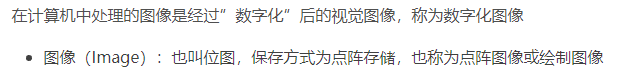
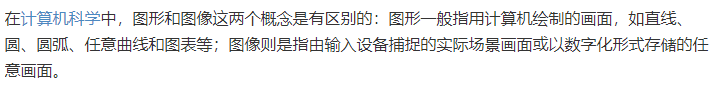
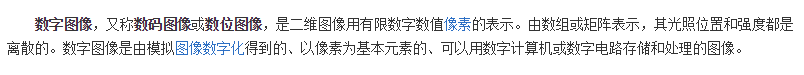
图像基础

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));

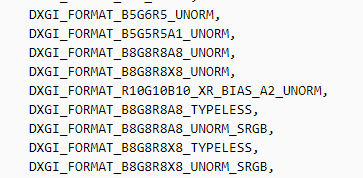
什么？这么简单的公式还复杂？那么我们分开两步：

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。

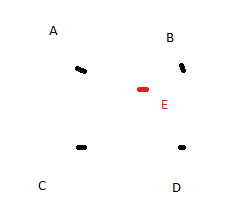
看到了吗，这就是一个最简单的采样。图片缩小了，但是找到同比例的地方，也就是u，v坐标的地方，把最近的点的颜色取出来即可。

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：

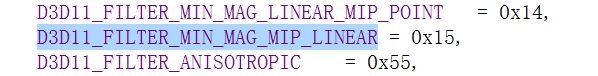
Col1 = A \* 0.25 + C \* (1 - 0.25);

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差不多的，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。

哈夫曼树。

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。