示 x 和 y 都是无穷远点，并且 x 和 y 的增长率还不一样。如果将该坐标用于矩阵运算，则可以发现矩阵中的平移因子对该点无效，缩放和旋转却可以作用到该点上面，这也和向量不能进行平移，但是可以缩放旋转的结果一致，这些都是我们使用齐次坐标的理由。

4.2 透视投影

4.2.1 二维透视投影

相机的成像和人眼睛成像原理类似，都是凸透镜成像的原理，如果在二维空间上投影，则类似于图 4-1 所示。

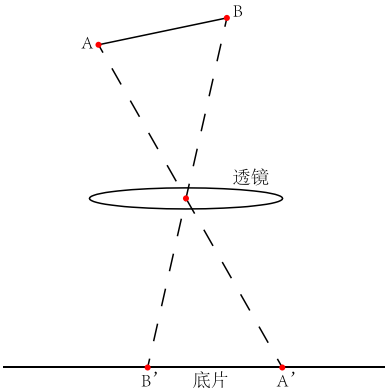


图 4-1 凸透镜成像

而 OpenGL 和 Direct3D 的投影有细微的差别，他们采用类似图 4-2 的投影方式：

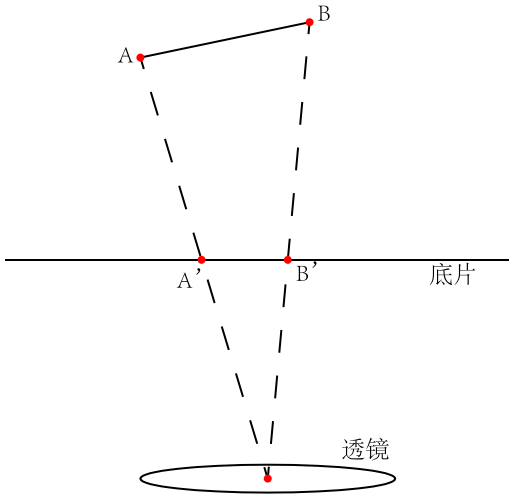


图 4-2 类似 OpenGL 和 D3D 的投影

我们可以发现，在 OpenGL 和 D3D 的投影方式中，底片和透镜的位置被交换了，但是这并不会得到不同的结果。

我们将图 4-2 作一些处理，变成图 4-3 的样子：

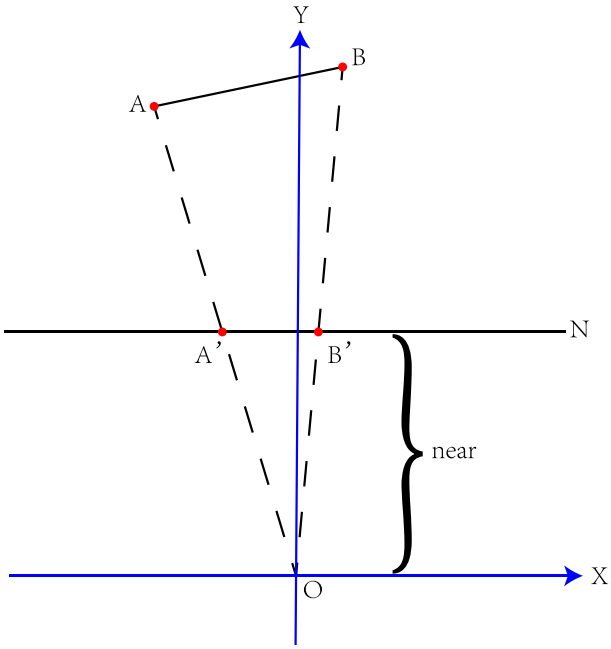


图 4-2 相机坐标系

我们以相机凸透镜的中心 O 为原点，过原点作一条和底片平行的直线作为 X 轴，过原点作一条和 X 轴垂直的直线作为 Y 轴，得到的这个坐标系叫做相机坐标系。因为底片和 X 轴平行，所以我们主要知道底片和原点 O 的距离就可以确定底片的方程为 y=N，而底片和 O 的距离就叫做 near，底片叫做 N 直线，在 3D 投影中叫做 N 平面，也叫做近平面，因为下一章我们介绍 CVV 的时候还会介绍 far face（远平面）。也就是说如果我们知道了透视参数的 near 值(后面会简写成 n 或者 N)，就可以求出相机坐标系中任意一个坐标点在底片上的投影位置。

对于任意一点(x,y)，在底片上投影的位置为：

$$x' = \frac{nx}{y}$$

4.2.2 三维透视投影

在三维透视投影中，OpenGL 和 Direct3D 采用类似如下的相机坐标系：

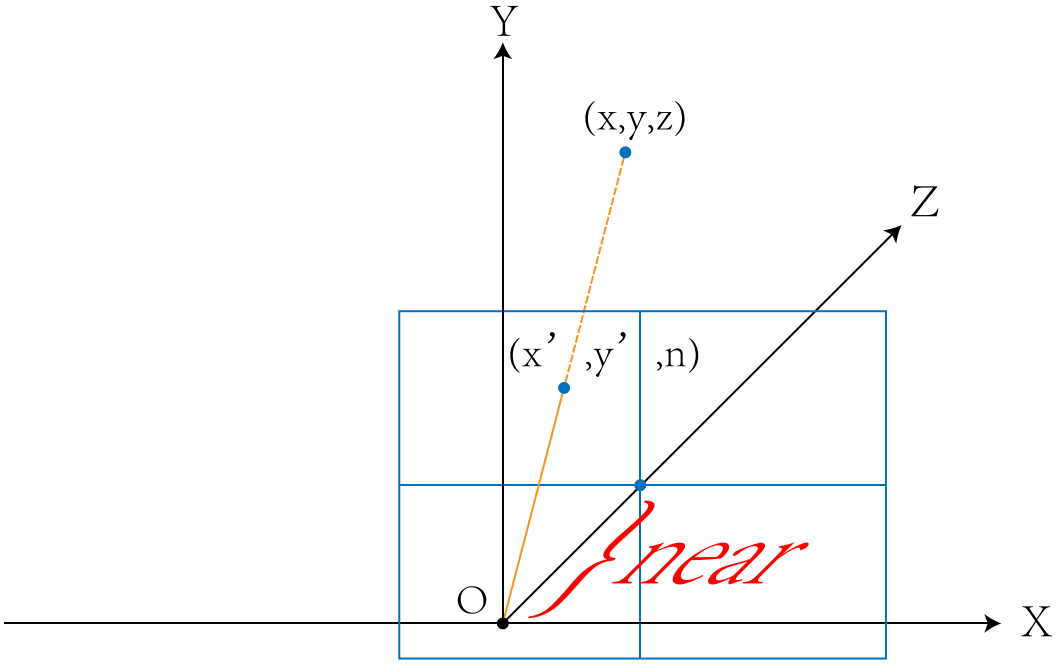


图 4-3 透视投影示意图

和 2D 透视投影类似，我们创建一个底片平行于 YOX 平面，底片和原点 O 的距离为 near(后面会简写成 n 或者 N)，我们把底片所在的平面称为 near 平面。则任意相机坐标系中的任意一点(x,y,z)在 n 平面投影的的坐标计算公式为:

$$x' = \frac{xn}{z}$$

$$y' = \frac{yn}{z}$$

坐标(x,y,z)经过透视投影后的坐标为$(\frac{xn}{z},\frac{yn}{z},n)$，但是 z 分量暂时没什么意义，因为所有的点经过投影之后 z 分量都是 n，所以我们可以设置为 any，即坐标(x,y,z)经过透视投影后的坐标为$(\frac{xn}{z},\frac{yn}{z},any)$。并且我们可以很容易的证明：空间中的直线经过透视投影变换之后在平面上仍然是一条直线，所以对于一条线性，我们只需要计算两个端点投影之后的屏幕坐标，然后在屏幕上将其连接起来即可。

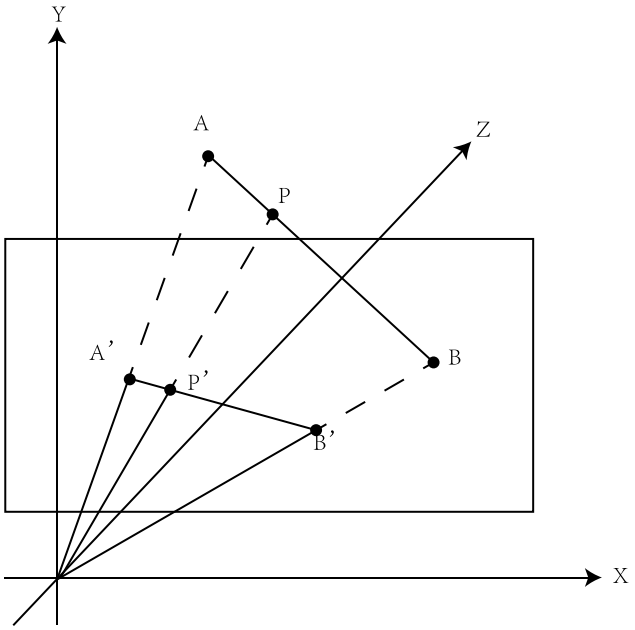


图 4-4 直线投影之后仍然为直线的证明图示

如上图 4-4 所示在空间中有 AB 两点，其在平面上的投影为 A'B'，我们可以看出 AB 原点 O 构成了一个三角形，而线段 A'B'为三角形和屏幕平面的相交直线，所有在△AOB 上面的点经过投影之后都会落在△AOB 和屏幕平面相交的直线上。

4.2.3 用齐次坐标表示透视投影

在上节中我们知道一个点(x,y,z)经过投影之后变为$(\frac{xn}{z},\frac{yn}{z},any)$，如果我们用其次坐标表示经过投影之后的点，则可以用$(\omega\frac{xn}{z},\omega\frac{yn}{z},\omega*any,\omega)$，如果令ω=z，则该齐次坐标点可以写成$(nx,ny,any,z)$，因为 any*z=any。

我们把上节投影变换的公式修改为：

$$x' = xn$$
$$y' = yn$$
$$z' = any$$
$$\omega' = z$$

通过这种表示方法，我们可以用一个齐次坐标表示出投影，同时还能附带一些额外信息，如ω分量就携带了顶点的原始信息，这个ω分量在进行透视校正插值和裁剪的时候很有用。还有一个有意思的地方就是如果使用齐次坐标表达投影，经过透视变换，我们不仅仅进行了投影，还把这个三维坐标变成了四维坐标。

读者一定要清楚的认识到：如果单论透视投影变换，我们用普通的透视投影公式和齐次坐标这两种表达方式都没问题，但是因为齐次坐标可以携带一些额外信息，这些信息在后续处理中可以简化我们对问题的分析和处理，所以后面章节的投影我都将会使用齐次坐标的表达形式。

本章主要介绍一些透视变换的数学知识，所以没有代码。

第5章 透视校正插值

5.1 空间三角形的绘制

本节代码在\Part2\src\chapter5\normal_perspective 中

通过上一章介绍的透视投影，可能有的读者已经跃跃欲试了，想要把一个三角形投影到屏幕上，然后绘制出来，我们在本章中可以开始尝试这样做一下。

我们在第二部分不在使用 Part1 的 lib 代码，因为里面有些东西是为了图形学入门而准备的，如 Point2 在三维处理中就已经很明显不够用了，所以我们在 Part2 中完全放弃了 Point2，将使用三维点的齐次坐标 Point4 来处理。其他代码几乎也都进行了少量的修改，从现在开始，每个小节我们都会对 lib 代码进行更新，每个小节的 lib 代码都和其他代码不一样，并且放在了章节目录中。

在渲染程序中，对顶点坐标的变换通常都是由用户来做的，这个变换程序叫做 Vertext Shader，简称 VS。我们在 Main.cpp 中定义了一个函数 Perspective，用于将三维坐标进行透视投影变换，这段代码非常的简单，先把一个三维坐标的齐次坐标规范化成标准的三维坐标，然后进行透视变换。

在 GraphicsLibrary 中新增一个函数 DrawTriangle，这个函数接受三个 Point4 和一个 ValueLength，这三个顶点是经过透视投影变换之后的坐标。我们可以看到这个函数除了函数开始处增加了对 Point4 规范化之外，和 Part1 的 MultipleInterpolationArrayIn2D 函数几乎一模一样，需要注意的是我们三角形是在二维屏幕上绘制的，所以该点的 z 和ω分量暂时是没有意义的，我们可以任意设置，我们在屏幕上只需要 x 和 y。我们在 main 函数中分别创建了 a,b,c 三个顶点，然后进行透视变换，得到 ta,tb,tc 三个结果顶点，然后把 ta,tb,tc 绘制到屏幕上，运行程序得到如下的结果。

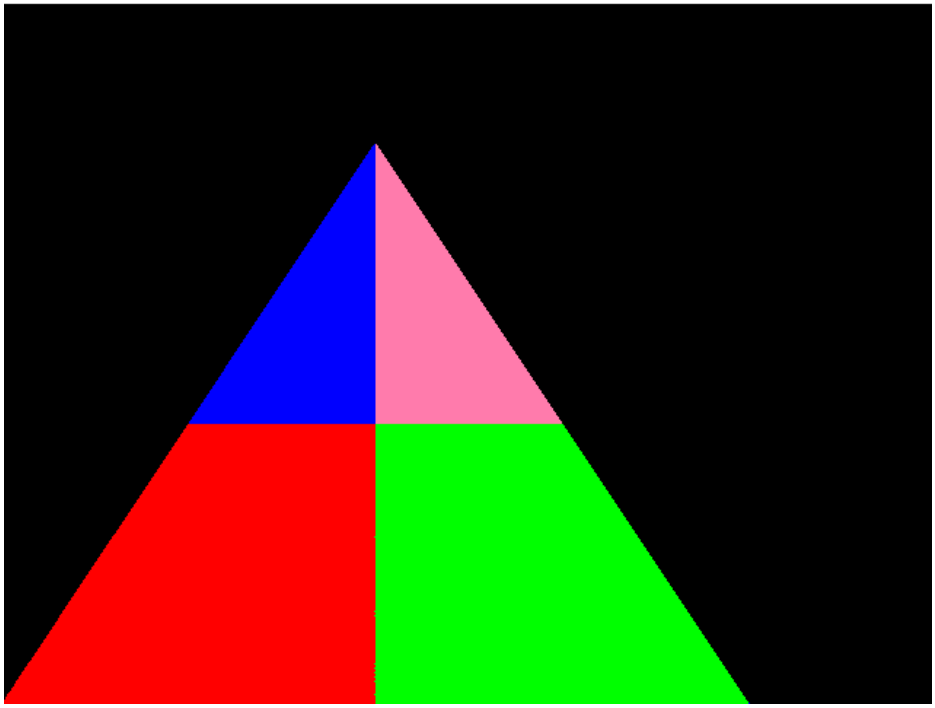


图 5-1

5.2 绘制两个三角形拼接成一个正方形

本节代码在 Part2\src\chapter5\two_triangle 中。

我们完全复制 5.1 的 lib 代码，只把 main 函数做轻微的修改，先绘制第一个三角形，然后绘制第二个三角形，这样就可以在屏幕上面绘制出一个完整的正方形了。我们在 main 函数中绘制完第一个三角形后又增加了一组三角形，只是把三角形的坐标和纹理坐标改变了一下，绘制完两个三角形后就得到一个正方形。

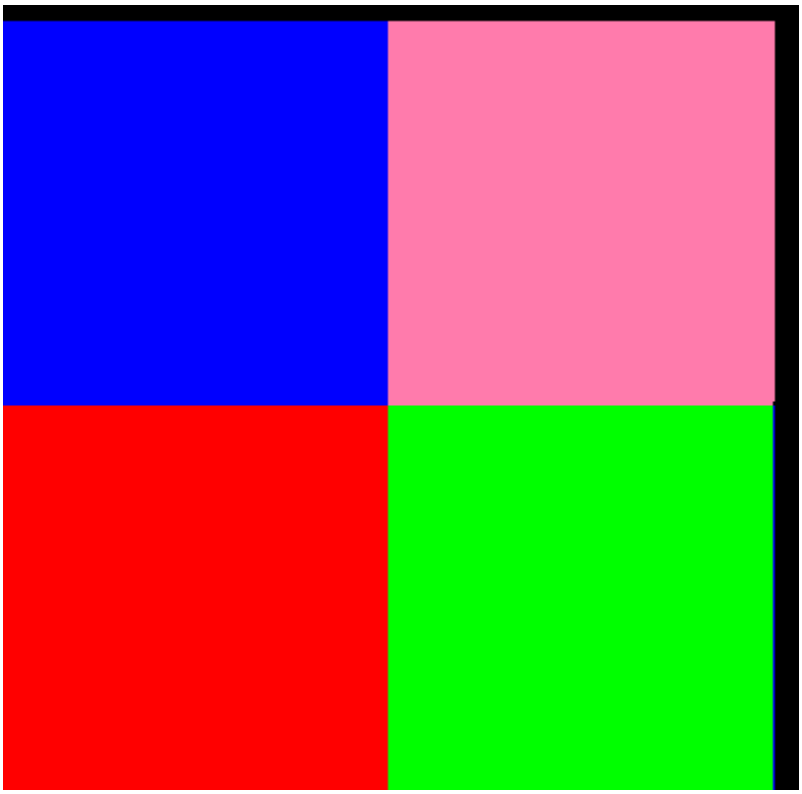


图 5-2 两个三角形拼接成一个正方形

5.3 仿射变换导致的错误

我们在上面两个小节中对纹理属性的插值方法是错误的，只是刚好被投影的平面和 near 平面平行，在这种情况下，我们在屏幕空间进行线性插值刚好得到正确的结果，如果我们让两个三角形和 near 平面有一点点夹角，那么立马就能发现错误。同样的，我们几乎把 two_triangle 的代码完整的复制过来，修改成 err_interpolation 中的代码，仅仅改变了两个三角形的坐标，让三角形和 near 平面成 30°的夹角，我们来看看结果，如图 5-3 所示，在图形的中间，横着的色块边界本来应该是一条直线的，但是我们在这里看到，边界变成了直线，这个错误的原因我们将会在下一小节解释说明。

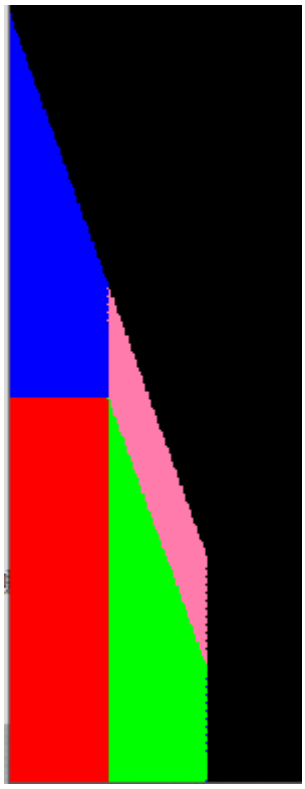


图 5-3 直接在屏幕空间插值导致的错误结果

5.4 透视校正插值

这里我先解释两个词，原始空间：指的是没有经过投影变换的坐标系，屏幕空间：屏幕上的坐标系，我们可以看到不管是 2D 投影还是 3D 投影，投影到屏幕空间之后都会丢失一个维度。我们先考虑一下二维情况下的透视投影，如下图 5-4 所示：

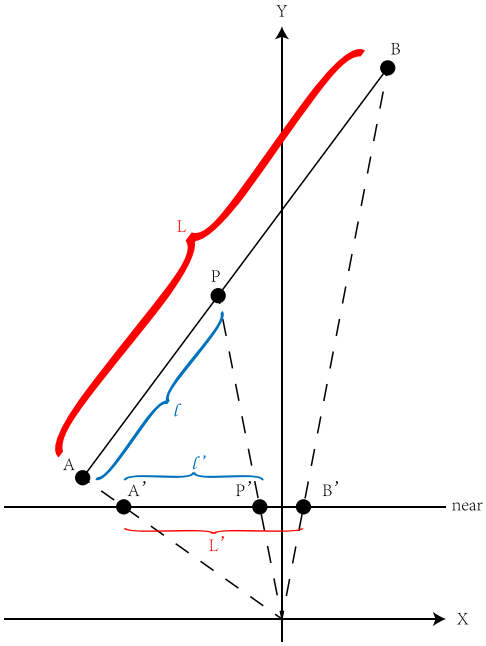


图 5-4 屏幕空间和原始空间比例不一致

我们通过线性插值的知识知道对于 P 来说，B 点的权值$w_B = \frac{l}{L}$，线段 AB 投影到屏幕上之后的线段为 A'B'，P 点投影之后的点为 P'，在屏幕上的 B'点权值为 $w_{B'} = \frac{l'}{L'}$，我们之前的程序是在线段 A'B'上面进行线性插值，很容易发现：$\frac{l}{L} \neq \frac{l'}{L'}$，即$w_B \neq w_{B'}$，而且$w_B$才是真正需要计算的结果，而$w_{B'}$是一个错误的结果，这就导致了我们的纹理插值出现错乱。

我们设在真实空间中对于 P 点的$w_B = t$,投影到屏幕上点 P'的$w_{B'} = t'$，则我们可以得到如下方程(对于下面的所有方程，a 表示原始空间中的点 a，a'表示 a 点投影到屏幕后的投影点，其他类似)：

$$x_{p'} = \frac{nx_p}{y_p} = \frac{n[(1-t)x_a + tx_b]}{(1-t)y_a + ty_b} \qquad ①$$

$$x_{p'} = (1-t')x_{a'} + t'x_{b'} \qquad ②$$

$$x_{a'} = \frac{nx_a}{y_a} \qquad ③$$

$$x_{b'} = \frac{nx_b}{y_b} \qquad ④$$

联立方程①②③④可以得到：

$$\frac{n[(1-t)x_a + tx_b]}{(1-t)y_a + ty_b} = (1-t')\frac{nx_a}{y_a} + t'\frac{nx_b}{y_b} \qquad ⑤$$

因为 t 是原始空间的权值对于线性插值是有效的，而 t'是屏幕空间的权值，我们当然希望解出一个$t = f(t')$的函数来，这样就可以在屏幕空间计算出原始空间的权值，最后解出(推导过程见附录一)：

$$t = \frac{y_a t'}{y_b + (y_a - y_b)t'} \qquad ⑤$$

我们在上一章有说过，我们使用齐次坐标计算投影时用ω分量记录原始顶点的原始深度值，也就是上面方程的y_a和y_b可以写成ω_a和ω_b，即上述方程可以写成

$$t = \frac{\omega_a t'}{\omega_b + (\omega_a - \omega_b)t'} \qquad ⑦$$

并且如果在原始空间中的直线上有一线性变化的属性值 V，通过第三章的学习我们很容易得到如下等式：

$$\frac{V_p - V_a}{V_b - V_a} = \frac{x_p - x_a}{x_b - x_a} = \frac{y_p - y_a}{y_b - y_a} = \frac{\omega_p - \omega_a}{\omega_b - \omega_a} = t$$

通过上式我们可以计算出V_p, x_p, y_p, ω_p等值，我们先计算一下V_p

$$\frac{V_p - V_a}{V_b - V_a} = t$$

将⑦带入上式等号右侧的 t 中，得到：

$$\frac{V_p - V_a}{V_b - V_a} = \frac{\omega_a t'}{\omega_b + (\omega_a - \omega_b)t'}$$

最终解出(推导过程见附录二)：

$$V_p = \frac{\omega_b V_a + \omega_a V_b t' - \omega_b V_a t'}{\omega_b + \omega_a t' - \omega_b t'} \qquad ⑧$$

虽然上图中我们举例子的图片是用二维直线投影到一维屏幕进行解释，但是从三维空间投影到二维屏幕也是一样的情况，上一章介绍到在相机坐标系下，三维点的投影计算公式为

$$x' = \frac{nx}{z}$$

$$y' = \frac{ny}{z}$$

我们不管是用$x' = \frac{nx}{z}$还是$y' = \frac{ny}{z}$计算都会得到一致的结果，并且$x' = \frac{nx}{z}$使用于二维和三维投影，说明我们上面的计算在三维投影的情况下同样成立。

如果读者还有印象的话，应该会记得我们在屏幕上绘制三角形的时候，会先将三角形切分成上下两个三角形，然后分别绘制，如下图:

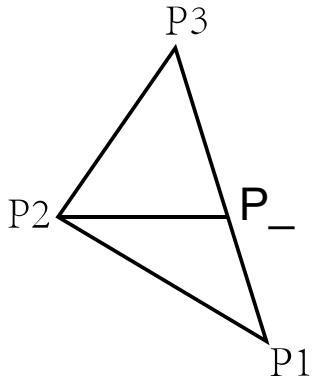


图 5-5 三角形的切分

我们在绘制三角形的时候会使用线性插值计算 P_点的坐标和属性值，但是通过上述知识，我们知道使用线性插值计算出来的点 P_的坐标和属性值可能不再正确。假如已知空间三角形△ABC 经过投影变换后，投影到屏幕上成为了△ $P_1P_2P_3$，然后我们在屏幕空间使用△ $P_1P_2P_3$来绘制图案。现在设 P_的齐次坐标为$P'(x_{p'}, y_{p'}, n, \omega_{p'})$，该点的 x,y 分量是在屏幕空间上面的坐标点，所以直接在屏幕空间进行线性插值是没有问题的，第三个分量暂时没有意义，第四个分量ω用于计算正确的属性值，所以我们必须正确计算出 P_的ω(原始深度)。我们在说属性校正插值的时候提到过下面这个等式:

$$\frac{V_p - V_a}{V_b - V_a} = \frac{x_p - x_a}{x_b - x_a} = \frac{y_p - y_a}{y_b - y_a} = \frac{\omega_p - \omega_a}{\omega_b - \omega_a} = t$$

我们可以用$\frac{\omega_p - \omega_a}{\omega_b - \omega_a} = t$和$t = \frac{\omega_a t'}{\omega_b + (\omega_a - \omega_b)t'}$计算出屏幕空间上任意一点的ω值，我们可以推导出如下方程(具体推导过程见附录三):

$$\omega_p = \frac{\omega_a \omega_b}{\omega_b + \omega_a t' - \omega_b t'} \qquad ⑨$$

更直观的表达式为:

$$\frac{1}{\omega_p} = \frac{1}{\omega_a} + (\frac{1}{\omega_b} - \frac{1}{\omega_a})t' \qquad ⑩$$

大部分的资料介绍透视校正插值的时候都会给出下面两个公式:

$$\frac{1}{\omega_p} = \frac{1}{\omega_a} + (\frac{1}{\omega_b} - \frac{1}{\omega_a})t' \quad ㈠$$

$$\frac{V_p}{\omega_p} = \frac{V_a}{\omega_a} + (\frac{V_b}{\omega_b} - \frac{V_a}{\omega_a})t' \quad ㈡$$

先用㈠计算出ω_p的值，再用㈡计算出V_p的值，从而得到正确的插值结果，实际上如果我们把㈠带入㈡中，可以推导到和⑨一样的结果(推导过程见附录四)，可能有一些读者看过其他相关文献，为了不使他们迷惑，特意作此说明。

接下来我们就要使用透视校正插值进行正确的纹理插值，相关代码见 Part2/src/chapter5/Perspective_Correct_Interpolation。我们给 Point4 新增了一个函数 Normalize_special，这个函数在规范化齐次坐标的时候对ω分量进行了一点小操作，不将其设置为 1，而是进行了原样保留，x,y,z 三个分量则和普通的规范化一样，都是使用该分量除以ω。这次我们在 DrawTriangle 函数开始的地方使用了 Normalize_special 函数，这样三个顶点都保留了ω值，在计算 P 点的时候使用公式⑨计算出其ω值。

我们在计算每个像素的属性值的时候也会有一些变换了，以前我们在计算属性值 V 的时候，先计算其在纵向扫描线上的增量，然后在填充一行扫描线的时候再计算其在 X 方向的增量，这样可以快速的使用增量来得到正确的结果。

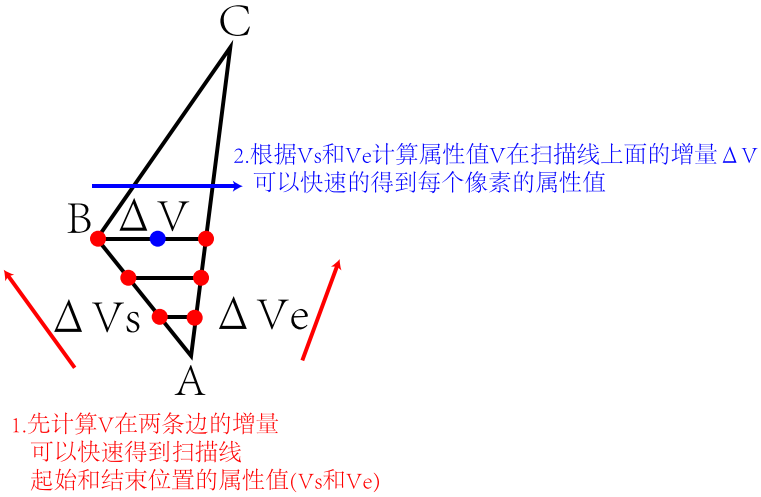


图 5-6 未使用透视校正插值时的属性计算方法

首先按照本节前面所描述的，我们在计算 P_点的ω和属性值的时候不再使用线性插值计算。

GraphicsLibrary.cpp 中的第 60 行和 65 行分别计算ω值和每个属性的值

```
60: double p_omega = (pa->W * pc->W) / (pc->W + (pa->W - pc->W) * p_t);//计算 P'点的 ω 值
65: p.ValueArray[i] = (pc->W * pa->ValueArray[i] + (pa->W * pc->ValueArray[i] - pc->W * pa->ValueArray[i]) * p_t) / (pc->W + (pa->W - pc->W) * p_t);
```

因为我是把一个三角形切分成上下两个三角形来绘制的，所以很多变量我都会写两个并且把他们合并成一个长度为 2 的数组，如 DXleft[2]等等，下面我就只介绍绘制下半三角形的的过程，因为绘制上半三角形的过程完全一样。

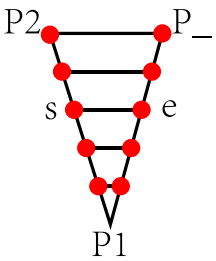


图 5-7 透视校正插值

上图中 s 和 e 表示扫描线的起始和结束点，我们在计算 Vs、Ve、和 V 的时候不能直接得到其增量，即ΔVs、ΔVe 和ΔV 不再是定值，但是 t'的增量还是定值，也算是一个小安慰了。我们先计算 t'在三角形左右两条边上的增量Δt'，将其存储于 dt_left 和 dt_right 中，用于快速计算每条扫描线起始和结束点在边上的比例值 t'。

```
88: double dt_left[] = { 1 / (P2.Y - P1.Y),1 / (P3.Y - P2.Y) };
```

```
double dt_right[] = { 1 / (P_.Y - P1.Y),1 / (P3.Y - P_.Y) };
```

在每条扫描线绘制之前使用透视校正插值计算出点 s 和 e 的ω值:

```
double sw = (lwa * lwb) / (lwb + (lwa - lwb) * t_left);
double ew = (rwa * rwb) / (rwb + (rwa - rwb) * t_right);
```

同样的使用透视校正插值计算 s 和 e 的属性值:

```
svs[vindex] = (lwb * lva + (lwa * lvb - lwb * lva) * t_left) / (lwb + (lwa - lwb) * t_left);
evs[vindex] = (rwb * rva + (rwa * rvb - rwb * rva) * t_right) / (rwb + (rwa - rwb) * t_right);
```

然后在绘制每个像素的时候，使用 dt 和 t 快速计算该像素在扫描线 s、e 中间的比例:

```
double dt = 1 / (ex - sx);//t 在扫描线上面的变化率
double t = 0;
```

接着我们用 svs、evs 和 t 在扫描线上使用透视校正插值计算出正确的属性值 vs:

```
vs[vindex] = (wb * va + (wa * vb - wb * va) * t) / (wb + (wa - wb) * t);
```

我们可以看到，这个程序得到了正确的纹理属性，纹理不再变形。

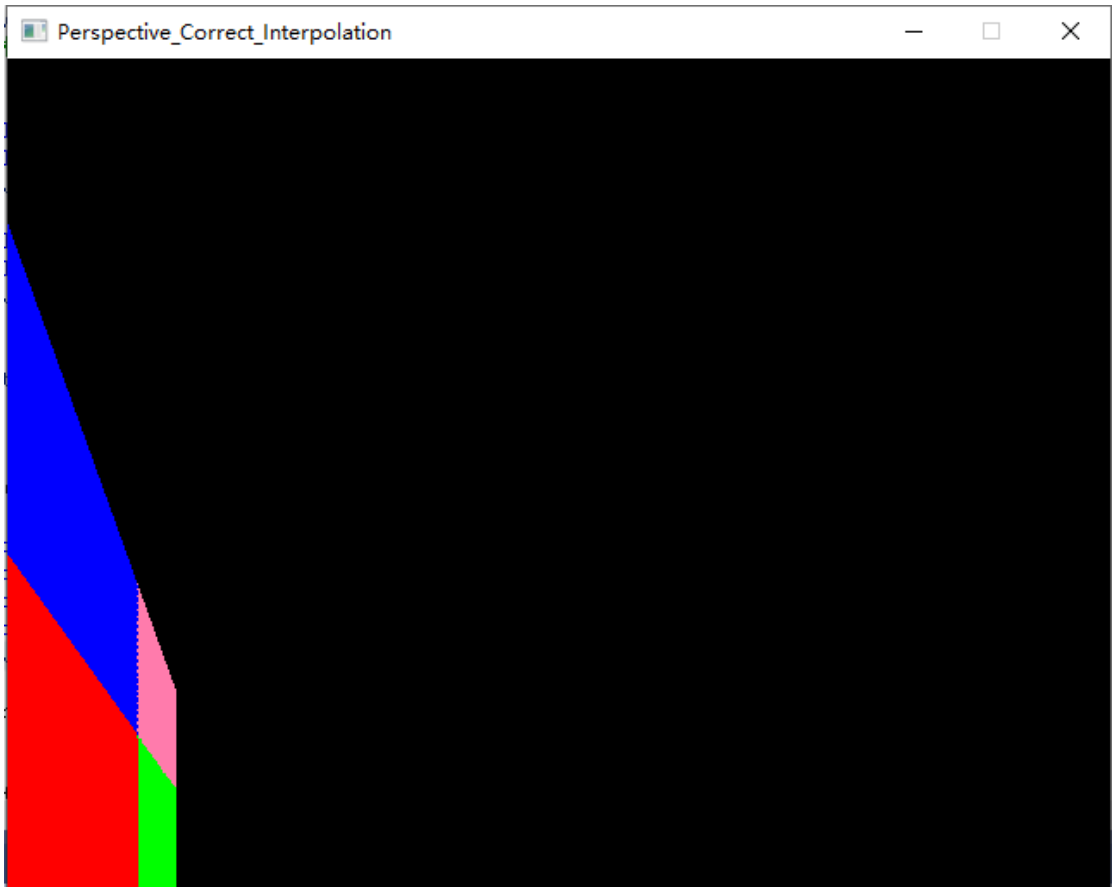


图 5-8 使用透视校正插值得到正确的纹理结果

5.5 附录

5.5.1 附录一

从 $\frac{n[(1-t)x_a+tx_b]}{(1-t)y_a+ty_b}=(1-t')\frac{nx_a}{y_a}+t'\frac{nx_b}{y_b}$ 到 $t=\frac{t'}{\frac{y_b}{y_a}+(1-\frac{y_b}{y_a})t'}$的推导

因为做题习惯不一样，所以我的推导顺序不一定和大家的一样，但是可以做一个参考，我推了得头都晕了。

$$\frac{n[(1-t)x_a+tx_b]}{(1-t)y_a+ty_b}=(1-t')\frac{nx_a}{y_a}+t'\frac{nx_b}{y_b}$$

$$\frac{x_a+t(x_b-x_a)}{y_a+t(y_b-y_a)}=\frac{x_a}{y_a}+t'(\frac{x_b}{y_b}-\frac{x_a}{y_a})$$

$$x_a+x_bt-x_at=\left[\frac{x_a}{y_a}+\frac{x_b}{y_b}t'-\frac{x_a}{y_a}t'\right][y_a+y_bt-y_at]$$

$$x_a+x_bt-x_at=\left[\frac{x_a}{y_a}+\frac{x_b}{y_b}t'-\frac{x_a}{y_a}t'\right][y_a+y_bt-y_at]$$

$$x_a+x_bt-x_at=\frac{x_a}{y_a}y_a+\frac{x_b}{y_b}y_at'-\frac{x_a}{y_a}y_at'+\frac{x_a}{y_a}y_bt+\frac{x_b}{y_b}y_btt'-\frac{x_a}{y_a}y_btt'-\frac{x_a}{y_a}y_at-\frac{x_b}{y_b}y_att'+\frac{x_a}{y_a}y_att'$$

$$x_a+x_bt-x_at=x_a+\frac{x_b}{y_b}y_at'-x_at'+\frac{x_a}{y_a}y_bt+x_btt'-\frac{x_a}{y_a}y_btt'-x_at-\frac{x_b}{y_b}y_att'+x_att'$$

$$x_bt=\frac{x_b}{y_b}y_at'-x_at'+\frac{x_a}{y_a}y_bt+x_btt'-\frac{x_a}{y_a}y_btt'-\frac{x_b}{y_b}y_att'+x_att'$$

$$x_bt-\frac{x_a}{y_a}y_bt-x_btt'+\frac{x_a}{y_a}y_btt'+\frac{x_b}{y_b}y_att'-x_att'=\frac{x_b}{y_b}y_at'-x_at'$$

$$t\left(x_b+\frac{x_a}{y_a}y_bt'+\frac{x_b}{y_b}y_at'-\frac{x_a}{y_a}y_b-x_bt'-x_at'\right)=\frac{x_b}{y_b}y_at'-x_at'$$

$$t=\frac{\frac{x_b}{y_b}y_at'-x_at'}{x_b+\frac{x_a}{y_a}y_bt'+\frac{x_b}{y_b}y_at'-\frac{x_a}{y_a}y_b-x_bt'-x_at'}$$

$$t=\frac{(\frac{x_b}{y_b}y_a-x_a)t'}{x_b-\frac{x_a}{y_a}y_b+(\frac{x_a}{y_a}y_b+\frac{x_b}{y_b}y_a-x_a-x_b)t'}$$

$$t=\frac{t'}{\frac{x_b-\frac{x_a}{y_a}y_b}{\frac{x_b}{y_b}y_a-x_a}+\frac{\frac{x_a}{y_a}y_b+\frac{x_b}{y_b}y_a-x_a-x_b}{\frac{x_b}{y_b}y_a-x_a}t'}$$

把分母全部换成y_ay_b，看起来是不是有点恐怖，就是简单的因式分解和化简，别害怕

$$t=\frac{t'}{\frac{\frac{x_by_ay_b}{y_ay_b}-\frac{x_ay_by_b}{y_ay_b}}{\frac{x_by_ay_a}{y_ay_b}-\frac{x_ay_ay_b}{y_ay_b}}+\frac{\frac{x_ay_by_b}{y_ay_b}+\frac{x_by_ay_a}{y_ay_b}-\frac{x_ay_ay_b}{y_ay_b}-\frac{x_by_ay_b}{y_ay_b}}{\frac{x_by_ay_a}{y_ay_b}-\frac{x_ay_ay_b}{y_ay_b}}t'}$$

$$t=\frac{t'}{\frac{x_b y_a y_b - x_a y_b y_b}{x_b y_a y_a - x_a y_a y_b}+\frac{x_a y_b y_b + x_b y_a y_a - x_a y_a y_b - x_b y_a y_b}{x_b y_a y_a - x_a y_a y_b}t'}$$

$$t=\frac{t'}{\frac{y_b(x_b y_a - x_a y_b)}{y_a(x_b y_a - x_a y_b)}+[\frac{x_b y_a y_a - x_a y_a y_b}{x_b y_a y_a - x_a y_a y_b}+\frac{y_b(x_a y_b - x_b y_a)}{y_a(x_b y_a - x_a y_b)}]t'}$$

$$t=\frac{t'}{\frac{y_b}{y_a}+(1-\frac{y_b}{y_a})t'}$$

$$t=\frac{y_a t'}{y_b+(y_a-y_b)t'}$$

5.5.2 附录二

V_p的推导:

$$\frac{V_p - V_a}{V_b - V_a} = \frac{\omega_a t'}{\omega_b + (\omega_a - \omega_b)t'}$$

$$V_p - V_a = \frac{\omega_a t'(V_b - V_a)}{\omega_b + \omega_a t' - \omega_b t'}$$

$$V_p - V_a = \frac{\omega_a V_b t' - \omega_a V_a t'}{\omega_b + \omega_a t' - \omega_b t'}$$

$$V_p = \frac{\omega_a V_b t' - \omega_a V_a t'}{\omega_b + \omega_a t' - \omega_b t'} + V_a$$

$$V_p = \frac{\omega_a V_b t' - \omega_a V_a t'}{\omega_b + \omega_a t' - \omega_b t'} + \frac{\omega_b V_a + \omega_a V_a t' - \omega_b V_a t'}{\omega_b + \omega_a t' - \omega_b t'}$$

$$V_p = \frac{\omega_a V_b t' - \omega_a V_a t' + \omega_b V_a + \omega_a V_a t' - \omega_b V_a t'}{\omega_b + \omega_a t' - \omega_b t'}$$

$$V_p = \frac{\omega_b V_a + \omega_a V_b t' - \omega_b V_a t' + \omega_a V_a t' - \omega_a V_a t'}{\omega_b + \omega_a t' - \omega_b t'}$$

$$V_p = \frac{\omega_b V_a + \omega_a V_b t' - \omega_b V_a t'}{\omega_b + \omega_a t' - \omega_b t'}$$

5.5.3 附录三

使用$\frac{\omega_p-\omega_a}{\omega_b-\omega_a}=t$ 和 $t=\frac{\omega_a t'}{\omega_b+(\omega_a-\omega_b)t'}$ 对ω_p的推导

$$\frac{\omega_p-\omega_a}{\omega_b-\omega_a}=t$$

将$t=\frac{\omega_a t'}{\omega_b+(\omega_a-\omega_b)t'}$带入上式等号右侧

$$\frac{\omega_p-\omega_a}{\omega_b-\omega_a}=\frac{\omega_a t'}{\omega_b+(\omega_a-\omega_b)t'}$$

$$\omega_p-\omega_a=\frac{\omega_a\omega_b t'-\omega_a\omega_a t'}{\omega_b+\omega_a t'-\omega_b t'}$$

$$\omega_p=\frac{\omega_a\omega_b t'-\omega_a\omega_a t'}{\omega_b+\omega_a t'-\omega_b t'}+\frac{\omega_a\omega_b+\omega_a\omega_a t'-\omega_a\omega_b t'}{\omega_b+\omega_a t'-\omega_b t'}$$

$$\omega_p=\frac{\omega_a\omega_b t'-\omega_a\omega_a t'+\omega_a\omega_b+\omega_a\omega_a t'-\omega_a\omega_b t'}{\omega_b+\omega_a t'-\omega_b t'}$$

移动一下分子顺序，方便化简

$$\omega_p=\frac{\omega_a\omega_b+\omega_a\omega_b t'-\omega_a\omega_b t'+\omega_a\omega_a t'-\omega_a\omega_a t'}{\omega_b+\omega_a t'-\omega_b t'}$$

$$\omega_p=\frac{\omega_a\omega_b}{\omega_b+\omega_a t'-\omega_b t'}\text{(推导出⑨)}$$

$$\frac{1}{\omega_p}=\frac{\omega_b+\omega_a t'-\omega_b t'}{\omega_a\omega_b}$$

$$\frac{1}{\omega_p}=\frac{\omega_b}{\omega_a\omega_b}+\frac{\omega_a t'}{\omega_a\omega_b}-\frac{\omega_b t'}{\omega_a\omega_b}$$

$$\frac{1}{\omega_p}=\frac{1}{\omega_a}+(\frac{1}{\omega_b}-\frac{1}{\omega_a})t'\text{(推导出⑩)}$$

5.5.4 附录四

使用

$\frac{1}{\omega_p}=\frac{1}{\omega_a}+\left(\frac{1}{\omega_b}-\frac{1}{\omega_a}\right)t'$ 和 $\frac{V_p}{\omega_p}=\frac{V_a}{\omega_a}+\left(\frac{V_b}{\omega_b}-\frac{V_a}{\omega_a}\right)t'$ 推导出$V_p=\frac{\omega_b V_a+\omega_a V_b t'-\omega_b V_a t'}{\omega_b+\omega_a t'-\omega_b t'}$

因为:

$$\frac{1}{\omega_p}=\frac{1}{\omega_a}+\left(\frac{1}{\omega_b}-\frac{1}{\omega_a}\right)t'$$

$$\frac{1}{\omega_p}=\frac{\omega_b}{\omega_a\omega_b}+\frac{\omega_a t'}{\omega_a\omega_b}-\frac{\omega_b t'}{\omega_a\omega_b}$$

$$\frac{1}{\omega_p}=\frac{\omega_b+\omega_a t'-\omega_b t'}{\omega_a\omega_b}$$

$$\omega_p=\frac{\omega_a\omega_b}{\omega_b+\omega_a t'-\omega_b t'}$$

我们将$\omega_p=\frac{\omega_a\omega_b}{\omega_b+\omega_a t'-\omega_b t'}$带入$\frac{V_p}{\omega_p}=\frac{V_a}{\omega_a}+\left(\frac{V_b}{\omega_b}-\frac{V_a}{\omega_a}\right)t'$中得到:

$$\frac{V_p}{\frac{\omega_a\omega_b}{\omega_b+\omega_a t'-\omega_b t'}}=\frac{V_a}{\omega_a}+\left(\frac{V_b}{\omega_b}-\frac{V_a}{\omega_a}\right)t'$$

$$\frac{V_p}{\frac{\omega_a\omega_b}{\omega_b+\omega_a t'-\omega_b t'}}=\frac{\omega_b V_a}{\omega_a\omega_b}+\frac{\omega_a V_b t'}{\omega_a\omega_b}-\frac{\omega_b V_a t'}{\omega_a\omega_b}$$

$$\frac{V_p}{\frac{\omega_a\omega_b}{\omega_b+\omega_a t'-\omega_b t'}}=\frac{\omega_b V_a+\omega_a V_b t'-\omega_b V_a t'}{\omega_a\omega_b}$$

$$V_p=\frac{\omega_b V_a+\omega_a V_b t'-\omega_b V_a t'}{\omega_a\omega_b}\frac{\omega_a\omega_b}{\omega_b+\omega_a t'-\omega_b t'}$$

$$V_p=\frac{\omega_b V_a+\omega_a V_b t'-\omega_b V_a t'}{\omega_b+\omega_a t'-\omega_b t'}$$

第6章 深度测试

在日常生活中，我们可以很容易的知道在观察一个物体的时候，后面的图案会被前面的图案所遮挡，就像图 6-1 所示：

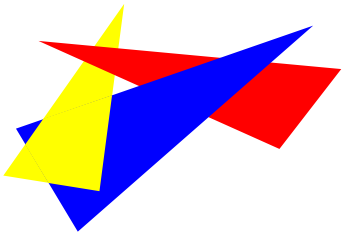


图 6-1 多边形的遮挡

在图 6-1 中，因为三个三角形各个位置和人眼睛的距离不一样，距离更远的部分就会被距离更近的所遮挡，这就导致了这些三角形相互遮挡。而处理这些遮挡关系的算法就称为消隐算法。下面我就介绍两种消隐算法：画家算法和 Z-Buffer 算法。

6.1 画家算法

画家算法处理处理遮挡问题的策略非常的简单：先绘制距离更远的物体，然后绘制距离更近的物体，这样距离远的物体应该被遮挡的部分就会被后面绘制的图案所覆盖。这种思想就像画油画，一层一层的，所以被叫做画家算法。

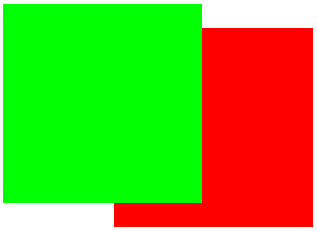


图 6-2 画家算法

在绘制之前对所有需要绘制的片面进行按距离排序，先绘制距离远的片面，然后绘制近的片面，对于上图 6-2 来说，先绘制红色正方形，然后绘制绿色正方形，这样就能得到正确的遮挡结果。但是画家算法也存在一些问题，如果被绘制的面和相机底片不是平行关系，那么距离应该怎么计算？如果同两个面有相交，那么画家算法就得不到正确的结果，如下图 6-3 所示：

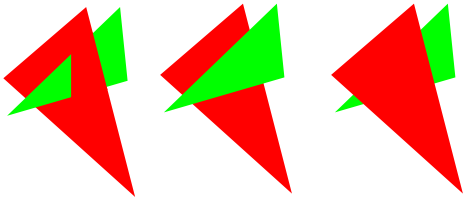


图 6-3 左：正确的渲染结果 中：先红后绿渲染顺序 右：先绿后红渲染顺序

可以看到，不管我们先绘制绿色三角形还是先绘制红色三角形，都得不到正确结果。

6.2 Z-Buffer 算法

Z-Buffer 算法可以得到正确的遮挡关系，其原理和画家算法则是两个不同的解题思路，在栅格化片面的时候，我们最终还是要对每一个需要处理的像素进行绘制。Z-Buffer 的思路则是为每一个像素设置一个缓冲区，记录该点的深度值，如果绘制当前像素的深度值小于记录的深度值，则将其绘制并更新缓冲区内的深度记录，如果绘制当前像素的深度值大于记录值，则放弃操作。

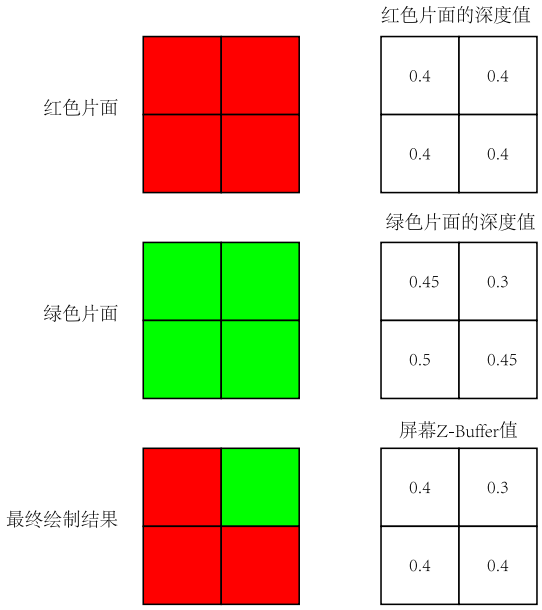


图 6-4 Z-Buffer 算法示意

假设我们的屏幕有 4 个像素，那么我们就创建一个 Z-Buffer 缓冲区，数量和像素数量一致。我们先绘制一个红色的片面，其四个像素的深度值都是 0.4，绘制完红色片面之后 Z-buffer 的值就和红色片面一致了，全部都是 0.4。然后再绘制第二个绿色片面，这个面有点倾斜，可以看到他右上角的深度值比较小，左下角深度值比较大，我们在逐个像素绘制绿色片面时，如果当前像素的深度值比 0.4 小就绘制，并更新 Z-Buffer，否则放弃该像素的绘制。可以发现绿色片面只有右上角被绘制了，其他都没有被绘制，因为其他像素相比红色片面更远，并且我们可以发现不论是先绘制绿色片面还是红色片面，都能得到正确的结果，这个深度对比的过程叫做深度测试。

通过上面的讨论，现在的重点就是怎么计算每个像素的深度值，这个函数一定要满足一个要求：该函数关于原始深度单调递增。我们在上一章将透视校正插值的时候知道可以通过公式$\omega_p = \frac{\omega_a \omega_b}{\omega_b + \omega_a t' - \omega_b t'}$得到该像素对应的原始深度值，原始深度自己本身很明显是关于自己单调递增的。读者应该还记得，我们在进行透视变换的时候，齐次坐标的 z 分量还没有被使用，我们可以在透视变换的时候把顶点的原始深度保存至 z 分量中，即采用下面这种变换公式：

$$x' = xn$$
$$y' = yn$$
$$z' = zz$$
$$\omega' = z$$

我们将(xn, yn, zz, z)的各个分量同时除以ω分量得到$(\frac{xn}{z}, \frac{yn}{z}, z, 1)$，可以看到这个公式中 z 分

量保存的就是原始深度值，使用$V_p = \frac{\omega_b V_a + \omega_a V_b t' - \omega_b V_a t'}{\omega_b + \omega_a t' - \omega_b t'}$进行透视校正插值之后和使用$\omega_p = \frac{\omega_a \omega_b}{\omega_b + \omega_a t' - \omega_b t'}$得到的结果是一样的。可能读者会有点疑惑：直接使用$\omega_p$来作为深度值不就行了吗,这样做有什么意义？在下一章说裁剪和 CVV 的时候，我们会对投影公式再进行一个简单的修改，变成最终版的投影公式，所以这里就先让读者适应一下，我们先就做一下无用功吧。

对 z 分量插值的过程和对其他属性插值的过程一模一样，但是因为我想把坐标本身的值和属性值分开，所以里面的代码看起来有点重复多余。相关代码见 part2/chapter6/depth_test。我们先对 main.cpp 中的 Persective 函数进行小修改，将计算结果的 z 分量改为 z'=zz：

```
return Point4(pn.X * n, pn.Y * n, pn.Z * pn.Z, pn.Z);//执行透视变换
```

在创建 GraphicsLibrary 类的时候会创建一个和屏幕像素一样多的 double 数组 Z_Buffer:

```
Z_Buffer = new double[gd.height * gd.width];
```

同时增加一个函数 clean_depth(double v),因为在绘制一帧画面的时候，如果 Z-Buffer 里面原来的记录都很小的话，会导致我们新绘制的像素都被放弃，所以我们在绘制一帧画面的时候要先将深度值设置成为一个较大的值，这个值我们先设置为 1000，等到介绍 CVV 的时候，我们可以统一设置成为 1.0。

在 DrawTriangle 函数的开始处我们仍然使用 Normalize_special 来做规范化。

在 GraphicsLibrary.cpp 中，将三角形分割成上下两个三角形的点 P 也加上了 z 分量的计算，使用透视校正插值即可：

```
double p_z = (pc->W * pa->Z + (pa->W * pc->Z - pc->W * pa->Z) * p_t) / (pc->W + (pa->W - pc->W) * p_t);//使用透视校正插值计算 p 点的 Z 分量
```

我们在绘制每个像素的时候对深度进行一次测试，如果测试通过则更新深度值，否则放弃当前像素的绘制：

```
if (z > Z_Buffer[y*graphicsdevice.width+x])//深度测试不通过的话则放弃本像素的渲染
{
        continue;
}
else//否则更新深度值
{
        Z_Buffer[y * graphicsdevice.width + x] = z;
}
```

我们在 main 函数中绘制第一个像素之前先将 Z_Buffer 处理一遍：

```
gl.clean_depth(1000);//先将深度值设置为一个较大的值
```

然后在绘制完前两个三角形拼接成的正方形之后再绘制一个距离较远的正方形，运行程序可以得到如下的结果。

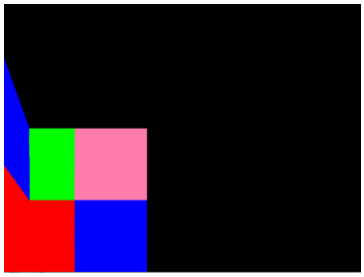


图 6-5 程序运行结果

第7章 裁剪

7.1 一些裁剪的基本知识

如果现在有一段曲线，我们需要保留一部分，那么去除我们不需要的这部分的过程就叫做裁剪。如下图 7-1:

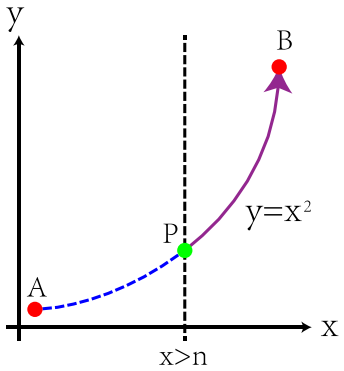


图 7-1 裁剪

假设我们有曲线$y = x^2$的一部分(上图中两个红色之间的部分)，设 A 点为曲线起点，B 点为曲线终点，现在我们使用一个直线$x = n$对其进行裁剪，并且保留$x > n$的部分，那么我们就能得到紫色实线这部分曲线，而虚线部分将被我们丢弃，绿色点就是新曲线的一个端点，如果该点是曲线进入裁剪边界的点，则称之为入点，反之，如果是曲线离开边界的点，则称之为出点。我们可以很容易的知道上图中曲线和边界的交点是一个入点，该入点坐标为(n, n^2)。需要注意的是：起点和终点的定义没有什么特殊要求，完全看编程人员的心情，但是如果是封闭多边形的边，我们通常会将多边形的相邻边首尾相连，这样更便于表达和处理，如下图 7-2:

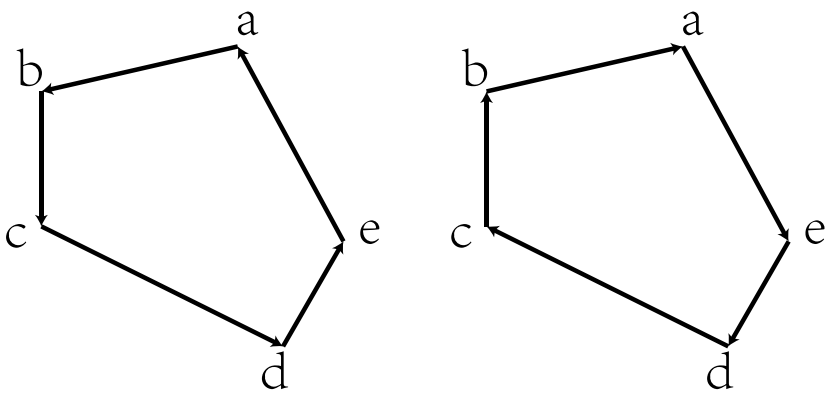


图 7-2 左逆时针多边形　右顺时针三角形

我们可以在上图左中看到，ab 边和 bc 边相邻，b 点为 ab 边的终点，同时也为 bc 边的起点，而在右图中 b 点则为 ab 边的起点，为 bc 边的终点。上面这种首尾相连的多边形表示方法更利于后面我们对多边形裁剪的讲解。

7.2 深度值的另外一种表示

现在假设有一条线段，我们对其裁剪，如下图 7-3:

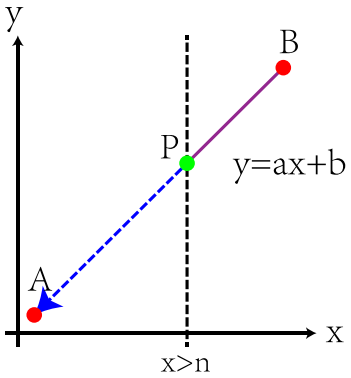


图 7-3 线段的裁剪

对于上面这种情况同样的很容易就可以计算出出点的坐标，设$t=\frac{P_x-A_x}{B_x-A_x}$，则

$$P_y=A_x+t(B_x-A_x)$$

当然上述公式也有失灵的时候，就是当$B_x=A_x$时分母为 0，所以后面我们还需要判断这个边界到底能不能对这条线段进行裁剪，只有边界条件能够对线段进行裁剪的时候，我们才对曲线进行裁剪。

我们应该还记得，曲线的端点往往还会携带一些属性，所以除了计算出点(入点)的坐标之外，还得计算其属性值。属性值的计算就得根据属性函数$v=f(p)$和出点(入点)p 计算了。曲线方程可以千变万化，属性值函数也可以变化无穷，而我们前面学习的知识都是关于线性插值的，所以我们可以做一些限定：被裁剪的曲线必须是直线的一部分(即线段)，属性值在该曲线上必须是线性均匀变化的。通过上述设定，我们就可以很容易的对出点或者入点的坐标或者属性进行计算，共三步：

① 判断该线段是否需要裁剪

② 计算边界交点的 b 点权值 t

③ 通过 t 计算出边界交点的坐标和属性值

一个线段需要裁剪必须满足如下情况：该线段一个端点满足边界条件，另一个端点不满足边界条件，如果以图 7-3 来说明的话就是，A 端点不满足边界条件$x>n$，而 B 端点满足边界条件$x>n$，这条线段就需要裁剪。边界和线段有如下三种关系：

平凡接受：两个端点都满足边界条件，不需要裁剪

平凡拒绝：两个端点都不满足边界条件，直接放弃本直线的绘制

裁剪：一个端点在边界内部，一个端点在边界外部，需要使用边界条件对线段进行裁剪，如果是起点在边界外部，则线段和边界的交点为入点，反之则为出点。

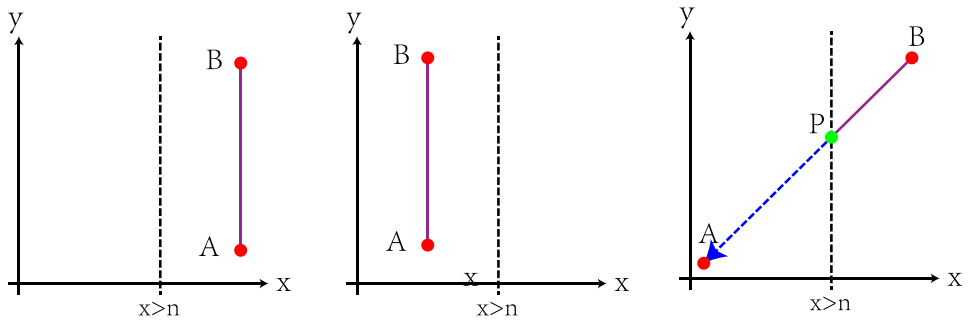


图 7-4 左：平凡接受　中：平凡拒绝　右：裁剪

可以看到，如果出现$B_x = A_x$这种情况时，该线段要么是平凡接受，要么是平凡拒绝，所以可以避免掉$B_x = A_x$这种情况的计算。

而第二步 t 的计算就很简单了，我们可以用下面这个公式：

$$\frac{V_p - V_a}{V_b - V_a} = \frac{x_p - x_a}{x_b - x_a} = \frac{y_p - y_a}{y_b - y_a} = \frac{z_p - z_a}{z_b - z_a} = \frac{\omega_p - \omega_a}{\omega_b - \omega_a} = t$$

计算出 t，并且我们可以看到，我们不仅仅可以用 x 计算，对于线段上任意线性变化的坐标或者属性，都可以辅助我们计算出 t，如：

已知裁剪边界为$\omega = n$，那么我们立马可以用公式$\frac{\omega_p - \omega_a}{\omega_b - \omega_a} = t$计算：$t = \frac{n - \omega_a}{\omega_b - \omega_a}$。第四章说过，如果使用齐次坐标表达投影，经过透视变换，我们不仅仅进行了投影，还把这个三维坐标变成了四维坐标(千万不要试图想象四维空间是什么样子，会想破脑袋的)，而不管对于多少维度的空间，如果该空间存在一条直线，那么直线上面的点必然满足如下方程：

$$\frac{x_p - x_a}{x_b - x_a} = \frac{y_p - y_a}{y_b - y_a} = \frac{z_p - z_a}{z_b - z_a} = \frac{\omega_p - \omega_a}{\omega_b - \omega_a} = \cdots = \frac{分量M_p - 分量M_a}{分量M_b - 分量M_a} = t$$

其中 ab 为直线上不在同一位置的两个点，直线就是这么神奇。还记得我们在上一章说改造的投影方程吗。

$$x' = xn$$
$$y' = yn$$
$$z' = zz$$
$$\omega' = z$$

原始空间中的一条直线 L 经过透视投影(还没做透视除法)得到的东西还是一条曲线吗，我们验证一下：设原始空间中有$A(x_a, y_a, z_a, 1)$和$B(x_b, y_b, z_b, 1)$两点，在直线上有一点$P(x_p, y_p, z_p, 1)$满足$\frac{x_p - x_a}{x_b - x_a} = \frac{y_p - y_a}{y_b - y_a} = \frac{z_p - z_a}{z_b - z_a} = t$，设 ABP 经过透视变换(还没有做透视除法)之后得到的点为 A'B'P'，则得到如下坐标：$A'(nx_a, ny_a, z_a z_a, z_a)$，$B'(nx_b, ny_b, z_b z_b, z_b)$，$P'(nx_p, ny_p, z_p z_p, z_p)$。我们发现这三个点的 x 分量满足如下关系：$\frac{nx_p - nx_a}{nx_b - nx_a} = \frac{x_p - x_a}{x_b - x_a} = t$，同时 y 和ω分量都得到一样的结果，但是 z 分量计算的时候就不同了：$\frac{z_p z_p - z_a z_a}{z_b z_b - z_a z_a}$根本就不能保证结果为 t，这说明在我们的投影算法中，一条三维直线经过投影之后得到的四维曲线不一定再是一条四维直线。

我们可以修改 z 分量的计算方法使得三维空间中的点经过透视变换之后变为四维点，并且三维空间中的直线在四维空间中仍然是直线。还记得我们在第六章的时候说过 z 分量的计算要遵循深度函数关于原始深度单调递增。所以我们可以构造一个线性的单调递增函数来替换之前的 z 分量计算，即将原来的$z' = zz$修改为$z' = az + b$，并且$\frac{z'}{z}$关于 z 单调递增，即$a + \frac{b}{z}$关于 z 单调递增。并且还有一个小细节就是 z>0，这个问题将在下节介绍近平面裁剪的时候会讲到我们可以使用裁剪来保证 z>0，这样即可保证这个深度函数是单调递增的了，在 z 大于 0 的情况下要让$a + \frac{b}{z}$关于 z 单调递增则只要 b<0 即可，a 可以取任意值。

经过修改，我们得到的新的投影公式为：

$$x' = xn$$
$$y' = yn$$
$$z' = az + b(b < 0)$$
$$\omega' = z$$

上述公式最后计算出来的深度值和原始深度不再是线性关系，所以也叫作伪深度，因为我们之前说过我们对深度函数的唯一要求就是这个函数要和原始深度单调递增，所以用这个公式来做深度测试是没有问题的。使用这种公式，三维空间中的直线经过透视变换到四维空间之后仍然是一条直线，又因为$\frac{V_p-V_a}{V_b-V_a} = \frac{x_p-x_a}{x_b-x_a} = \frac{y_p-y_a}{y_b-y_a} = \frac{z_p-z_a}{z_b-z_a} = \frac{\omega_p-\omega_a}{\omega_b-\omega_a} = t$，于是可以证明原先三维空间中的属性在四维空间中仍然是线性变化的，于是我们可以很方便的使用边界进行裁剪。

接下来的问题就是使用公式$z' = az + b$表示齐次坐标 z 分量的时候我们应该怎么计算每个像素的深度值，在屏幕空间上面使用线性插值还是透视校正插值呢？一个属性如果是在屏幕空间线性变化，我们就可以在屏幕空间使用线性插值计算；如果该属性在原始空间是线性变化的，则应该在屏幕空间使用透视校正插值；如果该属性不属于上述两种情况，那我们还没有研究过这个问题，暂时不予以讨论。还记得我们在透视校正插值中推导的公式⑩吗？

$$\frac{1}{\omega_p} = \frac{1}{\omega_a} + (\frac{1}{\omega_b} - \frac{1}{\omega_a})t'$$

我们这个公式中的ω就是原始空间中的 z，也就是在屏幕空间中$\frac{1}{z}$是线性变化的，我们根据这个公式可以得到$\frac{z'}{z} = a + \frac{b}{z}$在屏幕空间是线性变化的(推导见本章附录一)，所以我们可以在屏幕空间使用线性插值计算深度值。

我们将前一章的代码进行修改，在屏幕空间中使用线性插值计算"伪深度"，代码见 part2/chapter7/depth_reciprocal。

我们首先要做的就是修改透视投影的计算，我们将 main.cpp 中的 Persective 函数进行修改，还记得上面我们说过的，$z' = az + b$，并且我们只要求 b<0，对 a 没有要求，所以我们简单而且粗暴的令 a=0,b=-1，将返回值的 z 分量直接改为-1。

```
return Point4(pn.X * n, pn.Y * n,  -1, pn.Z);//执行透视变换
```

这样我们对透视投影计算的部分就已经修改完成了，接下来我们需要修改一下 GraphicsLibrary.cpp 中的 DrawTriangle 函数。我们在屏幕绘制三角形的时候会将三角形切分成上下两个三角形，而切割点 P 就是我们要处理的第一个点。

```
double p_omega = (pa->W * pc->W) / (pc->W + (pa->W - pc->W) * p_t);//计算 P'点的 ω 值
double p_z = pa->Z + (pc->Z - pa->Z) * p_t;//使用线性插值计算 z 分量
```

可以看到对于 P 点的 Z 值我们使用了线性插值计算，而ω分量仍然使用透视校正插值的相关算法计算，读者千万不要被绕晕了，什么时候使用哪种计算方法心里面一定要清楚。后面代码对 z 的线性插值就和以前的代码一样，记录一下 z 在三角形边的增量，然后递增处理。

```
double dz_in_line = (ez - sz)/(ex-sx);//z 在扫描线上面的增量
for (int x = (int)sx; x <= ex; x++, t += dt, z += dz_in_line)//更新比例值,更新深度值，因为这两个值在执行 continue 时也需要继续更新
```

可以看到我们对 z 值计算改为使用线性插值进行处理，并且使用了增量的计算方法，这样可以减少计算量。

7.3 近平面裁剪的必要性

上面介绍了这么多关于裁剪的知识，裁剪是不是必须要做的呢？我们考虑一下下图 7-5 中的情况：

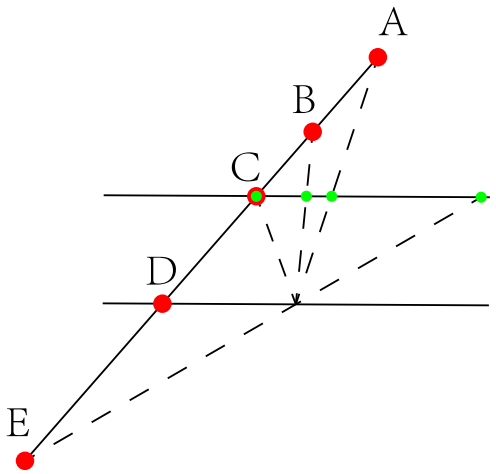


图 7-5 裁剪的必要性

如果在空间有一线段 AE 如上图所示，根据透视投影的规则将这些点和相机原点连接，连线和屏幕的交点为投影点，我们可以看到 A、B、C 几个点都是正确的，但是 D、E 就得不到正确的结果了。首先说明一下 D 点，因为 D 点的深度值为 0，这个点和原点的连线和近平面根本没有交点，所以无法求出投影点，而 E 点落在了相机后方，根据常识，眼睛或者相机是无法看到后面的物体的，E 点也是无法计算的，所以我们必须对线段 AE 进行裁剪，舍弃深度值小于 0 的部分。

这样可以保证所有需要做透视除法的点 z 值都是大于 0 的，也就是我们可以用 z>0 作为裁剪边界，按照常识 n 作为一个相机或者人眼系统，在 z<n 的时候已经进入了相机内部，所以我们可以更加激进一点使用 z≥n 作为裁剪边界条件，而 z=n 则为该直线和边界的交点(出点或者入点)，我们只要计算出 t 就可以很轻松的完成交点的坐标和属性值计算。因为我们用齐次坐标的ω分量保存顶点的原始深度，所以可以用下面这个式子计算出 t。

$$t = \frac{\omega_p - \omega_a}{\omega_b - \omega_a}$$

7.4 裁剪的实现

前面几个小节的知识点我说得非常零碎，关于裁剪具体细节我们并没有好好讨论过，现在我们对裁剪的细节好好的整理一下。

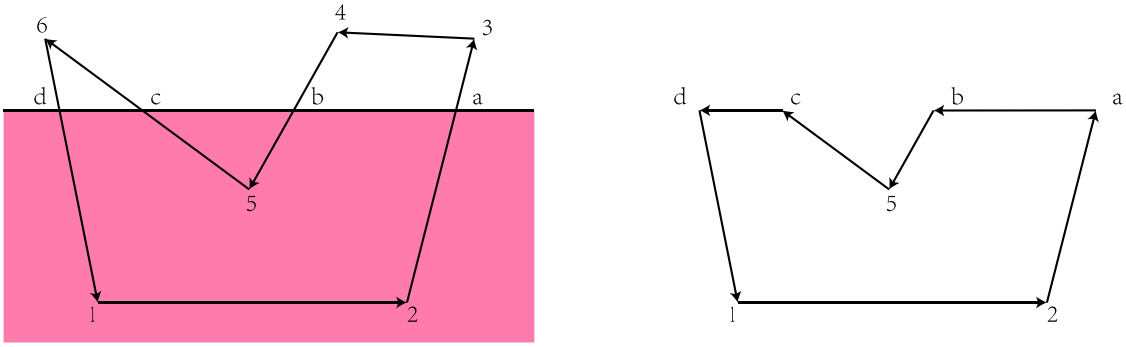
我们先讨论一下线段的裁剪，线段的裁剪经过下面几个步骤：

① 我们先任选一个端点作为线段的起点，另一个则为终点。

② 判断线段是否需要被边界裁剪，如果被平凡接受，则保留原始线段；如果被平凡拒绝，则放弃绘制该线段；如果被裁剪，则进入③。

③ 一条线段被裁剪，必然是一个端点在边界外部，一个端点在边界内部。如果在外部的点是起点，则交点为入点，剩余线段是以入点为新起点，原先的终点为终点的一段线段；反之若线段终点在界外，则交点为出点，剩余线段是以原先的起点为起点，出点为终点的一段线段。

在知道了线段裁剪之后，我们就可以学习一下直线边界对二维多边形的裁剪了：



左图粉色区域为保留的区域，右图为裁剪之后剩余的多边形

图 7-6 多边形使用直线边界裁剪示意图

上图 7-6 中就是一个使用直线边界裁剪多边形的例子，我们可以看到多边形如果和直线有交点，这些交点必定是成对出现的，上图中的 ab 和 cd 为两对交点。每一对交点都是一个出点、一个入点的组合，每一对交点都会形成一条新的边，该边的起点为一对交点中的出点。

① 将多边形分割成依次连接的线段，并且准备一个类似栈的容器，保存裁剪之后多边形的有序顶点。

② 任意选一条线段作为起始线段，依次将这些和边界进行测试和裁剪。

③ 如果该边被平凡拒绝，则进入下一条边的判断(回到②)。

④ 如果该边被平凡接受，将边的起点压栈。

⑤ 如果该边被裁剪，当边与边界交点为入点时，将入点压栈；若交点为出点，则将边的起点压栈，然后将出点压栈。

⑥ 循环③④⑤，直到所有边都和边界进行测试裁剪处理。

上面这个裁剪方法是将所有边的起点记录到栈中，如果读者想改成记录终点，只需要修改④⑤两步即可。

并且如果我们在三维空间中用一个平面作为边界对一个多边形进行裁剪，因为两平面相交为一条直线，我们可以认为是在使用相交线对多边形进行裁剪。所以我们可以使用和二维空间用直线对多边形裁剪一样的操作步骤，并得到一样的结果。

如果是对凸多边形进行裁剪，则剩余多边形的顶点最多只能比原多边形顶点多一个，如下图 7.7 所示：

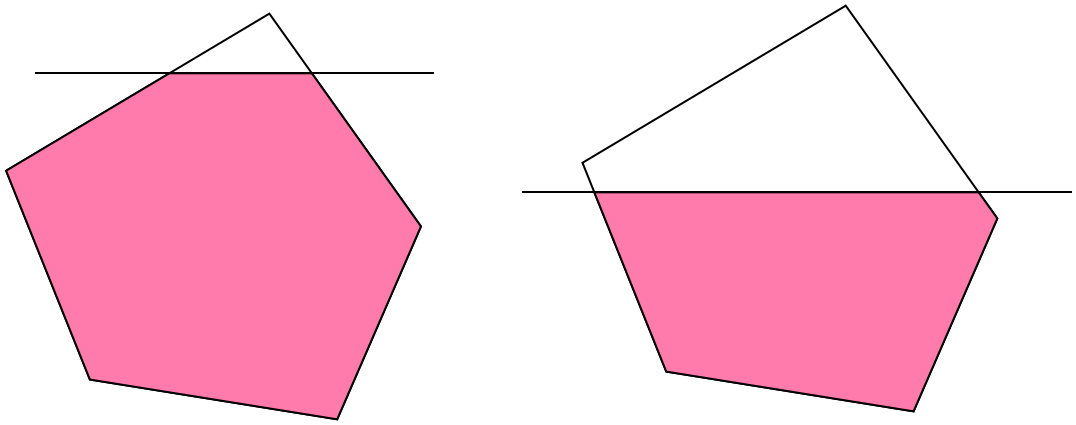


图 7-7 直线对凸多边形的裁剪

如果用直线对凸 n 边形进行裁剪，则裁剪之后的新多边形也是一个凸多边形，并且该

多边形的顶点数量最多为 n+1，上图中左图粉色区域有 6 个顶点，右图只有 5 个顶点。所以我们可以用一个 n+1 的数组作为栈容器。

上面说明的裁剪是在齐次空间中进行的，而我们截止至 7.2 小节所描述的绘制流程如下：

图 7-8 截止至章节 7.2 的绘制流程

所以我们在投影变换之后，透视除法之前对齐次坐标执行裁剪操作即可。我们将 7.2 的代码中新增一个函数 clip，这个函数工作在透视投影变换之后。

本节代码见 Part2\src\chapter7\clip。我们在 GraphicsLibrary 这个类中新增一个函数 ClipAndDraw,这个函数接收的参数和之前的 DrawTriangle 几乎一样，只是多了一个参数 depth_boundary，这个参数用于描述一个平行于平面的裁剪平面深度值，如果 depth_boundary=n，则就是使用近平面裁剪。并且我们把 DrawTriangle 改为 private 函数，ClipAndDraw 将一个齐次坐标表示的三角形裁剪，然后调用 DrawTriangle 来绘制裁剪之后的三角形，所以我们当然不希望直接在 main 函数中再调用了 DrawTriangle 函数了。

我们在 GraphicsLibrary.cpp 中先实现一个裁剪线段的函数 ClipLine，这个函数负责判断线段是否被边界裁剪，如果需要裁剪则计算交点的坐标和属性值，并把计算结果保存至 result 中，因为这个函数的代码太简单，所以我就不多做说明了。

我们 ClipAndDraw 函数中将三角形的各条边依次调用 ClipLine 进行裁剪，并使用前面所说的方法将顶点入栈：

```
75: Point4 const* points[] = { &sa,&sb,&sc };//为了方便循环调用 ClipLine 函数对边进行裁剪，所以将边放入数组
```

我们将个顶点放入数组之后循环调用 ClipLine 函数对边进行裁剪：

```
78: int ret = ClipLine(*points[i],*points[(i+1)%3], tmp, ValueLength, depth_boundary);//points[0],points[1]为边 ab, points[1],points[2]为边 bc，points[2],points[0]为边 ca
```

然后 ret 返回结果，根据返回结果使用前面所说的③④⑤规则进行入栈处理，因为 DrawTriangle 函数只能绘制三角形，而我们可能会把三角形裁剪成四边形，因为凸 n 边形可以分解成 n-2 个三角形，所以我们会对裁剪结果进行判断，如果裁剪结果为一个四边形，我们可以把四边形分解成两个三角形：△abc 和△acd，然后将分解之后的三角形交由 DrawTriangle 函数绘制，如下图 7-9 所示：

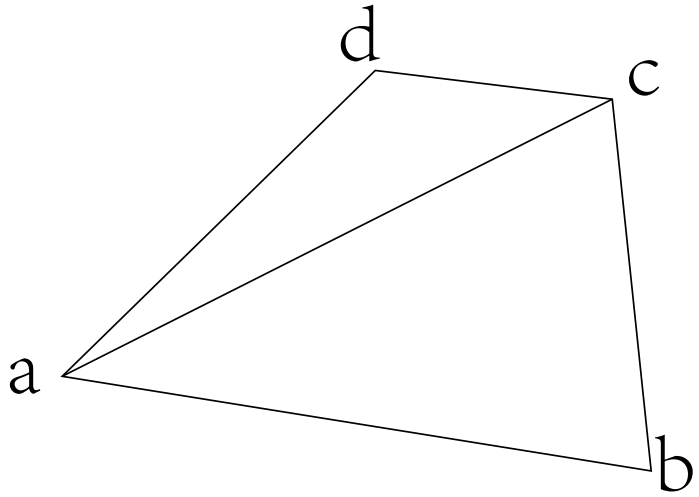


图 7-9 n 边形的划分

```
if (stack_index != 0)//如果三角形的三条边都被平凡拒绝，则不再需要绘制这个三角形了
{
        DrawTriangle(stack[0], stack[1], stack[2], ValueLength, FragmentShader);
if (stack_index == 4)
{
        DrawTriangle(stack[0], stack[2], stack[3], ValueLength, FragmentShader);
}
}
```

我们先绘制裁剪之后顶点 stack[0]、stack[1]、stack[2]围成的三角形，如果裁剪出四个顶点，则我们再绘制 stack[0]、stack[1]、stack[2]围成的三角形。

在 main 函数中，我们调用 gl.ClipAndDraw 函数绘制四个三角形的时候都传递了 depth_boundary=15，程序运行结果如下图 7-10，可以发现我们正确的对多边形进行了裁剪。

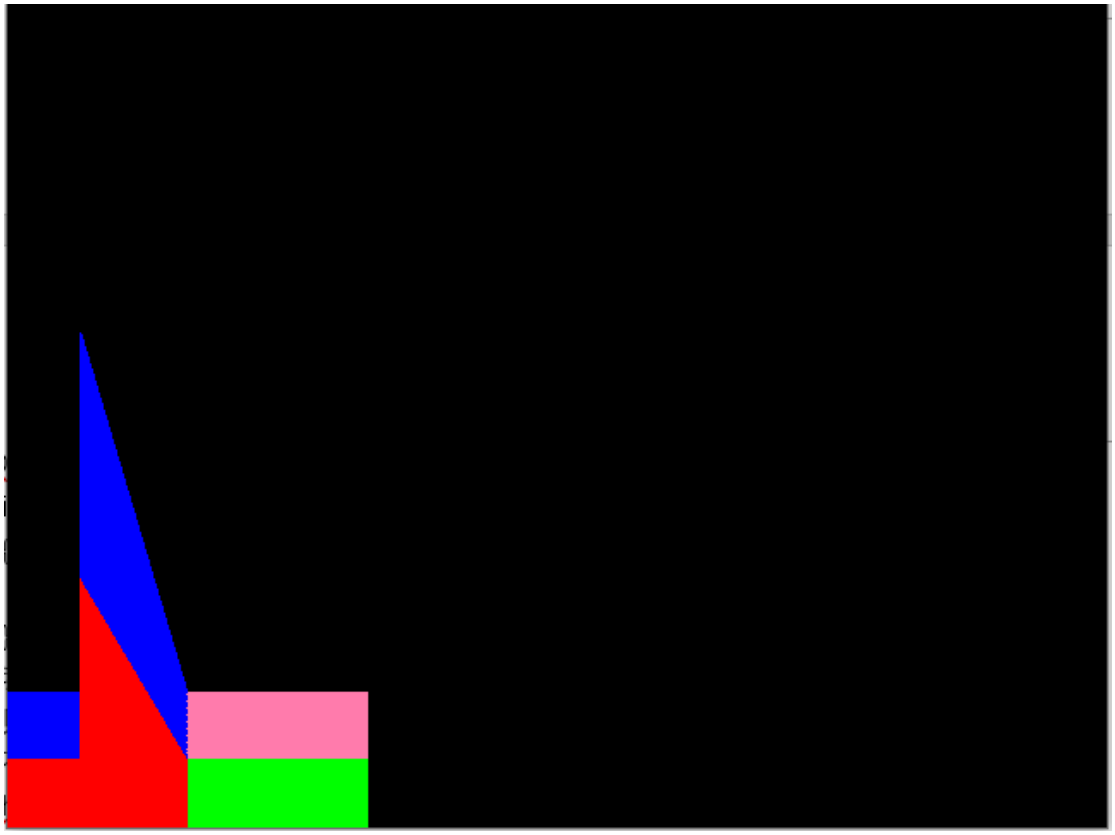


图 7-10 程序运行截图

7.5 附录

7.5.1 附录一

$\frac{z'}{z}=a+\frac{b}{z}$在屏幕空间线性变化的证明

设$\mathrm{f(z)}=\frac{z'}{z}$，其中$z'=kz+h$（为了和顶点 ab 区分，用 k、h 表示一次项和常数项，其中 k 和 h 为常数），则如果有$\frac{f(p)-f(a)}{f(b)-f(a)}=t'$(t’为屏幕上线段 ab 中 b 点的权值),则 f(p)在屏幕空间线性变化。

设$\frac{f(p)-f(a)}{f(b)-f(a)}=t'$，则有$f(p)=f(a)+[f(b)-f(a)]t'$

又因为$\mathrm{f(z)}=\frac{z'}{z}$和 $z'=kz+h$,得到$\mathrm{f(z)}=\frac{z'}{z}=\frac{kz+h}{z}=k+\frac{h}{z}$

则$f(p)=f(a)+[f(b)-f(a)]t'$可以化成:

$$k+\frac{h}{z_p}=k+\frac{h}{z_a}+\left[k+\frac{h}{z_b}-k-\frac{h}{z_a}\right]t'$$

上等式消去 k 可以化成:

$$\frac{h}{z_p}=\frac{h}{z_a}+\left[\frac{h}{z_b}-\frac{h}{z_a}\right]t'$$

等式两边同时除以 h 得到$\frac{1}{z_p}=\frac{1}{z_a}+(\frac{1}{z_b}-\frac{1}{z_a})t'$,又因为ω为原始空间的 z 值，所以还可以写成$\frac{1}{\omega_p}=\frac{1}{\omega_a}+(\frac{1}{\omega_b}-\frac{1}{\omega_a})t'$，该式与已知结论$\frac{1}{\omega_p}=\frac{1}{\omega_a}+(\frac{1}{\omega_b}-\frac{1}{\omega_a})t'$一致，所以$\frac{z'}{z}=a+\frac{b}{z}$在屏幕空间是线性变化的。

第8章 标准视锥体和设备坐标

8.1 设备坐标

本章中屏幕空间、设备坐标等名词会混用，这两个名词表达的是同一个意思，希望读者不要被绕晕。前一章我们介绍了近平面裁剪，并说明了近平面裁剪的必要性：必修舍弃深度值小于 0 的部分，那么是不是只进行深度裁剪就行了呢？我们看看下图 8-1：

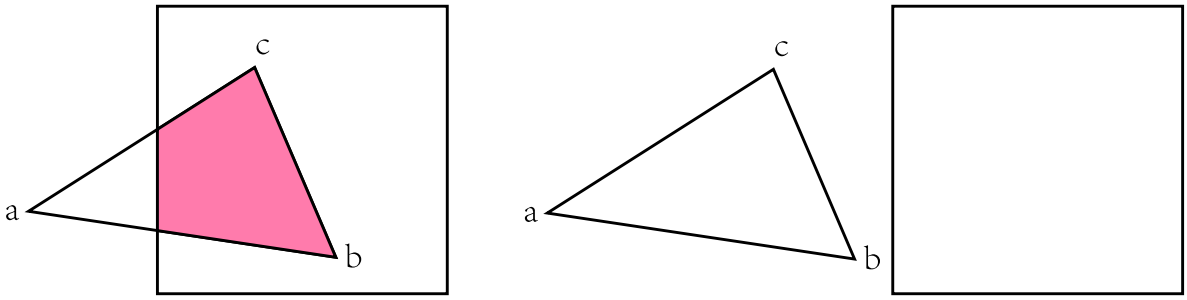


图 8-1 三角形超出屏幕范围

我们到目前为止学习的内容我都是刻意的让三角形经过各种运算之后还在屏幕内部，如果和图 8-1 类似，一个三角形有一部分在屏幕内部，一部分在屏幕外部，或者三角形全部在屏幕外部怎么办？如果我们继续使用之前的三角形绘制方法，就会存在使用函数 SetPixel(x,y)时，x、y 不属于屏幕内部的情况，所以具体会出现什么状态取决于 SetPixel 函数的实现，如果 SetPixel 判断坐标(x,y)不属于屏幕内部的时候放弃绘制的话，那什么也不会发生，否则就会出现 SetPixel 函数出故障的情况(大多数时候都会使程序崩溃，具体情况根据程序运行平台的不同会出现不同的结果)。

而为了更加的规范，所以我们使用业界的常规做法，使用上下左右远近六个平面对空间三角形进行裁剪，如下图 8-2 所示：

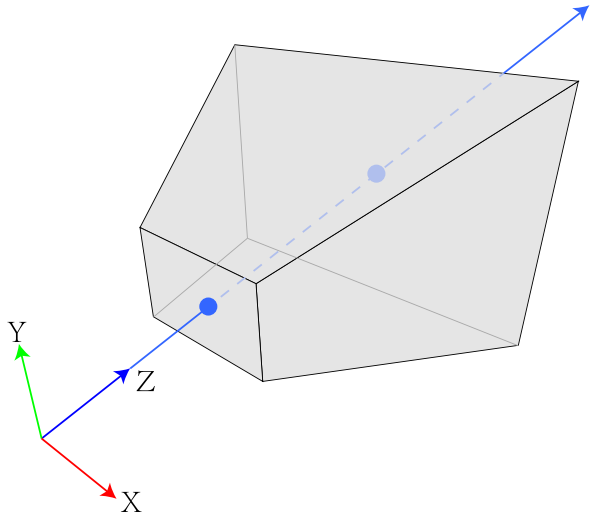


图 8-2 视锥体

距离原点最近并且垂直于 z 轴的这个小矩形就是屏幕，这个矩形位于 near 平面上。而与 near 平面平行的较大的矩形所处平面叫做 far 平面，从原点往屏幕四个角发射四条射线与 far 平面相交，四个交点围成的矩形即为上图中离原点较远并且垂直于 z 轴的矩形。同时

这 near 和 far 平面与四条射线中间还可以得到四个梯形，这四个梯形即为 left、right、top、bottom 平面，这个六面体叫做视锥体，视锥体是一个抽象的概念，现在我们就研究一些视锥体的特性。为了便于画图理解，我们从 Y 轴正方向往原点观察视锥体，可以看到一个类似下图 8-3 的形状：

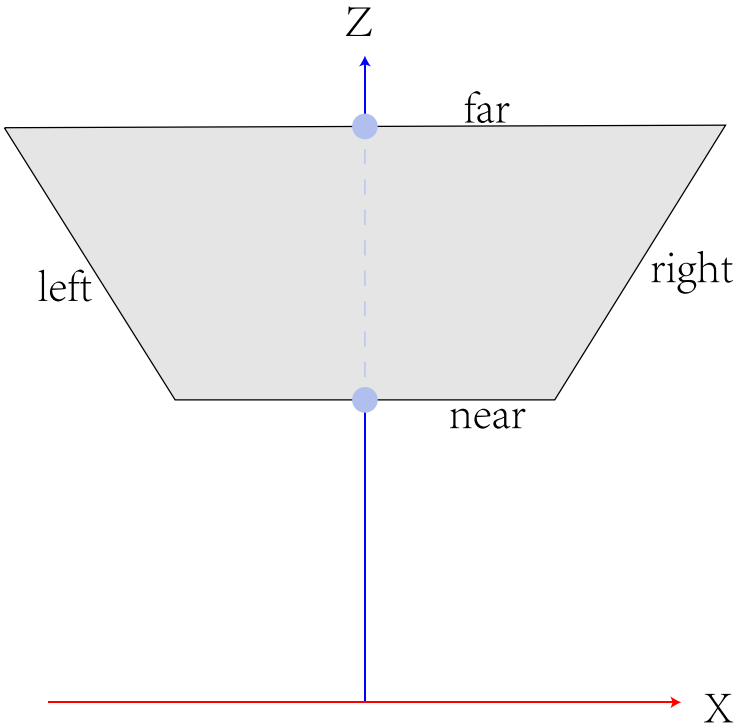


图 8-3 视锥体俯视图

如果我们对原始的顶点执行透视投影和透视除法之后，有一个很有趣的现象，如下图 8-4 所示：

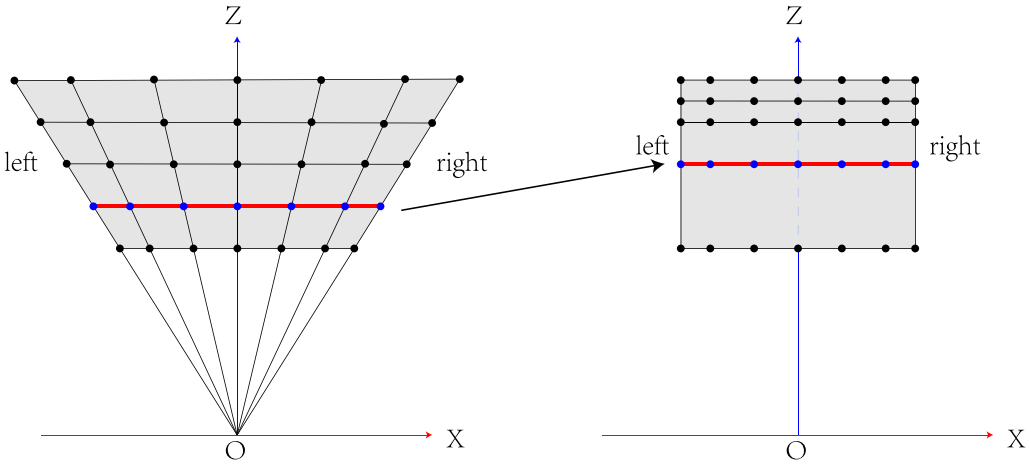


图 8-4 原始空间到屏幕空间的变化

我们可以看到，原始空间的坐标经过透视变换和透视除法之后，原来的视锥体变成了一个长方体(因为上图是俯视图，所以看起来就变成了一个矩形)，原始深度分量映射到屏幕空间长方体则使用倒数函数来映射，x、y 分量则是根据深度得到一个平面。图 8-4 中有五行共 35 个顶点，每一行顶点的深度值都是相同的。为了方便讨论，我们看一下从下往上数第二行蓝色顶点，确定一个平面和 z 轴垂直，然后让这个平面和 left、right 边界相交，再把交界线性的变换到屏幕的 left、right 边界，屏幕空间的这个长方体所使用的坐标就叫做设备坐标。

在上一章我们把屏幕空间坐标的计算公式改为：

$$x' = \frac{xn}{z}$$

$$y' = \frac{yn}{z}$$

$$z' = a + \frac{b}{z}(b < 0)$$

$$\omega' = z$$

上面式子中ω分量在计算透视校正插值是有用，但是对设备坐标来说没有意义，我们先不予以考虑。很明显相同深度下(z 相等时)x、y 分量线性变换到设备坐标的 X、Y 分量，不同深度值使用倒数函数变化到设备坐标的深度，通过画图和公示解释希望读者能够理解得更加深刻(我的表述能力有点差，我自己也不是很满意上面的解释，要是读者看不懂，可以多消化理解一下)。

8.2 设备坐标的标准化

通过上节介绍的知识，我们知道设备坐标在原始空间中是一个锥体，而在屏幕空间中则变成了一个立方体。我们可以想象的出来，在前面章节的投影中，我们都是以屏幕左下角作为原点，屏幕的一个像素为一个单位长度，即 z 轴垂直的穿过了屏幕的左下角，现在我们重新定义一下原点的位置和单位长度，X、Y、Z 轴的方向保持不变，如下图 8-5 所示：

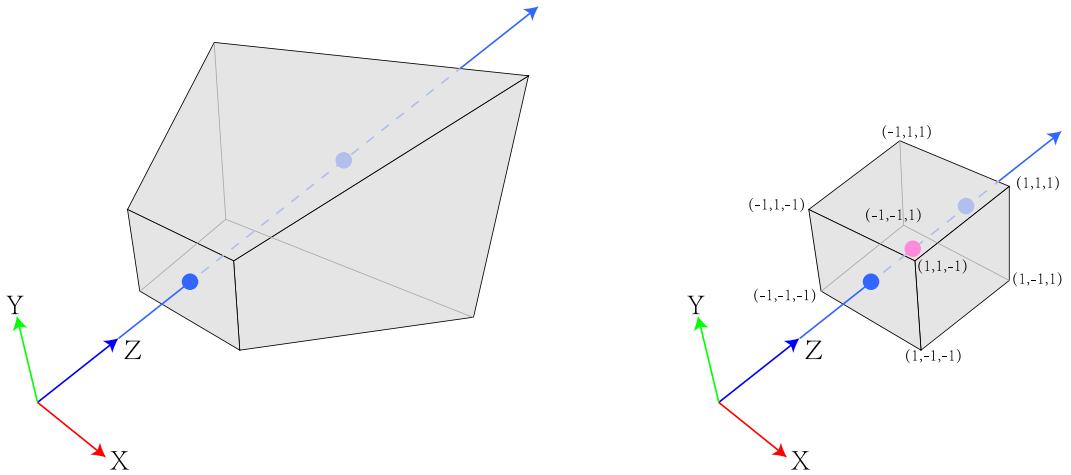


图 8-5 左：视锥体 右：视锥体变换到标准化设备坐标

在设备坐标中，把屏幕的宽和高平分成两个单位长度，并且令屏幕正中心坐标的 x、y 分量为(0,0)，则设备坐标的 x、y 的取值范围为[-1,1]，然后在把 near 平面到 far 平面的深度值也分为两位单位，深度值范围为[-1,1]，通过这种划分，设备坐标的坐标范围就变成了(-1,-1,-1)到(1,1,1)，需要注意的是，我们这些坐标都是没有具体长度单位的，这些数值都只是一个比例，如果屏幕宽高比不是 1:1，则在屏幕纵向的一个单位和屏幕横向的一个单位是表示不同像素距离的。(OpenGL 将设备坐标深度值划分到[-1,1]，DirectX3D 则将深度范围划分到[0,1]，这些划分都没有本质区别，我们目前选择了和 OpenGL 一样的深度区间划分)。

我们把设备坐标坐标范围限定在[-1,1]，经过标准化的设备坐标叫做 NDC(Normalize Device Coordinates)，通过图 8-5 可以知道 NDC 在原始空间中是一个锥体，这个锥体叫做标准视锥体 CVV(Canonical View Volume)。

我们需要对前面章节的代码和透视投影公式进行修改，本节代码见\Part2\src\chapter8\cvv。

首先我们得修改一下透视投影的计算公式，我们之前计算投影的时候用的是下面这个公式：

$$x' = xn$$
$$y' = yn$$
$$z' = az + b(b < 0)$$
$$\omega' = z$$

之前这个公式只用了 n 一个参数，现在我们要把 near、far、left、right、top、bottom，后面把这些边界简称为 n、f、l、r、t、b 这个 CVV 的六个边界都用上了。

首先我们先计算x'的公式，因为当 z 值固定时$x_{ndc} = \frac{xn}{z}$在 NDC 中是线性变化的，并且当$x = l$时，$\frac{xn}{z} = -1$，当$x = r$时，$\frac{xn}{z} = 1$，实际上就是把$\frac{xn}{z}$从$[l, r]$线性插值到[-1,1]区间，那么可以得到如下等式：$\frac{\frac{nx}{z}-l}{r-l} = \frac{x_{cvv}-(-1)}{1-(-1)}$，我们可以解出：$x_{cvv} = \frac{\frac{2nx}{z}}{r-l} - \frac{r+l}{r-l}$。

同样的把$\frac{yn}{z}$从$[b, t]$插值到[-1,1]，通过等式：$\frac{\frac{ny}{z}-b}{t-b} = \frac{y_{cvv}-(-1)}{1-(-1)}$得到$y_{ndc} = \frac{\frac{2ny}{z}}{t-b} - \frac{t+b}{t-b}$。

而$z_{ndc} = \frac{az+b}{z} = a + \frac{b}{z}$的计算和 x、y 的计算略有不同，因为$z_{ndc}$在 NDC 中不是线性变化的，但是不用担心，同样可以通过很简单的推导把公式计算出来，因为我们只需要求出 a 和 b 的值就行了。

当 z=n 时，$z_{cvv} = a + \frac{b}{z} = -1$,当 z=f 时，$z_{cvv} = a + \frac{b}{z} = -1$，可以得到下面两个等式：

$$\begin{cases} a + \dfrac{b}{n} = -1 \\ a + \dfrac{b}{f} = 1 \end{cases}$$

联立上述等式可以解出$a = \frac{f+n}{f-n}$，$b = \frac{2nf}{n-f}$。于是我们得到的投影公式为：

$$x_{ndc} = \frac{\frac{2nx}{z}}{r-l} - \frac{r+l}{r-l}$$

$$y_{ndc} = \frac{\frac{2ny}{z}}{t-b} - \frac{t+b}{t-b}$$

$$z_{ndc} = \frac{f+n}{f-n} + \frac{\frac{2nf}{n-f}}{z}$$

上述公式是最终计算出来的在屏幕空间的坐标，如果我们用齐次坐标表示，并令ω分量等于 z，可以得到如下公式：

$$x' = \frac{2nx}{r-l} - \frac{r+l}{r-l}z$$

$$y' = \frac{2ny}{t-b} - \frac{t+b}{t-b}z$$

$$z' = \frac{f+n}{f-n}z + \frac{2nf}{n-f}$$

$$\omega' = z$$

当然这和其他教程中 OpenGL 中推导出来的透视投影公式略有不同，因为其他教材使用的都是右手坐标系，即屏幕向外为 Z 轴正方向的坐标系，而我们的公式是以屏幕向里为 Z 轴正方向的坐标系。

我们就简单的修改一下 main.cpp 中的 perspective 函数，将透视投影公式修改为上面所说的公式，这里有一个小技巧就是令$\frac{r-l}{t-b} = \frac{屏幕宽}{屏幕高}$可以使绘制出来的图形不变形。

接下来我们还得修改一下 DrawTriangle 函数，因为这个函数之前接受的顶点都是用屏幕像素作单位，而现在用的是[-1,1]区间的点，所以我们需要把[-1,1]区间坐标的 X、Y 分量变换成屏幕像素坐标，这里只需要用一个简单的线性变换即可完成。

```
//将[-1,1]变换成[0,width-1]和[0,height-1]
a.X = (a.X + 1) / 2 * (graphicsdevice.width - 1);//因为屏幕宽度坐标为 0～width-1,高度为 0～height-1,所以这些运算都有-1);
a.Y = (a.Y + 1) / 2 * (graphicsdevice.height - 1);
b.X = (b.X + 1) / 2 * (graphicsdevice.width - 1);
b.Y = (b.Y + 1) / 2 * (graphicsdevice.height - 1);
c.X = (c.X + 1) / 2 * (graphicsdevice.width - 1);
c.Y = (c.Y + 1) / 2 * (graphicsdevice.height - 1);
```

这次在 main 函数中我们只测试一个三角形，可以得到如下的结果：

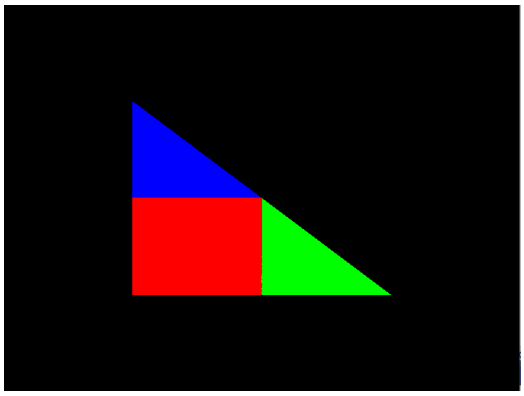


图 8-6 程序运行结果

8.3 将多边形裁剪到视锥体内部

前面我们说了裁剪的时候至少应该用 left、right、top、bottom、near 五个平面对三角形裁剪，用 l、r、t、b 裁剪是为了保证多边形始终在屏幕内部，用 near 裁剪是为了保证深度值不小于 0，而包括 OpenGL 和 DirectX3D 在内的很多图形库都会额外再使用 far 平面进行裁剪，让距离较远的部分不被渲染。

在前一章中我们知道裁剪是在齐次空间进行的，因为我们特意的保证了三维空间的直线经过透视变换到齐次空间之后仍然为直线，所以我们用这六个边界进行裁剪的时候也是在齐次空间中处理。

因为直线在齐次空间中仍然为一条直线，所以我们任意拿出两个分量，这两个分量在二维直角坐标系下也仍然是直线，类似于三视图，去掉一个维度变成二维图形之后直线仍然是直线。我们先讨论一下 left 边界，我们取出 x、ω分量得到下图 8-7：

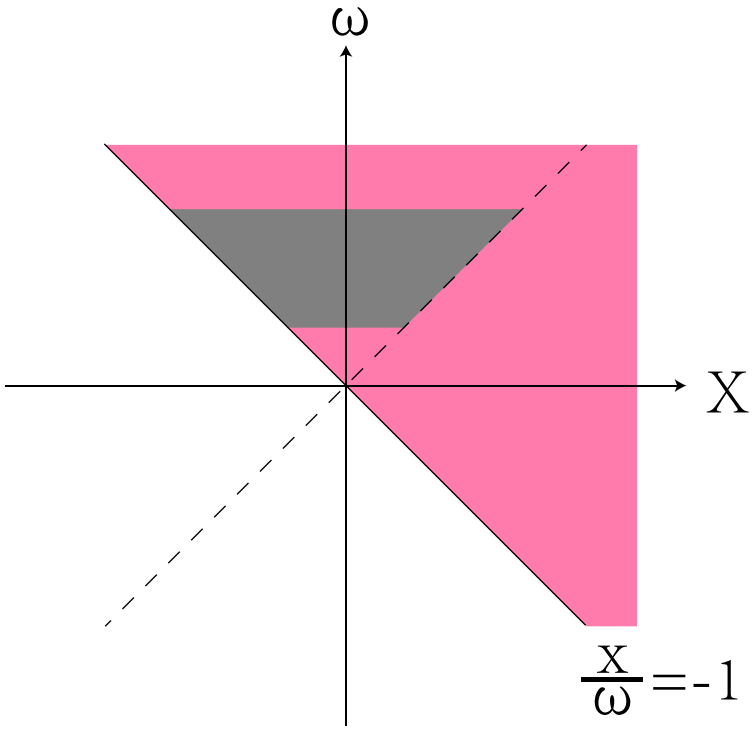


图 8-7 left 边界齐次坐标保留 x、ω分量示意图

上图中灰色区域为 CVV 范围，left 边界为$x = -\omega$的一条直线，保留区域为上图中的粉色区域，通过上图我们可以知道裁剪条件为$x + \omega \geq 0$，我们设线段 ab 和边界 left 的交点为 P，可以得到$\frac{x_p}{\omega_p} = -1$，因为裁剪需要计算出 b 点的权值 t，根据$\frac{x_p}{\omega_p} = -1$可以得到等式$\frac{x_a+(x_b-x_a)t}{\omega_a+(\omega_b-\omega_a)t} = -1$,于是解出$t = \frac{x_a+\omega_a}{x_a+\omega_a-x_b-\omega_b}$。

同样的，对于 right 边界得到如下图 8-8 示意图：

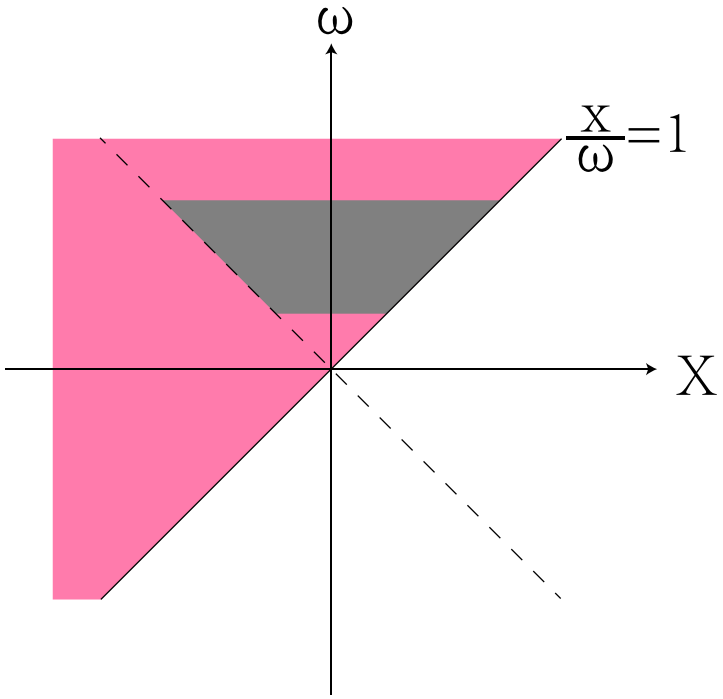


图 8-8 right 边界示意图

通过上图可以得到 right 边界为$x-\omega\le 0$, 线段 ab 和 right 边界交点 P 的关系为$\frac{x_p}{\omega_p}=1$, 可以得到$\frac{x_a+(x_b-x_a)t}{\omega_a+(\omega_b-\omega_a)t}=1$，解出$t=\frac{\omega_a-x_a}{\omega_a-x_a-\omega_b+x_b}$。推导 top、bottom、near、far 边界也是类似操作，保留ω分量和 y、z 分量即可通过示意图推导出来。我们将边界整理一下，得到下表 8-1：

表 8-1 各边界和不等式

边界	不等式 1	不等式 2	t 值公式 1	t 值公式 2
left	$x+\omega\ge 0$	$\omega+x\ge 0$	$\frac{x_a+\omega_a}{x_a+\omega_a-x_b-\omega_b}$	$\frac{\omega_a+x_a}{(\omega_a+x_a)-(\omega_b+x_b)}$
right	$x-\omega\le 0$	$\omega-x\ge 0$	$\frac{\omega_a-x_a}{\omega_a-x_a-\omega_b+x_b}$	$\frac{\omega_a-x_a}{(\omega_a-x_a)-(\omega_b-x_b)}$
bottom	$y+\omega\ge 0$	$\omega+y\ge 0$	$\frac{\omega_a+y_a}{y_a+\omega_a-y_b-\omega_b}$	$\frac{\omega_a+y_a}{(y_a+\omega_a)-(y_b+\omega_b)}$
top	$y-\omega\le 0$	$\omega-y\ge 0$	$\frac{\omega_a-y_a}{\omega_a-y_a-\omega_b+y_b}$	$\frac{\omega_a-y_a}{(\omega_a-y_a)-(\omega_b-y_b)}$
near	$z+\omega\ge 0$	$\omega+z\ge 0$	$\frac{\omega_a+z_a}{z_a+\omega_a-z_b-\omega_b}$	$\frac{\omega_a+z_a}{(z_a+\omega_a)-(z_b+\omega_b)}$
far	$z-\omega\le 0$	$\omega-z\ge 0$	$\frac{\omega_a-z_a}{\omega_a-z_a-\omega_b+z_b}$	$\frac{\omega_a-z_a}{(\omega_a-z_a)-(\omega_b-z_b)}$

上表中不等式 2 是不等式 1 变换而来，同样的 t 值公式 2 也是 t 值公式 1 变换得到的，之所以这样变换，是为了让编程更加方便，因为在 t 值公式 2 的公式里包含了不等式 2 的判断条件，可以少写几行代码，能少写几行算几行，写多了头晕，具体实现请参考本节代码说明部分。

需要注意的是我们不能使用$\frac{x'}{\omega}>-1$作为 left 边界的判别条件，因为$\frac{x'}{\omega}$得到的是经过透视

除法之后的 NDC 坐标，原始空间的直线到屏幕空间已经不再是一条直线了，并且当$\omega \leq 0$时透视除法会得到完全错误的坐标点。

上一节的代码还保留着前一章 near 平面的裁剪，所以我们只需要在裁剪这个地方进行修改，将其改为用六个边界轮流裁剪，这样就完成了 CVV 的完整裁剪。代码见\Part2\src\chapter8\cilp_in_cvv。

我们将 Graphics.cpp 中 ClipLine 函数修改一下，其参数 clip_flag 表示用哪个边界进行裁剪，在这个函数中我们用不等式 2 和 t 值公式 2 减少了代码量。

在 ClipAndDraw 函数中，我们依次使用六个边界对三角形进行裁剪，因为对多边形每次裁剪会得到一组新顶点，所以我们准备了六个栈用来接收六次裁剪的结果，并且前一次的裁剪结果作为后一次裁剪的输入，加上第一次裁剪的输入，总共需要七个容器，因为凸 n 边形被直线边界裁剪之后剩余部分仍然为凸多边形，并且边数最多为 n+1，所以三角形第一次被裁剪可能为凸 4 边形，第二次裁剪可能为凸 5 边形，裁剪六次之后最多只能为凸 9 边形。所以我们的容器容量依次为 3，4，5，6，7，5，9，最后将裁剪完毕的多边形拆分成多个三角形，然后依次使用 DrawTriangle 进行绘制，代码改动的部分就结束了。为了测试裁剪效果，我特意将三角形右下角的顶点深度修改成比 near 更小一点的值，程序运行结果如下图 8-9 所示，三角形右下角被裁剪了一部分。

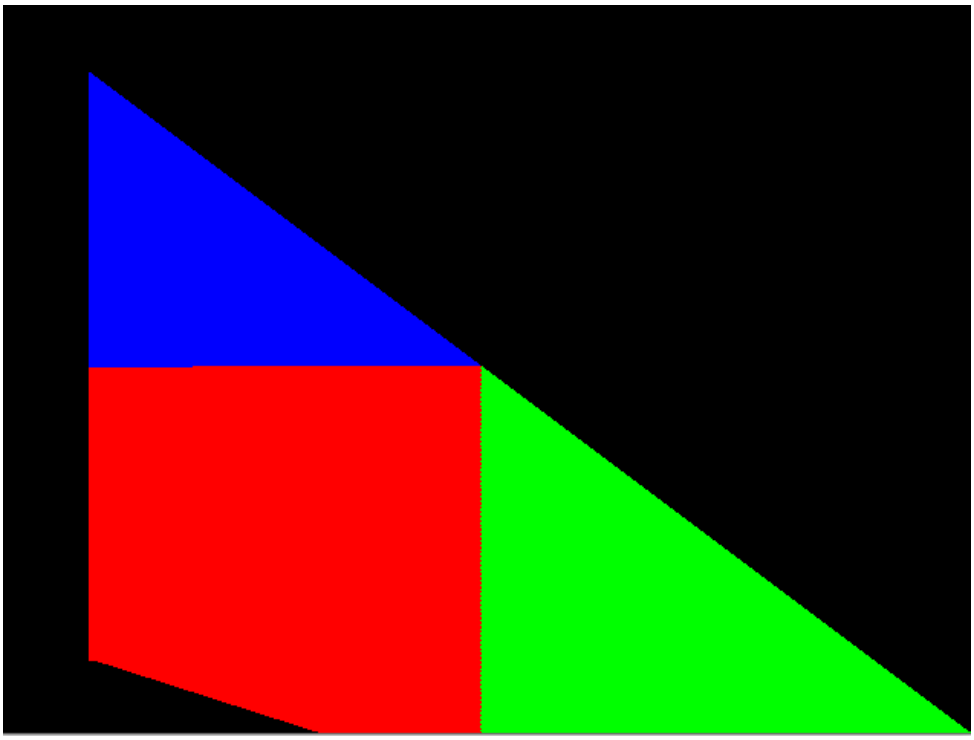


图 8-9 程序运行结果

8.4 一个核心功能完备的渲染器出炉了

通过前面这么多知识的学习，一个核心功能完备的渲染器已经被我们完成了，我们可以得到如下图 8-10 的渲染过程:

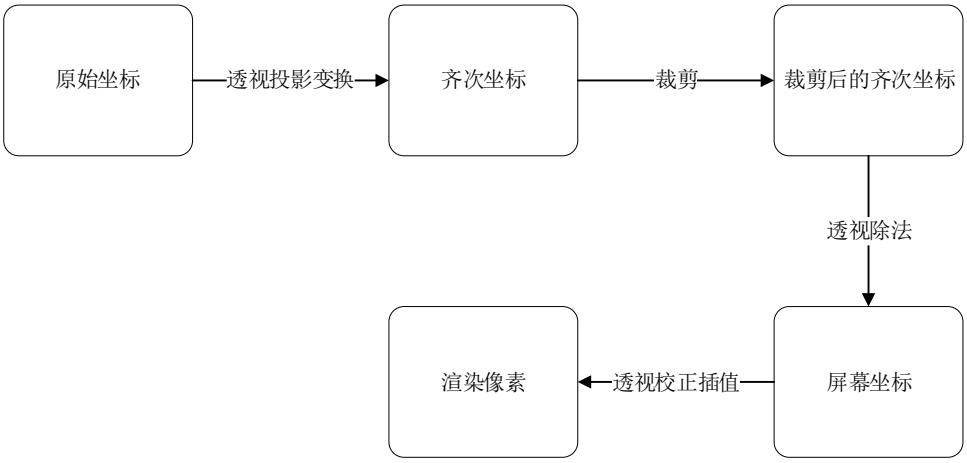


图 8-10 三维渲染过程

而上面这个渲染流程并不是三维渲染的唯一渠道，裁剪的时候我们可以在三维空间内先用视锥体将多边形裁剪完毕，然后再将裁剪之后的顶点直接投影到屏幕上。同样的，透视校正插值也不是唯一的插值方法，我们可以通过屏幕上的像素坐标反推出该像素在原始三角形上面的位置，然后用该位置进行线性插值。以上很多问题都没有唯一解，更多的时候我们是在选用一个实现起来更加简单的方法而已，希望读者的不要被我前面章节所说的内容束缚住。

第三部分：渲染器的功能扩展

我们在第八章的时候已经知道渲染物体需要经过哪些步骤，并且经过这些步骤我们正确的把模型渲染出来了，我们将对前面章节所做出的渲染器进行功能扩展，使其使用起来更加的方便。本部分将会模仿 OpenGL 的渲染管线来扩充我们渲染器的功能，实现 VBO、顺时针逆时针三角形的剔除、深度复位、像素复位、双缓冲、用矩阵完成坐标变换、纹理读取、漫反射光照数学模型等功能，最终让画面动起来。

第9章 渲染器功能升级

9.1 OpenGL 渲染管线简介

管线一词来自于 pipeline，就是流水线的意思，渲染管线则翻译自 Render pipeline，指渲染的过程。我们在前面章节中绘制模型时都是在 main 函数中定义一个三角形的坐标和顶点属性之后投影，然后传递至 GraphicsLibrary 库渲染，渲染完一个三角形之后再继续用同样的方法处理下一个三角形，如果一次要绘制很多个三角形，那么代码看起来将会非常的凌乱，并且也不容易管理这些顶点，所以引出了顶点缓冲对象和这个概念，顶点缓冲对象通常也被叫做 VBO(Vertex Buffer Object)。

OpenGL 的渲染流程大致上如下图 9-1 所示：

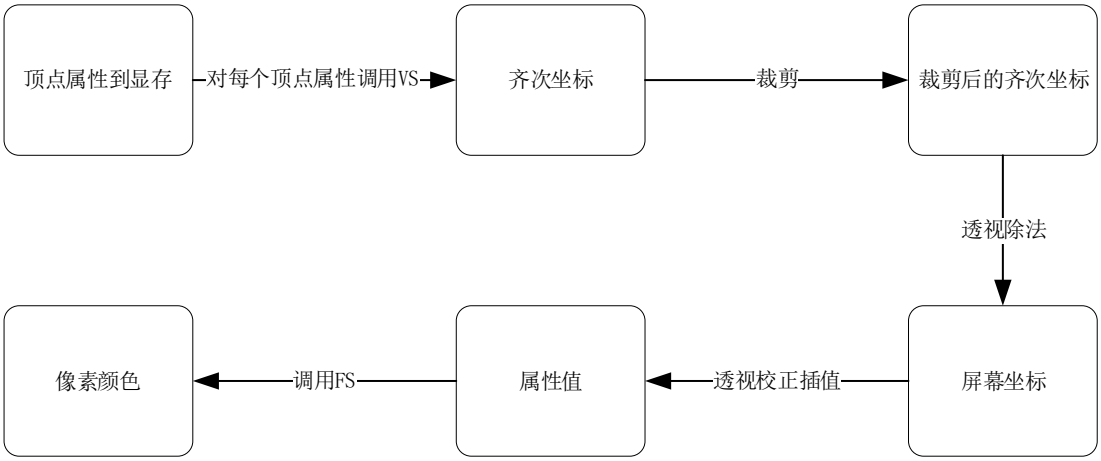


图 9-1 OpenGL 渲染流程

在最初的时候需要把顶点数据复制到显存中，这些数据包含了顶点的一些属性。然后 OpenGL 根据规则依次从 VBO 中取出一个三角形的三个顶点，并对每一个顶点调用一次顶点着色器程序，在顶点着色器中使用顶点属性计算坐标，并执行旋转、缩放、平移、投影等仿射变换，然后在齐次空间中对变换之后的三角形进行裁剪。在裁剪完毕之后对每个多边形执行透视除法，然后依次使用(重心坐标插值)线性插值和透视校正插值对顶点属性进行插值，并将插值的结果传递给片元着色器程序，由片元着色器最终输出像素的颜色。其中用户需要进行的操作有复制顶点数据至显存、编写 VS(vertex shader)代码、编写 FS(fragment shader)代码。

9.2 改造我们的渲染器

我们将前面章节的代码进行修改，改为使用 VBO 进行渲染，这样就可以大幅度的减少 main 函数代码的复杂度，本章代码在\part3\src\chapter9\section2 中。

首先我们为 GraphicsLibrary 类增加 VBO 设置的相关代码，在 OpenGL 中可以创建多个 VBO，而这里为了实现更加方便，我们只允许使用 vbo1 和 vbo2。描述一个缓冲区需要知道该缓冲区包含了多少个顶点的信息，每个顶点又是怎么样使用这些数据的，在 GraphicsLibrary.h 中对 vbo 的定义如下：

```
45: //定义两个 VBO
46: double* vbo1 = nullptr;
47: double* vbo2 = nullptr;
48: size_t vbo1_size = 0;//vbo1 中每个顶点拥有的属性数量
49: size_t vbo2_size = 0;//vbo2 中每个顶点拥有的属性数量
50: size_t position_length = 0;//总顶点数量
```

我们使用 setVBO 函数来设置 vbo 的相关参数，在设置好 VBO 参数后 draw 函数将会作为 graphicslibrary 渲染管线的入口。我们先对每一个顶点调用一次顶点着色器程序，这个程序将该顶点的各个原始属性进行运算，得到该顶点的齐次坐标点和齐次坐标点对应的属性。在 GraphicsLibrary.cpp 的 51 行，我们取出 vbo1 和 vbo2 中该顶点对应的原始属性传递给 VertexShader 函数。

```
51: VertexShader(vbo1 + i * vbo1_size, vbo2 + i * vbo2_size, vertexs[i % 3], varyings[i % 3]);
```

该函数接收顶点原始数据后需要做两件事，1、使用 vbo 计算出该顶点投影之后的齐次坐标，并将该坐标赋给 gl_Vertex 变量，该顶点将作为渲染器渲染三角形时使用的顶点；2、使用 vbo 计算该顶点的属性值，并将该属性值赋给 varying 变量。在 OpenGL 中，属性从齐次空间中的顶点传递给屏幕顶点时，这个变量就需要声明为 varying 类型，也就是在屏幕空间使用透视校正插值计算出来的变量就是 varying 变量。

VS 的实现在 main.cpp 中，如下图 9-2 所示：

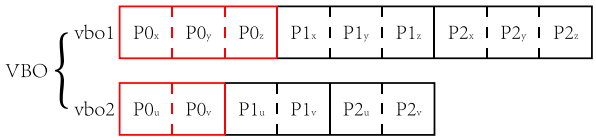


图 9-2 vbo 示意图

我们将 vbo1 设置为每个顶点 3 个属性，vbo2 每个顶点 2 个属性，渲染器每次调用顶点着色器的时候都会传递该顶点的属性进来(对应图 9-2 红色部分)，我们对 vbo 进行运算，然后设置 gl_Vertex 和 varying。

```
41: Point4 p(vbo1[0], vbo1[1], vbo1[2], 1);
42: gl_Vertex = Persective(p, -320, 320, -240, 240, 5, 20);
43: varying[0] = vbo2[0];
44: varying[1] = vbo2[1];
```

顶点着色器代码在 main.cpp 的 VS 函数中，我们使用 vbo1 的三个值计算透视投影的结果，并赋给渲染器的内置变量 gl_Vertex，然后将 vbo2 的两个值直接赋给 varying。

接下来的改造就更为简单了，还是类似的裁剪流程，之前我们是把顶点属性附带到顶点中，所以裁剪的时候只要准备好接收顶点的容器就行了，现在我们把顶点和属性分离出来，所以裁剪的时候还得准备接收 varying 的容器，所以在 ClipTriangle 函数中多了几行代码。在裁剪之后使用 DrawTriangle 函数绘制三角形，因为我们把顶点和属性值分开了，所以原先用于计算顶点属性值的代码都相应改成计算 varying 的代码。在使用透视校正插值计算每个像素的 vraying 之后，调用片元着色器即可，片元着色器在 main.cpp 的 FS 中。

可能是之前我写代码的时候无意识的向 OpenGL 渲染管线靠拢，所以我们的代码改动量并不大。因为我们目前只是把渲染的流程梳理了一遍，所以程序运行效果和第八章一模一样，我就不贴截图了。

9.3 三角形剔除

我们知道一个 3D 模型通常都是由多个三角形拼接而成的，不用三角形表示的模型也能把多边形拆分成多个三角形。通常我们都会刻意的让三角形成为顺时针三角形或者逆时针三角形，如下图 9-3 所示：

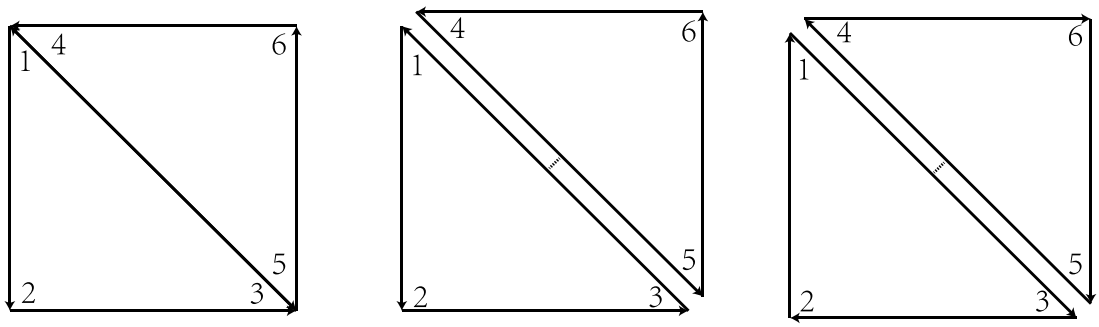


图 9-3 左：正方形 中:逆时针三角形 右:顺时针三角形

一个矩形方形可以由两个三角形拼接而成，上图表示的就是一个正方形，为了易于查看，所以我将两个分开一小点距离，形成图 9-3 中和右的样子。我们在使用 3ds Max 等建模软件生成模型的时候，就可以设置生成的模型三角形的方向。通过三角形方向，我们可以区分一个三角形的正反面，如下图 9-4 所示,我们定义了一个逆时针三角形，则该三角形的正面是蓝色箭头所指的方向。

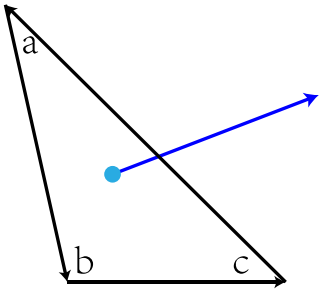


图 9-4 三角形方向

可以发现，对于一个逆时针三角形来说，如果该三角形经过旋转，最后落在屏幕上变成了顺时针三角形，则这个三角形变成了背面朝向屏幕，从相机坐标系原点往 Z 轴正方向观察，只能看到三角形的背面，而且上述情况对于顺时针定义的三角形同样适用，当顺时针三角形背面朝向屏幕时，他在屏幕上就变成了逆时针三角形，如下图 9-5 所示：

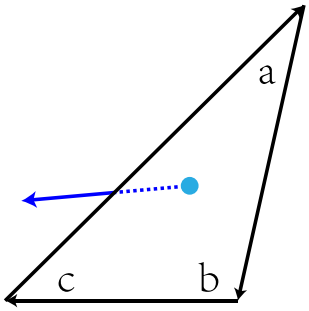


图 9-5 三角形背面朝向屏幕

如果一个封闭的三维模型所有的外表面都定义成同一种方向的三角形，当三角形背面朝向屏幕时，说明我们观察到的是该模型的内表面，而在相机没有穿过物体边界的正常情况下，任何模型的内表面都是会被遮挡住的，如下图 9-6 所示：

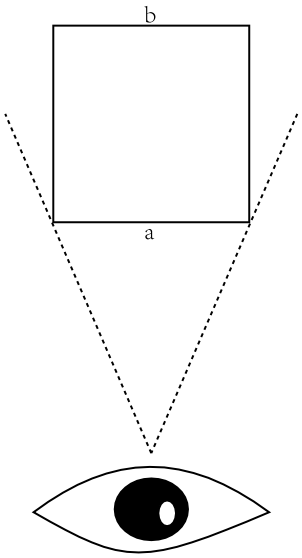


图 9-6 遮挡示意图

从上图中我们可以看到，b 平面此时是背面朝向屏幕的，此时应该被 a 平面所遮挡。所以我们可以使用顺时针三角形剔除或者逆时针三角形剔除来放弃背面朝向屏幕的三角形，这样可以大幅度的减少不必要的运算，至于到底使用顺时针还是逆时针取决于模型外表面三角形的定义。

还记得我们在第二章说过的有向面积吗？我们可以利用有向面积作为三角形方向的判断条件。

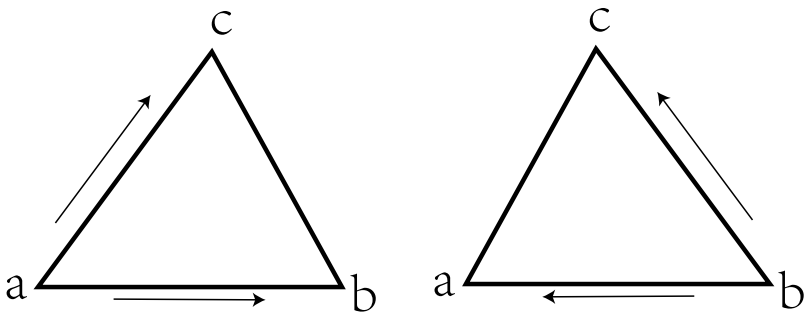


图 9-7 顺逆时针三角形的有向面积

如上图 9-7 所示，如果我们构造一个以屏幕向外为 z 轴正方向，水平向右为 x 轴正方向，垂直向上为 y 轴正方向的右手坐标系，那么使用$\frac{\overrightarrow{ab}\times\overrightarrow{ac}}{2}$得到的面积叫做有向面积，很明显当屏幕上△abc 为逆时针三角形时，向量叉积大于 0，当△abc 为顺时针三角形的时候叉积小于 0，因为我们同样的只是判断正负数和是否等于 0，所以下面的分母 2 在代码中同样可以省略。

本节代码见\part3\src\chapter9\ccw，我们在 GraphicsLibrary.h 中新增了三个宏，用于描述裁剪规则：

```
#define _FrontFace_N 0 //normal 不剔除
#define _FrontFace_CW 1 //cw clockwise 保留顺时针
#define _FrontFace_CCW 2 //ccw counterclockwise 保留逆时针
```

然后在 GraphicsLibrary 这个类中新增一个变量 FrontFace 用于记录三角形剔除规则，之前的代码在屏幕上绘制三角形前会进行面积是否为 0 的判断。而这次我们仅仅增加一小点代码，除了判断面积为 0 之外，还根据 FrontFace 记录的值进行有向面积的符号判断：

```
double square = a.X * b.Y + b.X * c.Y + c.X * a.Y - a.X * c.Y - b.X * a.Y - c.X * b.Y;//得到有向面积的 2 倍
if (abs(square)<1e-15)//当面积为 0 时放弃本三角形的绘制
{
    return;
}
if (FrontFace == _FrontFace_CW && square > 0)//当需要保留顺时针三角形但是有向面积为正数时放弃
{
    return;
}
else if (FrontFace == _FrontFace_CCW && square < 0)//当需要保留逆时针三角形但是有向面积为负数时放弃
{
    return;
}
```

读者在 main 函数中对 gl.FrontFace 进行赋值即可看到运行效果，所以我也就不截图了。

9.4 屏幕刷新

前面学习了这么久的知识，我们都只是渲染一帧静态的画面，如果我们要渲染动画，那么我们需要渲染不同时刻的画面，然后让画面在屏幕上面高速替换就行了。因为这涉及到人眼视觉暂留的一些效应，我也很难解释清楚，原则上来说就是画面在单位时间内替换的速度越快，即帧数越高，人眼看起来就越流畅。

9.4.1 上一帧残留数据清理

如果我们在绘制第一帧画面之后再接着绘制第二帧的画面，就会出现画面叠加的情况，如下图 9-8 所示：

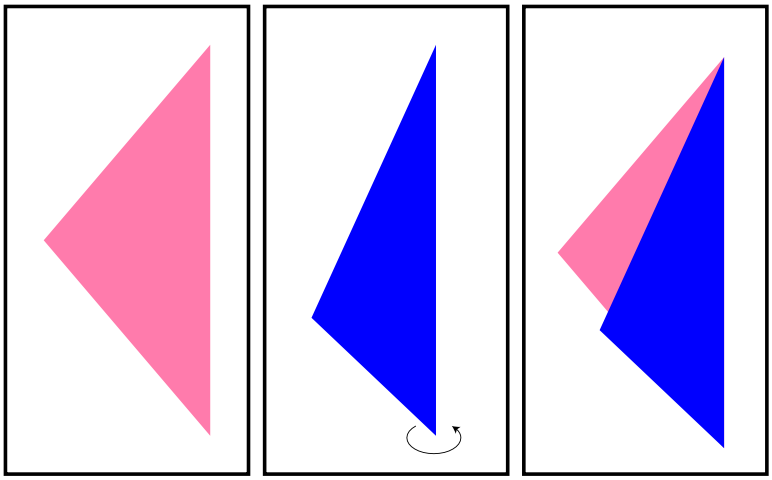


图 9-8 左:第一帧 中:第二帧 右:一二帧叠加

我们先绘制一个三角形，如图 9-8 左所示，然后我们在第二帧的时候让这个三角形绕箭头方向旋转一定角度，并且让三角形颜色变蓝，如果在绘制第二帧的时候没有清除前一帧的数据，那么就会出现画面叠加的情况，所以我们在绘制每一帧画面的时候，可以选择性的清除前一帧所残留的数据，当然有时候因为一些特殊的需求也会刻意的不清除数据。

目前我们在绘制前需要清除的数据有前一帧所残留的各像素值和 Z-buffer，前者残留会导致画面出现错误的叠加，后者则会导致深度测试出现不是预期的情况。为了实现这两个功能，我们为 GraphicsLibrary 类新增两个函数 clear_color 和 clear_depth，这两个函数分别用于清除屏幕的颜色和深度缓冲区。

```
void clear_color(COLORREF c);//将屏幕所有像素的值设置为 c
void clear_depth(double d);//将深度缓冲区的所有深度值设置为 d
```

因为在不同的设备上可能有不同的快速清理像素颜色的方法，我们的代码使用了 std::fill 填充显存缓冲区来实现，而 EasyX 则自带了一个函数 cleardevice 用于清空屏幕，同样地读者可以使用其他方法作为 clear_color 的具体的实现。

之前的 clean_depth 函数以后不再使用，我们将其移除。我们在 main 函数中定义一个循环，每绘制一帧画面后间隔 50ms，每绘制一帧之后将 n 平面值减小一点，直到 n 值小于等于 0 时才结束。

```
gl.clear_depth(1.0);//将深度缓冲区中的所有值设置为 1.0，因为我们在 cvv 裁剪中限定了 far 平面的深度值为 1.0，再远的像素不再绘制
gl.clear_color(RGB(0, 0, 0));//将所有像素值设置为黑色
gl.draw();//渲染画面
```

运行程序可以看到一个逐渐变小的三角形。

9.4.2 双缓冲

上一节的程序已经可以使画面动起来了，但是这个程序在一些电脑上执行的时候会存在闪烁或者画面撕裂的现象。因为我们在渲染像素的时候是一个像素一个像素地渲染，在我们渲染一个像素地过程会有较大的计算量，所以同一帧画面可能出现前一部分已经渲染好了，但是后面部分的像素仍然在计算中，这些种种原因导致了画面撕裂或者闪烁。

为了解决这个问题，人们引入双缓冲的概念，如下图 9-9 所示：

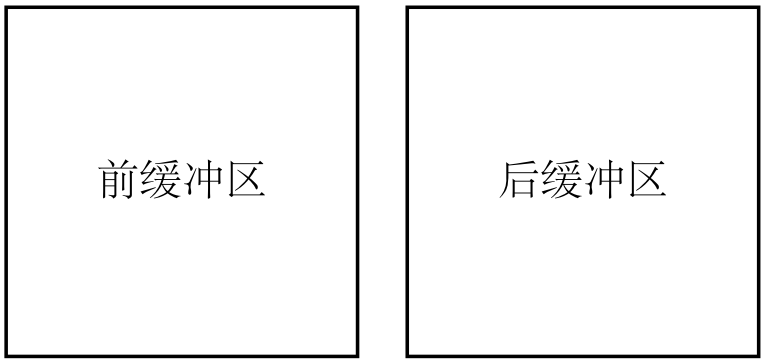


图 9-9 双缓冲示意图

把屏幕绘制区域分成前缓冲区和后缓冲区，前缓冲区是直接显示在屏幕上面的显存区域，而后缓冲区和屏幕没有直接联系，可以是一段普通的内存区域，我们之前绘制像素的时候都是直接往前缓冲区写数据。而现在我们可以把要所有对像素绘制的操作都往后缓冲区绘制，等绘制完毕后再将后缓冲区的数据复制到前缓冲区，因为复制缓冲区消耗的时间远远小于像素渲染的时间，这样就可以大幅度的减少画面撕裂或者闪烁。

本节代码在\part3\src\chapter9\double_buffer 中，我们先在 GraphicsDevice 这个类中定义一个缓冲区 backBuffer，在构造函数中将其初始化为和前缓冲区一样大小。

```
backBuffer = new DWORD[width * height];
```

需要注意的是上述代码是在 windows 中使用 EasyX 时编写的代码，因为 Linux 的

FrameBuffer 和 EasyX 的显示缓冲区结构略有不同，所以代码也稍有不同，但是原理都是一样的，有兴趣的同学们自行学习一下。

我们把 GraphicsLibrary 中所有对前缓冲区的操作都修改为对后缓冲区的操作，首先我们将对 SetPixel 函数的调用修改为 SetPixel_Back，然后将 clear_color 函数的实现也修改为对 GraphicsDevice::clear_color_bcak 进行调用。我们在 GraphicsDevice 和 GraphicsLibrary 都新增了一个函数 flush，GraphicsLibrary 中的 flush 调用 GraphicsDevice 中的 flush，这个函数调用 std::copy 将后缓冲区的数据复制到前缓冲区中。在 OpenGL 等图形库中这个函数通常叫做 Swap,用于交换前后缓冲区，而我们的代码只把后缓冲区复制到前缓冲区，没有实现交换功能，所以取名为 flush，同学们可以运行一下代码看看效果。

第10章　矩阵

说起图形学，一个必不可少的数学工具就是矩阵，通过矩阵可以让我们的坐标变换显得更加的简洁，需要注意的是本章中所有对顶点的运算都是采用：[运算矩阵]×[顶点列主序矩阵]的表示形式。

10.1透视投影矩阵

我们在第八章的时候把透视投影方程修改成为下面的样子：

$$x' = \frac{2nx}{r-l} - \frac{r+l}{r-l}z$$

$$y' = \frac{2ny}{t-b} - \frac{t+b}{t-b}z$$

$$z' = \frac{f+n}{f-n}z + \frac{2nf}{n-f}$$

$$\omega' = z$$

如果我们将原始三维空间中的坐标点 p(x,y,z)写成齐次坐标的形式 p(x,y,z,1),并且用一个4×1 的矩阵表示他，则可以写成：

$$p = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

如果我们将投影公式写成如下矩阵：

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{2nf}{n-f} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

可以得到如下等式：

$$p' = \begin{bmatrix} x' \\ y' \\ z' \\ \omega' \end{bmatrix} = \begin{bmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{2nf}{n-f} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

我们可以用矩阵方便的表示投影变换，而上面由投影公式修改而来的矩阵就叫做透视投影矩阵。

10.2缩放矩阵

一个三维点以原点为中心缩放的公式为：

$$\begin{cases} x' = ax \\ y' = bx \\ z' = cx \end{cases}$$

其中 a、b、c 为 x、y、z 三个轴方向上的缩放倍数，所以我们同样可以得到一个矩阵表示该缩放运算：

$$\begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

则可以得到如下等式：

$$p' = \begin{bmatrix} x' \\ y' \\ z' \\ \omega' \end{bmatrix} = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

10.3平移矩阵

设一个三维点的平移向量为$\vec{h} = (a, b, c)$，则新顶点的计算公式为：

$$\begin{cases} x' = x + a \\ y' = x + b \\ z' = x + c \end{cases}$$

我们同样的可以用如下矩阵表示平移变换：

$$\begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

10.4旋转矩阵

设一个三维点 P(x,y,z)以 z 轴为中心绕 z 轴旋转，角度为θ，则新顶点的坐标计算公式为

$$\begin{cases} x' = \cos\theta\, x - \sin\theta\, y \\ y' = \sin\theta\, x + \cos\theta\, y \\ z' = z \end{cases}$$

于是我们同样的可以用矩阵表示该变换：

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

并且我们可以用类似的方式得到以 x 轴为中心绕 x 轴旋转α的矩阵为：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

以 y 轴为中心绕 y 轴旋转β的矩阵为：

$$\begin{bmatrix} \cos\beta & 0 & -\sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

需要注意的是，上述的旋转矩阵适用于右手坐标系，而我们目前的运算都是在左手坐标系中进行的，这样会导致围绕 Y 轴旋转的方向变反，围绕 X、Z 轴旋转的方向会不会反请同学们思考一下。在日常的学习中右手坐标系使用的情况更多，但是如果我们能弄清楚这些变换的数学知识，那么左右手坐标系对我们来说并没有本质的区别。

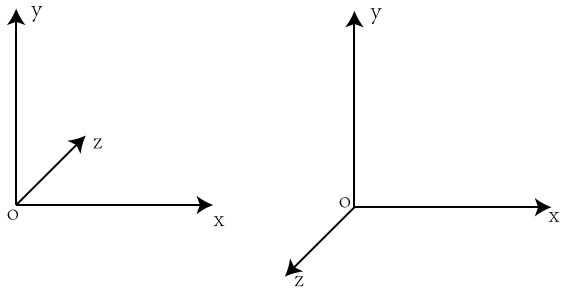


图 10-1 左: 左手坐标系 右: 右手坐标系

10.5坐标变换的一些技巧

如果我们想让顶点 P 以点$c(c_x, c_y, c_z)$作为中心缩放，但是上面介绍的运算是以原点作为缩放中心，所以我们可以先把顶点 P 按照向量$(-c_x, -c_y, -c_z)$平移，然后进行旋转，旋转完毕之后再按照向量(c_x, c_y, c_z)平移回来即可，缩放操作也可以先将旋转轴平移至和坐标轴重合，然后旋转，最后平移这种类似的操作步骤。

设顶点 P 先后经过 AB 两个变换，则可以表示为:

$$P_a = AP$$

$$P_b = BP_a$$

我们可以将这两个变换合并成:

$$P_b = B(AP)$$

使用矩阵乘法结合律，上面这个式子同样可以写成:

$$P_b = (BA)P$$

那么一个很好玩的结论就出现了，如果我们要对顶点进行 n 个变换，那么我们可以将这 n 个变换的矩阵先相乘，最后和顶点相乘即可，即:

$$P' = M_1 M_2 \dots M_n P = (M_1 M_2 \dots M_n)P$$

10.6完成旋转动画

因为本章知识涉及到矩阵运算，这需要一丢丢的线性代数知识，所以代码我就不进行仔细说明，但是所有使用矩阵来运行的计算都可以将矩阵拆开硬算，只是因为矩阵可以更方便的表示而已，不用矩阵同样能完成同等操作。

本章代码在\part3\src\chapter10\matrix 中，我们新增了一个类 Matrix 用于完成一些矩阵运算。

在 Matrix 中增加了用于矩阵相乘的函数和旋转平移缩放等矩阵生成函数，可以传入指定参数后快速的生成矩阵。

对一个顶点绘制的过程中，通常是先对顶点进行旋转、平移、缩放等仿射变换，然后将顶点坐标进行透视投影变换。模型变换的矩阵通常叫做 Model 矩阵，将世界坐标系变换到相机坐标系的矩阵叫做 View 矩阵，透视矩阵通常叫做 Perspective 矩阵，所以通常将顶点 o 的透视投影变换用如下公式表达：

$$o' = PVMo$$

如果我们将 o 的表达式从$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$修改为$[x \quad y \quad z \quad 1]$，则前面所有的矩阵也转置一下即可，投影公式也变成$o' = OMVP = O(MVP)$，这就是 MVP 矩阵的由来。因为我们目前将世界坐标系和相机坐标系重合，所以就没有用上 View 矩阵。

因为 P 矩阵通常是不变的，所以我们把它定义为全局变量，而且不对其进行修改

```
Matrix p_M = Matrix::Perspective(-3.2, 3.2, -2.4, 2.4, 1, 10);//透视投影矩阵
```

在 for 循环刷新动画的过程中，我们将旋转中心点先平移至原点，然后旋转一定角度后平移回来，就得到了一个旋转的三角形。

```
auto move_1 = Matrix::Translate(0, 0, -4);
auto rotate_1 = Matrix::RotateY(i);
auto move_2 = Matrix::Translate(0, 0, 4);
```

这三个矩阵合在一起就是 M 矩阵，我们用 P×M 得到 MVP 矩阵。

```
mvp = p_M * move_2 * rotate_1 * move_1;
```

旋转动画到此处已经结束。

第11章　走进图形学的大门

如果读者能够学习到这里，说明已经掌握前面这些知识了，我们在前面所学的知识已经足够做一个有模有样的 3D 渲染器。本章将会介绍一些提升渲染画面的知识，让我们渲染出更好看的画面，因为本章内容涉及到很多与渲染无关的内容，如图片和 3D 模型文件的解析，这些可以自行在网上查阅资料或者参考我的代码，所以我就不做代码解释了。通过本章的学习，我们也勉强摸到 3D 图形学的大门了。

11.1漫反射光照模型

我们之前渲染的画面都是将纹理的颜色直接投影到屏幕像素，这种投影方式缺少了一点明暗质感，如果我们渲染一个纯色的模型，在屏幕上面将会分不出模型的边界。在图形学中有一个光照模型叫做 Phong 光照模型，这个模型原理比较简单，实现起来也很容易。

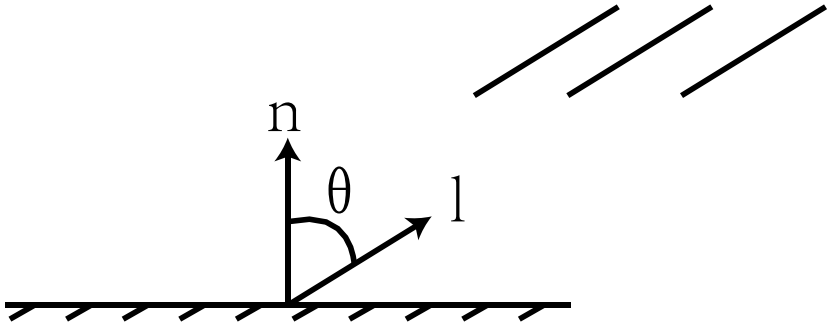


图 11-1 Phong 漫反射模型

假设现在有一平面被一组平行光照射，我们使用$c = n * \cos\theta$表示该平面的亮度，其中 c 为亮度,n 为平面法线，θ为平面法线和光线向量的夹角。在这个公式中如果令$|l| = 1$，那么光照公式可以写成$c = n.l$，两个向量点乘就可以得到平面亮度。

当然仅仅使用这个公式还有一些问题，如果平面和光线向量夹角接近 90°时，平面将会变得非常暗，所以可以加上一个环境光照强度 a，最终平面的亮度为:

$$c = n.l + a$$

因为透视校正插值是原始三维空间中的线性插值，所以我们可以将法向量信息附加到顶点属性中，如下图所示:

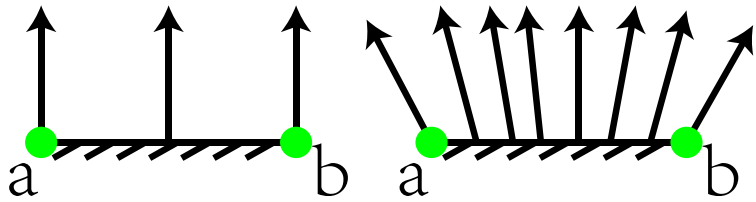


图 11-2 法线的插值结果

如果 ab 两个顶点的法向量方向一致，那么在平面上任意一点的法线都和 ab 两点的法线方向一样，如果 ab 两点法向量不相等，则平面中间的法向量为 ab 两点的插值结果。我们在图 11-2 的右图中可以看到，如果 ab 两点法向量方向不同，则法向量方向在平面上是均匀变化的，利用这个特性可以模拟一些曲面的光照效果。

11.2纹理生成

我们在之前的渲染过程中都是使用一个数组作为纹理图像缓冲区，这个缓冲区只有 4 个像素，如果所以我们可以使用某些方法把图片转换成数组，这样我们就可以使用任意图片作为纹理贴图了，因为图片格式各不相同，所以我做了一个函数可以把 24 位无压缩的 bmp 位图转换成 c 语言数组，有兴趣的可以自行学习图片格式和解析。

11.3模型读取

通过 3ds max 等工具建模之后导出模型比我们手动计算顶点设置 vbo 更加方便，所以我做了一个函数读取 obj 模型文件，需要注意的是 3D 建模软件导出的模型通常都是使用右手坐标系，而我们使用的坐标系是左手坐标系，所以在读取顶点坐标和法向量的时候需要将 z 分量取反，有兴趣的同学可以自行学习 3D 模型文件的解析。

11.4最后一个练习题

我们在第二章介绍绘制三角形的时候曾经遇到一个问题，因为三角形的坐标点是使用浮点数表示，而我们在光栅化三角形的时候使用的像素坐标都是整数，这样会导致像素的坐标计算出现一些问题，比如在扫描线递增的时候 x 坐标变得非常大或者非常小，导致得到一个不属于三角形内部的坐标，我们采用的解决方法就是将扫描线的 sx 和 ex 坐标限制在三角形内部。现在我们的渲染器代码在像素属性计算上还有一个类似的问题，如果同学们学到这里，根据我下面的描述应该可以很轻松的修复这个问题，所以这里将作为一个练习题，希望同学们先自己尝试解决问题再看答案。

问题描述：为了更容易观察问题，在 code\exercise 目录下放置了一个 a.obj 和 a.bmp 文件作为模型和纹理，如果同学们用这个模型纹理替换掉 code\part3\src\chapter11\render\model 中的文件,可以观察到如下的渲染结果：

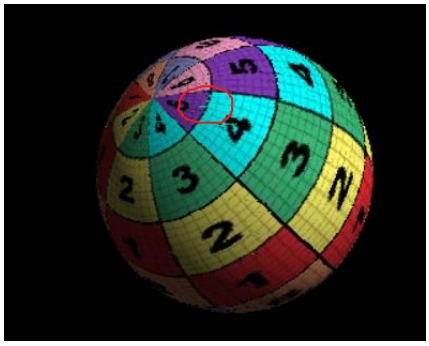


图 11-3 属性值计算错误

原因：因为在光栅化三角形的时候，我们在计算属性值的时候为了加快计算速度，采用了增量计算的方法，如下图所示：

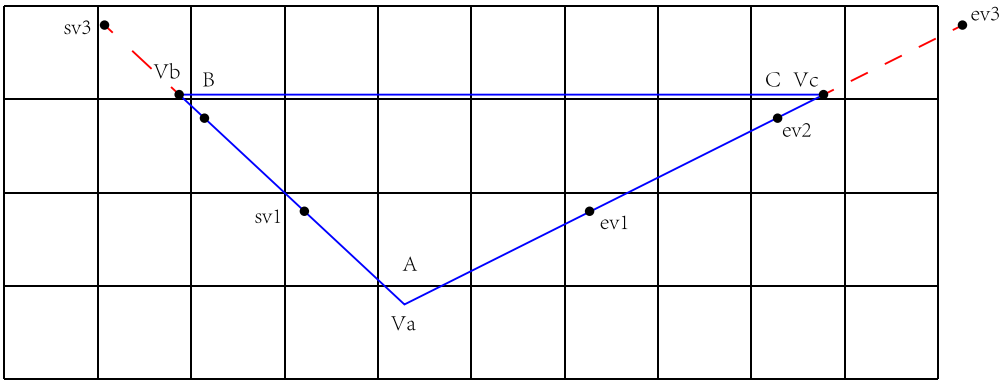


图 11-4 错误原因示意图

我们可以简单的看一下示意图，我们使用 y 的增量计算同一条扫描线上面的起始属性 sv 和结束属性 ev，我们在三角形的边上面计算 sv 和 ev 的增量Δsv、Δev，然后使用 sv1=Va+Δsv，ev1=Va+Δev，我们使用此方法得到每条扫描线的起始和结束属性。但是如果上图中的 B 点和 A 点 Y 值距离(垂直距离)不是整数倍，则会得到错误的属性值，如上图中 sv3 是我们使用增量累加出来的属性值，而 Vb 则是三角形的实际属性值；对于点 C 也一样，如果和边起始点的垂直距离不是整数倍，也会出现同样的问题。在 part1 部分中我们渲染的错误原因就是这么简单，而在后续章节使用透视校正插值后，我们采用类似的方法计算 t 的增量，那么错误的原因就是 t 值累加出错，然后我们使用错误的 t 值进行透视校正插值。

我们的代码在光栅化三角形的时候会把三角形分解成如下两种三角形，然后再一一渲染，根据上面的分析，在最上面的扫描线中都可能得到错误的属性值(红色顶点和红色边)。

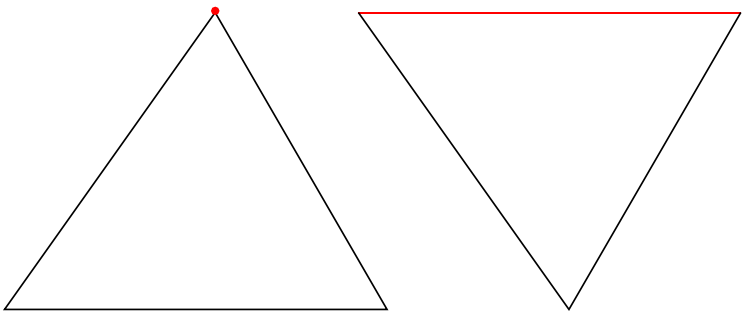


图 11-5 渲染三角形的两种情况

解决方法：这个问题修复起来比较简单，通过上面的解释同学们应该能够自行解决问题，所以作为一个练习题给同学们处理，如果修复正确，将会看不到毛刺现象(不要翻到下一页，会看到答案的)。

答案：修复 bug 之后的代码在\code\part3\src\chapter11\exercise 中，我们在 GraphicsLibrary.cpp 只新增了两行代码就修复了这个问题：

```
t_left = min(1, t_left);//非常简单的 bug 修复，因为按照分析，t 值在递增后可能大于 1(在计算最后一条扫描线的时候),所以我们将 t 限制在小于 1 即可
t_right = min(1, t_right);
```

修复原理在注释中已经说明，我们对 t 值进行了限制，如果要修复 part1 中的问题，也采用类似原理，限制住属性值 v 的范围，将其限制为图 11-4 中 Vb 和 Vc 即可，所以我就不对前面的代码进行说明，如果同学们有兴趣，真的得自己修复问题了。

本书配套代码已经上传至 github.com/yangzhenzhuozz/Renderer, 如果发现本书内容的错误，请发送电子邮件至 775697360@qq.com。