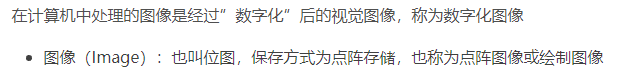
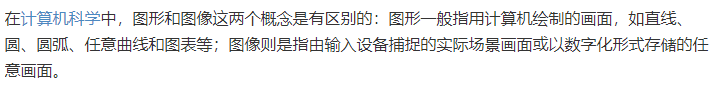
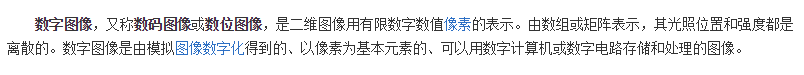
图像基础

什么是计算机图像？这是个好问题。我看过无数的定义，下面随便抄几个出来：







菜逼们，看到这几个解释，明白了吗？清楚了吗？不清楚？背下来，考试的时候这就是标准答案！

可能是我天赋不大够，老实说，我的图形图像之路充满了坎坷。看到以上这类解释，对我而言，就是用一些我不懂的术语，去解释另外一个我不懂的东西。就像第一个解释，刚入门的菜鸟，是不是得再去深入理解什么是“点阵”？点阵储存是个什么鬼？

第二个解释，“以数字化的形式储存”，怎么以数字化的形式储存？

第三个解释，是百科上面的解释，别人能不能理解我不知道。如果在我刚入门的时候去解释这个东西，我只能是一头雾水。

这里，我决定用我自己的语言和脉络，来解释一下什么是图像。解释图像之前，我来先对现代科技的一些基本的概念做一些我自己的阐述，不保证正确。

我认为，现代科技的一个重要基础，就是量化/数字化。什么是量化，而为什么需要量化？举例：在没有量化的时候，今天冷不冷？冷！多冷？好冷！

而量化了的时候呢？今天冷不冷？冷，多冷？-30度。

现代科技的基础，是把距离、重量、力量、能量、温度、角度、气压、电流……一切你能想到的东西，做了量化/数字化。只有数字化以后，才能参与计算。

图像的实质，是颜色的量化！

颜色的量化有好几种方式，最常见的是RGB，其他较常用的还有HSL/HSV，这个主要是给美术用的。RGB是纯数字，美术看不懂，但是美术能看懂色度，饱和度，亮度，所以增加了一个这东东，直接套用公式转换即可。还有视频里面常见的YUV，诸如此类的东西一大堆，这里不做详细的介绍，这里主要讲RGB。

RGB的意思是：任何颜色可以分解成红、绿、蓝的混合。这就是色光三原色原理。那么问题来了，颜色是如何保存的？这里，我大概介绍一下颜色的发展历程。

大概在1996年，我开始见到第一台286电脑。那会显示器是黑白显示器，颜色只有黑白两种颜色，编程的时候，颜色的描述只有两个数字：0和1. 0是黑色，1是白色。

接下来，在386的年代，用Qbasic编程，颜色就多了一些，我最早的时候，用过16色编程，0是黑色，1是蓝色，4是红色，15是白色（年代久远，有可能记错了）。我第一个比较大型的网络游戏，那是1997年用Qbasic写过一个中国象棋，基于Novell网络。

接下来的几年，个人计算机迅猛发展，短短几年时间，从16色就进入了16位色，32位色的年代。

这里，只讲解一下 16位色，然后估计很容易久能想明白32位色，8位色。回到之前讲过的C++基础，16位就是2字节。那么颜色是怎么保存的？可以是565，5551，655，556，如果是自己做存取的话，你想怎么搞怎么搞。通用的话我用最多的是565，5551.后者带一个alpha通道。

01111 110011 11001

红 绿 蓝

所以，16位色理论上能描述的颜色数是：

2^5 \* 2^6 \* 2^5 = 2^16，也就是65535种颜色。

那么，32位能描述多少颜色？是2^32那么多吗？不是的，能描述的是2^24那么多。所以说，24位颜色跟32位颜色其实是一样的，只是32位颜色多了一个8位的alpha通道。

这里，讲讲Alpha（透明度）。我刚入门的时候，那还是十几年前。美术讲什么alpha，我一头雾水，只知道这东西是透明值。只是听说是0-255之间。至于透明度是如何起作用的，完全不知道。

在现在各种资料满天飞的年代，理解这类东西容易了太多。

首先，alpha是透明度的量化！可以是0，1两个数字，可以是2位（8位的颜色图，就有2222的格式，用2位描述透明度），可以是8位。

透明度只有在混合的时候才有意义。计算公式非常简单：假设一个8位的alpha，透明度为100，当前图片颜色为Col1，背景颜色为Cold

，公式为：

Col = Col1 \* (float(100) / float(255)) + Col2 \* (float(255 - 100) / float(255));

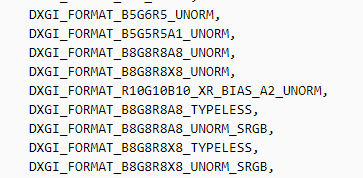
什么？这么简单的公式还复杂？那么我们分开两步：

Float alpha = 100 .0f/ 255.0f;

Col = Col1 \* alpha + Col2 \* (1.0f - alpha);

如果这么简单的公式都看不明白，放弃吧，图形学绝对跟你无缘。

看明白了这样简单的数据储存，很多看起来牛逼哄哄高大上的东西，就会发现也就那么回事。



看图，这是微软的图片格式定义，使用dx11，dx12的时候都会用到，看起来很高大上的样子，新手看到就容易懵逼。其实懂了图像原理，一切看起来都是如此简单。上面两个就是16位颜色图，一个是565，一个是5551.下面的都是各种32位标准颜色图。至于后面的什么typeless之类的，无非就是存储的格式罢了。例如：你创建了一张贴图，等于申请了一块显存，那么你存储的时候，可以是以整数储存，可以是无符号整数，也可以是无类型，我的理解，这个typeless跟c++里面的void\*指针是一样的货色。使用的时候，直接memcpy，后续再指定类型。一般情况下，你用不到那么多。

DXGI定义了上百种颜色，其实一般人用到的不多，后续讲到纹理、贴图相关内容的时候再详细描述，这里还是回到图像的基础原理。

图像缩放。

为什么需要单独出来讲图像的缩放？这有什么好讲的吗？大部分菜逼程序员，都可以随意通过一个scale之类的接口，轻松实现图片缩放。这里面有什么诀窍吗？

我认为，大部分菜逼，为什么水平一直那么菜，就是没有刨根问底的精神，热衷于各种上层调用。你调用一个接口完成了的功能，那是你完成的吗？你真的理解了如何做一个图片的缩放吗？如果你亲自实现了各种图片的缩放，那么，计算机图形学里面所谓的“图片采样”，对你来说，将没有什么秘密可言。你就能深入理解到为什么图片的采样有可能会闪烁，听起来很高大上的mipmap是干什么用的，我们为什么需要这个东东。

这里，我以图片缩小为例，详细讲述一下缩小的过程。

假设图片1，宽高分别为W1，H1。缩小到图片2，宽高分别为W2，H2。

那么，问题就变成了：已知宽高分别为W1，H1的图片，求宽高分别为W2，H2的图片任意像素的颜色。

图片2中，假设某像素坐标为w, h。那么，该像素的颜色怎么计算？

最简单的计算方法：

Float u = w / W2;

Float v = h / H2;

Col = sample(Texture1, u, v);

这点代码，是不是看起来越来越熟悉了？我靠，这怎么看着那么像shader里面那个像素着色器的那个图片采样？

我们再来实现一下sample这个函数：

Col sample(Texture1, u, v)

{

Int x = u \* W1;

Int y = v \* H1;

Int Pos = y \* W1 + x;

Return Texture1[Pos];

}

由于这点代码全在word里面手打，大小写，缩进之类的就不要吐槽了，伪代码大概看看就好。

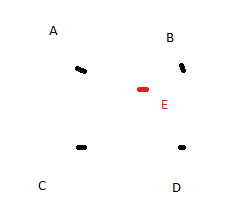
看到了吗，这就是一个最简单的采样。图片缩小了，但是找到同比例的地方，也就是u，v坐标的地方，把最近的点的颜色取出来即可。

那么，以上的代码，可以看作最简单的最近点采样吗？所以说我必须强调自己动手的必要性。这当然还是错的。错在哪里？因为浮点数直接转了整数，例如3.725，直接变成了3，自然不是最近点啊，最近点当然是4。所以，计算x，y的时候，还需要做一个浮点数的四舍五入。这点代码会写吗？不会？那么看下面的

Int x = (int)floor(u \* float(W1) + 0.5f);

这是标准写法。你要是不想写那么标准也可以，随意。

这已经是图片采样的真谛了吗？远远不够。最近点采样，显然会造成其他颜色的丢失。最常见的，是线性采样。何谓线性采样？如图：



假设ABCD是周边四个像素，而E是根据uv坐标算出来的位置，假设uv的坐标的小数部分为(0.7,0.25)。怎么计算E点的颜色？还是线性插值，请看下面的公式：

先纵坐标插值：

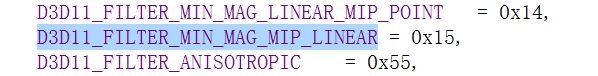
Col1 = A \* 0.25 + C \* (1 - 0.25);

Col2 = B \* 0.25 + D \* (1 - 0.25);

再横坐标线性插值：

Col = Col1 \* (1 - 0.7) + Col2 \* 0.7;

这个计算是如此的简单优雅，线性插值贯穿于整个图形学的始终。如果菜逼还是看不懂，放弃吧，图形学绝对不是你能够染指的东西。



这里是D3D11的一些采样设置，主要就三个，一个是最近点，一个是线性采样，一个是各向异性。前两个这里已经讲得很清楚了，后续一个后面讲到3D的时候再讲，因为那个已经跟3D有关，不仅仅是2D图片的事了。

讲到这里，估计很多大佬已经兽血沸腾，拍案叫绝，欲罢不能了。以为自己已经掌握了图片采样的真谛，已经无所不能。

还是那句：太天真了。

你用这个算法，把一张1000 × 1000的图片，缩小到100 × 100试试看？保证你惨不忍睹。

想不明白？想不明白等我介绍到纹理的时候再详细讲。现在全讲完了，后面我还怎么混？

什么？秒懂？大佬，你的天赋，有明日之星的潜质，以后混发达了，记得带带我，我会端茶倒水，我还会喊6666。

图片压缩。

最早最初级的压缩算法，叫RLE压缩。

RLE算法非常简单粗暴。假设一张图片有100个像素，前50个像素颜色是一样的，那么只需要记录这个颜色，并且记录这个颜色的区域即可。翻译成中文就是：0-50，红色。

这算法在早期颜色数很少的时候，非常流行。我没记错的话，微软在BitMap的格式里，还预留了RLE压缩。可是，随着32位真彩的流行，这算法已经完全无用。

为什么无用？很简单，因为早期颜色数量只有两种，16种的时候，颜色相同是很容易的事，所以这算法可以流行一时。颜色细分之后，在一张图片里面，要找到两个颜色完全一样的像素，已经是千难万难。看着一样的颜色，实际都有微小的差别。因此这算法淘汰也就是情理之中了。

调色板压缩。

调色板压缩算法，可以勉强看作是RLE压缩的升级版。在早期的2d游戏里曾经风靡一时。例如云风的成名作：风魂，应该就是用了调色板跟RLE压缩。

原理也比较简单：



看看这个图片，看着颜色不错吧，可以用16位颜色，也可以用32位颜色。由于2D游戏里面的人物，往往是N多图片的集合。例如一个人物可以有10套动作，一套动作可以有8个方向，那么这就是80张图片。如果一个动作有16帧，那么就是80 × 16那么多的图片。

各位大佬留意到了没有？虽然图片颜色看起来挺细腻的，但是颜色数其实可能真不多。假设我们把颜色数控制在256以内，颜色值是32位，那么我们可以创建一个256 × 4（32位4字节）的调色板，里面记录了这小于等于256种颜色，然后图片里面，则只需要用一个字节记录这个颜色的索引即可。

大概是这样：

颜色1，颜色2……颜色255.

图片里面，则是：

5， 5， 5， 5， 6， 8， 105， 30， 234……

这里的5，是个索引，意思是去调色板里取第五个颜色值。这样搞，索引可能会有很多相同，那么又可以做RLE压缩了。

大概也就是这个意思。

什么？看不懂？这都看不懂，网格模型里的顶点，索引坐标计算还能看懂吗？原理都是一样的啊。其实，学习图形学的时候，很多细节算法必须一点一点搞懂，因为很多算法其实都是相通的，所以千万不要出现看不懂就放弃跳过，总有你跳不过的时候。如果这样的心态，放弃吧，这种心态，一辈子搞技术无望。

哈夫曼编码。

当我看到RLE算法的时候，我觉得这算法挺好的，简单粗暴。但我看到哈夫曼树的时候，我震惊了，世上竟然有这么聪明的人，能想出来这么好的东西。我以为这已经是图像编码的顶点。可惜的是，这基本排不上号：（。为什么？因为排得上号的都是有损压缩啊。对于图像压缩来说，有损才是方向正确。255和254的白色，我认为人眼压根分不出来，这一点点损失算得了什么？

毫无疑问，哈夫曼编码是史上最优无损编码之一，只是跟有损相比，就没有可比性了。因此，在图像压缩里面，只有极少数的地方有应用。我依稀记得，JPEG的编码里，只有较少的篇幅是用到了这个编码，可见无损编码在图像压缩里面的地位：（

好了，是时候讲一下哈夫曼编码的大概原理了。鉴于我绝不打算抄袭别人的东西来凑字数，所以我只讲原理。就我个人而言，能通俗易懂的把原理讲清楚，那么再去读一大篇带算法甚至带复杂数学公式的文档，会容易很多。

举例：“四是四，十是十，十四是十四，四十是四十。”

假设毫无压缩，一个字是两个字节（Unicode编码，按最低来算，有的编码还不止两个字节），这里一共16个汉字，三个标点，那么就是19 \* 2 = 38个字节。

那么很容易想到，如果我先找出重复次数最多那个字，假设是“四”，那么我用一位来描述，重复次数第二多的字，假设是“十”，用两位来描述，重复次数第三多的字，假设是“是”，用三位来描述，以此类推。

看到这里，各位大佬眼睛一亮，哇靠，好牛逼，好像真的可以啊。我跟哈夫曼大神的距离其实没有那么远啊，我一下子就想明白了啊。事实上有那么简单吗？太天真。

这种思路很容易想到，为什么直到大神出现之前就没有人解决呢？因为有很多问题需要解决。

全是010101000111之类的二进制编码，如何断句？说穿了，010101000111这么一段东西给你，你如何直到里面是几个字符？是前两位代表一个，还是前四位代表一个？说穿了，就是如何让一段连续的二进制数据能无歧义的解释出来？

菜逼们这时候可能会逼逼了，我中间加分隔符啊！还能这样想的，估计没看我之前写的c++基础，就算看了，也没看懂。任何一个分隔符，最次用一个ascii码，例如什么空格，什么下划线，至少是8位。好不容易弄出来个压缩编码，你直接加分隔符，还能好好的玩耍吗？

这里，涉及到了一个哈夫曼树。哈夫曼树的核心，主要是解决了两个问题：第一个，把最常用的节点权重最大，也就是说，最常用的节点，用最短的编码。第二个，能够把各个节点无歧义的表示出来。

菜逼跟大神的距离，在于菜逼也能轻易想出来要飞上天，需要装一个翅膀。但是大神能够设计出这个翅膀需要多大，频率多快，能耗多少，才能把人飞上天。

这里，我就不详细介绍实现细节了。再详细介绍，估计也是去网上抄图抄书了，这不是我的本意。有兴趣的，自己去随便找找，这类烂大街的资料已经足够多了。

YUV（这个到底定义为编码，还是颜色空间，还是\*\*\*？我犹豫了）

在本章的开头，我已经讲过：图像是颜色的量化！所以，如果按照这个理论，YUV其实也是颜色量化的一种方案，分别是亮度，色度，饱和度。

这里，为什么要单独介绍这一部分？因为人眼很好欺骗，因为这也是几乎所有图像压缩（视频）都会夹带的方案。我没有直接说所有，是因为我怕被打。就我知道的，视频几乎都做了这个压缩。

能这么做的重要原因：人类眼睛对亮度敏感，对色度，饱和度没那么敏感。（我记得以前看过文章，说狗的眼睛里所有世界都是黑白的。假设这是真的，如果我们给狗做图片压缩，不需要什么RGB三通道了，直接一个亮度就够了）。

好了，先看看下图：



这是随便网上找的一张图片，大佬们别告我侵权。

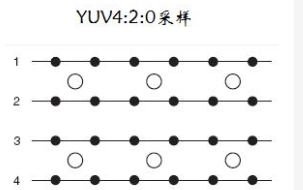
左边是正常的RGB图，假设是24位。右边是灰度图，那么就是8位就够。

大佬们，有没有发现，其实右边这张图，看起来就可以了？几十年前看黑白电视，玩黑白游戏，不也是很高兴？

是的，人眼就是这样容易满足，这才是YUV能够实现的基础。

理解了原理，大概谈一下实现。这里我只谈非常常用的YUV420，例如什么知名的H264，H265，各类常见的视频，第一步都是什么RGB转NV12，NV12我没记错的话，就是YUV420，懒得去查了，被喷了再改。

如图：



上图来自网络，别搞我。

这图的意思是：四个Y公用一组UV。

所以，假设四个像素，原本是24位，那么四个像素就是24 \* 4 = 96位。

YUV420呢？那么就是4 \* 8 + 8 + 8 = 48，直接压缩了一半！

什么？我已经这么努力，解释得这么清楚了，还是看不懂？那么，我已经无能为力了大佬。放弃吧，图形学对你太难。

看到这里，估计大家都懂了，为什么几乎所有的视频都采用了这种压缩。因为足够简单粗暴有用。另外一点，还可以跟以前的黑白电视机兼容，黑白电视机只需要一条亮度通道即可。

一些图像压缩思路介绍。

傅里叶变换（FFT）。

其实傅里叶变换是计算机图形学里面必须绕不过的一道坎。然而，我其实一直在纠结到底放哪里会比较合适。因为用到这个的地方特别多，例如做一个水面的波动noise，常用的要么是perlin noise，还有一个就是FFT noise了。Perlin noise鼎鼎大名，我记得那是拿过奥斯卡科技奖的大佬，perlin noise应用于图形学的方方面面，我在我自己的github里面还用这个生成过云彩，效果还很不错。有兴趣的可以去看看。

FFT跟图像压缩有什么关系？

开始这个话题之前，先说一些我自己想过的乱七八糟的东西，不保证有用，也不保证正确。

我们轻易能想到一些东西：

1. 图片绝大部分都是有规律的。例如蓝天白云的图片，主要就是蓝色和白色，甚少乱七八糟的其他颜色。
2. 图片大部分都是有一个过度的。例如蓝天白云，仔细打开图片观察，蓝色跟白色之间，再少都有一个过度。

那么，如果把这些数据弄出来，先转成YUV，用Y值，画一个曲线，能不能用一个函数来描述这条曲线？如果可以，岂不是我只需要记录很少的数据，就能够描述一条像素？或者是描述一个平面的像素？

什么？一张图片太大了，要找出来一个描述这个图片的函数太难？能不能转换一下思路，把图片分块，就找一个函数来描述一个小块？

想法很美好，现实很残酷。但是，这个思路我认为是没有问题的。问题在于，我们得找到一种切实可行的办法，来实现类似的思路。

这个时候，傅里叶变换，傅里叶级数是时候出场了。

首先，我们得变换一下思路：图像的Y通道，无非不就是0-255之间的一系列的数吗？这些数不就可以理解为一段信号吗？

其次，任何信号都可以被分解为基波和不同幅度的谐波的组合，意思不就是说，可以把一条信号分解成N个函数的集合吗？

是的，你没有看错，我们非常常用的JPEG图片，压缩算法就是差不多的原理，只是做了很大的变更。例如傅里叶变换被替换成了据说更好的离散余弦变换（至于为什么更好，其实我也不知道，没深入研究过，此处不好装逼）。至于为什么有损？在专业数学上能讲一大堆，例如人眼对高频信息不敏感，丢失掉高频信息之类的。但是最简单的可以这样理解：整数——浮点数——整数的转换，必然是有损耗的。

好了，图像相关的理论，先讲到这里。其他还有少部分这块相关的，放到纹理、材质里面再讲，那部分已经不仅仅是2d纹理通用的内容了。

图形基础

这里，3D图形终于要出场了。

出场的时刻，我们需要一些铿锵有力的语句，来镇一下场子：

1. 任何曲线，必然可以分解成N条线段！
2. 任何曲面，必然可以分解成N个三角形！

是的，必须要深刻理解了这两个理论，才能更好的去理解相关的原理。

由以上两个理论，我们可以推出推论：

只要有足够多的三角形，我能构建整个世界！

好了，这就是一切图形学的基础，这章讲完了。

什么？菜逼们要来打我？我写书又没收你钱，你凭啥打我。

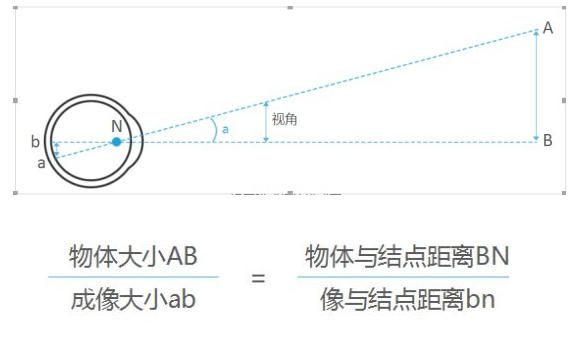
你给钱了？且慢，那我还是要继续讲一下，我单纯吐个槽。我直接把我几十年功力拿出来卖钱，就卖了几百块我容易吗？：（

这里，先不讲网格模型的一大堆东西，那一大堆东西我打算在专门的网格模型章节里面详细讲。这里，先要讲讲光栅化的基本原理。

一个模型，如何显示在屏幕上？

由于任何模型可以分解为任意多个三角形，问题可以描述为：一个三角形，如何显示在屏幕上？

这里面，有一个核心的东西，叫做透视投影变换。这个名词一出，菜逼立马懵逼了。先看下图：



这图不侵权，这是我另外一篇VR光学原理里面的图，放这里也成立。

我们来一个古龙体描述一下：

人眼为什么能看到这个世界？因为这个世界的光线反射进了人眼。

光线是直线传播吗？是的，光线是直线传播。

人眼不是倒立成像吗？是的，但是我这里用正立成像。

你这样装逼真的好吗？不好，但是不装逼我会死啊。

各位大佬，此处就不要用什么爱恩斯坦证明了光速是会弯曲的来打击我了。我承认你是物理学大神了可以不？

在光栅化里，光速默认是直线传播！不要以为光线跟踪里面就不是直线传播？你的物理学只在物理碰撞计算里才有用武之地！

好了，回到主题，光栅化三步曲：

此处先以一个顶点为例。

把顶点转换到世界坐标系，说穿了就是得到三角形三个顶点的世界坐标。（此处以后再细讲）

把顶点转换到镜头坐标系。

把顶点转换到屏幕坐标系。

顶点的定义，在任何引擎里面，基本都是这样的：

class Vector3

{

float x, y ,z;

}

这个坐标变换，一般很多书里，都用两个专业术语吓死你，一个是仿射变换，一个是齐次坐标。

我没入门那会，看到这两货，立马懵逼了，还专门去查是什么意思。那会是2006年，能查到的资料甚少，还不打知道该直接查英文，坑死我了。

别给吓到了，这里稍稍讲一下：

仿射变换，就是一个坐标系变换。假设你家房子，以房子中心为坐标原点，然后定义xyz轴的方向，你可以得到房子任何一个点的坐标。假设你现在要以地球球心为原点，然后再定义xyz轴方向，重新定义一个三维坐标系，那么，如何把你家房子原来坐标系里面的已知顶点，变换到地球坐标系？这就是仿射变换要做的是事情。

在图形学里，这是通过一个矩阵乘法来实现的。一个4\*4矩阵，可以同时描述平移、旋转、缩放。此处不详谈，我估计会在后续的3d数学里面详细谈一谈怎么实现更合适？

齐次坐标又是什么鬼？其实更加简单了。因为矩阵是4\*4矩阵，3\*3矩阵只能描述旋转+缩放，同时描述移动、旋转、缩放的只有4\*4矩阵。那么你用一个4\*4矩阵乘以一个三维向量，就不合适了，所以GPU里，自动就是xyzw，叫float4，w补齐1.

真相就是：你传入显存，只需要传入xyz坐标，然后写shader的时候，直接用float4，不会有任何影响，我经常这么干。

回到透视投影变换。那么，就很好理解了。说穿了，就是先变换到人眼（Camera）坐标系，然后再变换到屏幕。

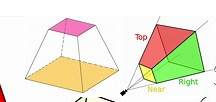
这里面，有一个非常有技术含量的东东，那就是投影矩阵。一次又一次，我都以为我理解了投影矩阵。一次又一次，我发现我的理解是如此的肤浅。

这里，我也不讲一些烂大街的货色，也不讲这个矩阵的推导，只讲一些有点意思的，我认为比较深入的东西。

先定义一下这个投影屏幕坐标系，这里有一个需要转换来转换去的东西，就是屏幕坐标跟投影坐标系。众所周知，屏幕坐标系左上角是0，0点，右下角是最大点，例如右下角是1920，1080. 而对于投影计算来说，这显然是不合适的。所以我们定义的投影坐标系，屏幕中心是0，0点，左下角是-1，-1，右上角是1，1.

这两个基本坐标系的转换计算，贯彻于整个计算机图形学的始终，一定要滚瓜烂熟。

视锥体就是一个梯形椎体。如图所示：



以上图片来自网络。

我们要的是什么？假设这个椎体里（外）有任意一个三维坐标点，我们需要有一个矩阵，这个矩阵乘以这个坐标点，能得到这个点在屏幕上的投影！这个矩阵，就是透视投影矩阵，我们就是需要求这样一个矩阵。

这个过程，很多人都能看懂，比较有意思的地方，在于z轴。

要理解这个z轴，先科普一些比较简单的硬件相关的东西。

早期的GPU，所有render target，输出必须是0-1之间的浮点数。这里可能涉及到一个以后经常用到的名词，render target。这东西你可以先不管，暂时理解为一张输出图片，例如输出到屏幕的图片。中文翻译为“渲染目标”。

z值这个叫深度值，说穿了，就是这个空间点到屏幕的距离值。专门有一个depth buffer来保存这个深度值。这个depth buffer的分辨率跟渲染窗口的分辨率是一样的，如果你渲染窗口的分辨率是1920 \* 1080，那么depth buffer的分辨率也是这个。

这个应该很好理解。假设我距离镜头100米，你距离镜头200米，我肯定在你前面，拍照的时候就把你给挡住了，不是吗？

问题出在这里。深度图保存的距离值，只能是0-1之间的浮点数，而不能是100，200这样的浮点数。

为什么这样？我认为跟GPU的设计有关。当然了，这个设计我认为是经过深思熟虑，并且很合理。在前面C++基础的时候，我有讲过浮点数的储存。内存中的浮点数储存，采用的是IEEE标准，这个储存，会有一个精度问题。忘记了的回头去看我写的c++基础，或者自行百度。

（此处需要加一个注释。在DX里面，depth是0-1之间，而GL据说是-1到1之间，GL我甚少用，不评价，别打我。这里，一切以0-1为准讨论）

采用0-1这类储存，减少了整数部分，能有效提升浮点数的精度。这点很好理解。

问题出在这里，这个depth，如何计算到0-1之间？

很多年来，我一直都是这么干的：

float depth =( position - canmera\_position).length / farplane;

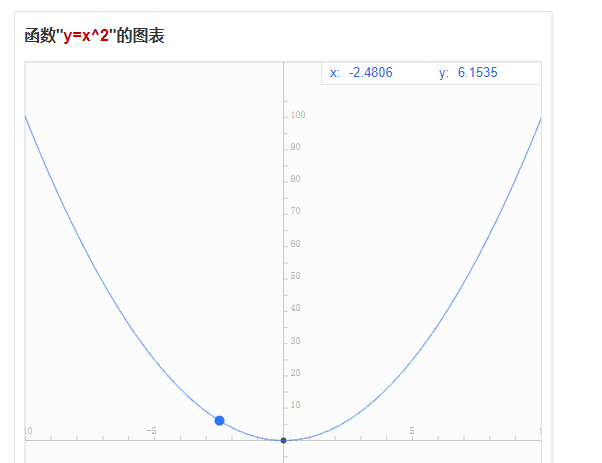
或者是：

float depth =(( position - canmera\_position).length / farplane)^2;

这两个，其实非常好理解。第一个，就是个线性，这就好比，A点到B点的距离是S，那么A和B之间任意一点的距离S1，就保存成S1 / S；不就是0-1之间了吗？

第二个，稍稍复杂一些，得出来的结果做了个平方。这里，估计很多人就理解不了了。所以我多次强调了基础，基础不过关，这里就懵逼了，完全不知道这么干的意义。

看图：



图片来自网络。

在0-1之间的数的平方，疏密程度是不一样的。我们可以轻易发现，数越小的地方，越密集。数越大的地方，越稀疏。这么解释，还是很难理解吗？

那么我们就用数学来直接解释，数字肯定没错。

0.9^2 - 0.8^2 = 0.81 - 0.64 = 0.17

0.2^2 - 0.1^2 = 0.04 - 0.01 = 0.03

显然，0.17 远大于 0.03

再回到之前的浮点数精度，这就催生了一个问题，0.01跟0.02是不一样的，但是0.01^2跟0.02^2可能是一样的！因为浮点数的精度就那么多。

因此，有限的32位浮点数，要尽量的根据你的目的做好利用。

例如，如果距离越远的地方，容易出现了重面闪烁，那么你可以调整这个深度曲线，来让远处的东西更加的清晰。常见的是调整为log函数。

所以这你以为你就是大佬了，你已经掌握了depth深度图的全部？事情远没有那么简单。我不知道为什么GPU渲染计算的时候设计得那么的复杂。我自己在实现阴影图的时候，相对简单很多，就之前提到过的线性或者平方。这里，我也不作详细的描述了，因为后续我能理解到的，全是从一个链接来的。我再来讲解，又会变成了抄书。有兴趣的可以自己去看。

<https://www.sjbaker.org/steve/omniv/love_your_z_buffer.html>

投影矩阵，除了z-buffer比较难以理解以外，还有一些一般人用不到的应用。

典型的例如畸形视锥的渲染。

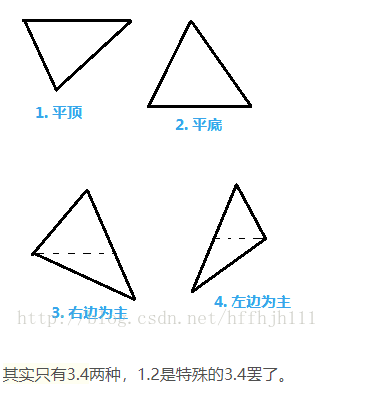
如你所见，所有渲染到计算机显示上上的游戏，基本上全是正视锥。也就是上图那种方方正正，四边都是梯形的视锥。而现实远没有那么简单。

例如VR渲染。各位尝试一下，闭上一只眼睛，剩下一只眼睛左看看，右看看，上看看，下看看。你会发现，眼睛本身就绝对不是一个方方正正的视锥。所以，在VR渲染的时候，这个矩阵的计算，又是另外一个专业领域了。这部分资料，可以在我的VR光学方案里面有提到，有兴趣的可以去看看。这里不再做详细的描述。

关于投影矩阵，这里不再作论述了，继续后面的话题。

综上，我们已经理解了一个顶点，是如何渲染到屏幕上的，关键就是一个矩阵计算，即可以算到屏幕上。那么问题来了，一个三角形，如何光栅化到屏幕上？

看图：



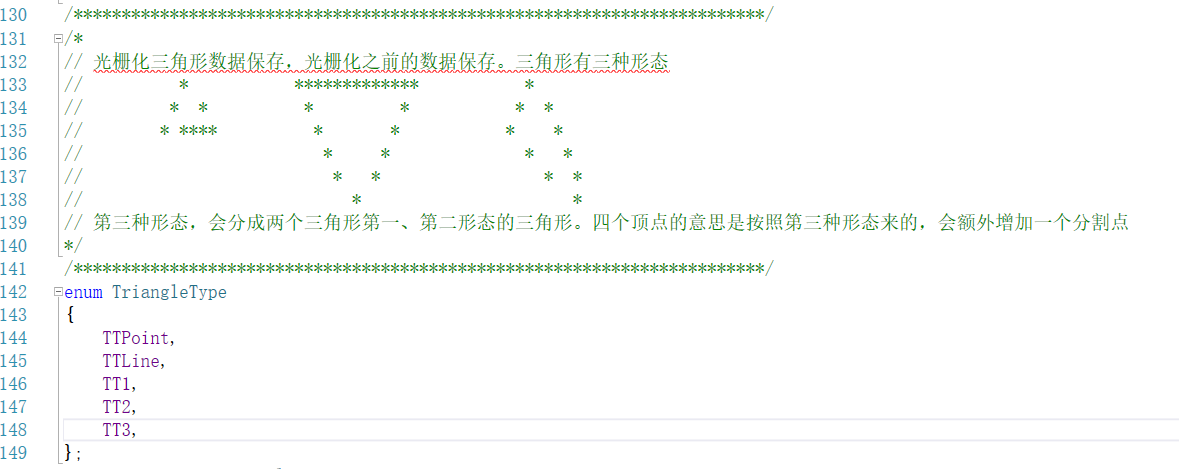
图片来自网络，懒得画了，侵删。

渲染一个三角形，我自己的软光栅分为以下几步（这里会贴一下自己的老代码，别怪我水，代码绝对自己手打，不存在侵权问题，也不存在没经过测试问题）：

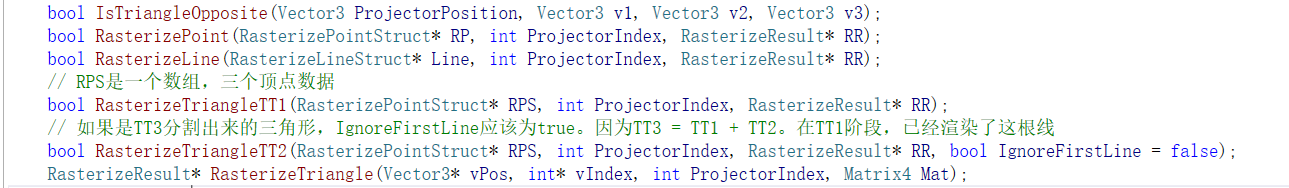
1. 判断面是正面还是反面。反面直接不渲染。



1. 三角形做一下分类：



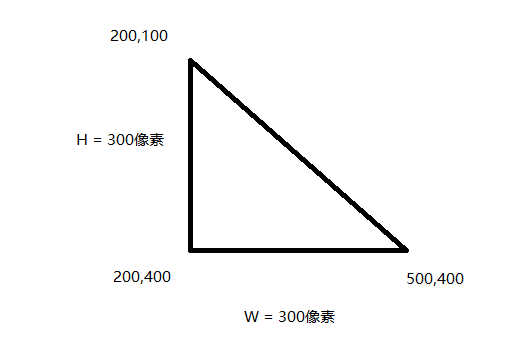
分别实现几个渲染函数，分别为点、线，第一种三角形，第二种三角形，第三种三角形。



这里讲一讲核心的地方。

已知三角形三个顶点V1、V2、V3，如何在屏幕上渲染这个三角形？

我是这样干的，按道理显卡里面也是这么干的：



我自己随手画的图，绝对不侵权

先计算三个顶点在屏幕上的坐标那么，已知该三角形的屏幕高度是300个像素，那么写一个for循环：



是的，你现在需要的是，计算出来有几条线，然后一条一条的画线！

什么？这代码太复杂了，看不明白？大佬，你现在学的是计算机编程的三大浪漫之一，这点复杂度，才刚刚开始。如果这都啃不下，放弃吧，浪费你的时间那是我的罪过。

好了，一个简单的光栅化过程，已经讲完了。这里，我们做了这一轮，可以稍稍思考一下，明白一些道理：

1. 三角形越多，渲染效率越低。
2. 大的三角形，是否一定比小的三角形渲染更慢？不一定，要看在屏幕上的像素数。
3. 屏幕分辨率越高，渲染效率越低。所以玩游戏，开4K分辨率跟开2k分辨率，显卡的计算量不止翻翻那么简单。
4. 如果你有一定的图形学基础，那么，看懂了这个过程，能轻易的理解了VS到PS之间的数据，只能做线性插值，因为本身就是一条一条线的渲染的。明白了这个过程，才可能深入的理解到后续光照里面的像素光照是怎么回事。

好了，此处真的已经把图像和图形的基础都介绍了一遍。至少对于图像的储存，压缩，有了一定的概念，也对光栅化的原理有了一定的理解。这一章暂时还想不到其他什么需要介绍的，就先到这里吧。