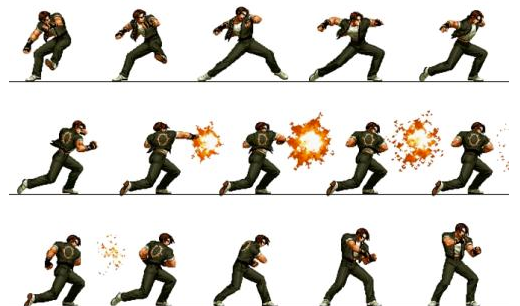
3D动画

3D动画，包含的内容非常多。各种各样的动画实现方式，我不大可能全部介绍。我只介绍一些常用的，我知道的。

先介绍最常见的序列帧动画。

这个常见到什么程度呢？所有的2D游戏，都是序列帧动画。所谓2.5D的，大部分也是序列帧动画。而3D的，也有很多用到序列帧动画。看图：



这个应该是KOF97里抠出来的？图片来自网络。

说穿了就是，N张图片的混合，这就叫做序列帧动画。例如一个2D游戏，一个人物角色，一个站立动作，八个方向，每个方向16帧。那么，一个人物的站立，就是8 \* 16 = 128张图片。

那么这些图片，是如何播放的呢？以一张个方向16张图片为例，假设播放速度为50ms一帧，每秒播放20帧：

Void Tick(float Delta) // 每帧调用函数

{

Static float TotalDelta = 0.0f;

TotalDelta += Delta;

If(TotalDelta >= 0.05f)

{

TotalDelta -= .0.5f;

// 播放下一帧图片

PlayNextPicture();

}

}

以上代码其实非常简单，只需要每帧检查一下是否满足播放下一帧的播放条件即可。一般来说，这种动画，有的是循环的，有的是不循环的。循环的意思是指：图片的最后一帧跟第一帧，是能够接在一起的。例如一个人物站立，播放完了，继续播放，肯定是必须循环的。就像上图那个草稚京，放那个荒咬技能，开始是站立，放完了还是站立，那么连续放的时候，不会觉得有什么不妥当的地方，这就是可以循环的。

一句话，这类动画的制作，要考虑衔接的问题。

这类动画的例子太多了，数不胜数。例如一些高档装备的光环，例如+10的装备，牛逼哄哄，光环加身，这类一般都是这种序列帧动画。

还有很多技能，都是序列帧。例如我一个火球放过去烧你，就可以做一个移动的序列帧，一边移动一边播放动画即可。

序列帧对于美术跟程序来说，都没有什么技术难点，但是，对于后台的技术来说，还是有一些技术要点的。

最典型的，就是序列帧怎么压缩的问题。我列举一些比较常见的压缩方法。

1. 大概在2000年后一点，网易做了大话西游。这个游戏，用的是云风的成名作《风魂》，这是一个2D引擎，图像处理部分使用了大量的汇编，主要是图片的处理，例如脏矩形处理等等。其中，就有这个序列帧的压缩应用。

这个算法我不确定我在之前有没有讲过，这里大概再讲一下。这个就是经典的调色板压缩。例如上图的草稚京，虽然有16张图片，你看他身体的颜色，都是差不多的，那个火焰的颜色，也差不多，所以全图其实颜色数是非常少的。我假设一个站立的图片有128个图片，放技能一个技能128个图片，一个人物我估计有一千多张图片（很多技能动作）。那么我首先确定，这一千多张图片的颜色数，不超过256。每个颜色的RGB，我可以用32位真彩色（上面的图不可能是32位真彩色，时代太早），带Alpha通道。

这样搞就简单了，我用索引记录每一个颜色数，索引用少一点，小于255即可。这样，我8位做索引就可以了。

这样的话，假设一张图片是128 \* 128的分辨率，那么就是128 \* 128个像素，每个像素1字节。128 \* 128 / 1024 = 16K。然后还可以做一个简单的RLE压缩。总的下来，可以大大的减少内存的占用。

这个压缩算法，当初可是风靡全国，一大堆游戏公司都用了风魂。我当时所在的深圳网域，也用了，开发了华夏1、华夏2、QQ华夏。

然而，这个压缩算法，用的人越来越少，现在甚至估计都没有什么人知道了，为什么？原因我估计有以下几点：

1. 、没有大公司支持。网易不是大公司？是也不是。能支持一种算法的，只能是一些底层的公司，例如MS，例如NV。他们主推的DDS格式，虽然小众，但是你搞windows 3D的必须要用。
2. 、算法本身有缺陷，例如针对的场景比较单一，例如用的汇编，众所周知，汇编的兼容性并不是那么好，需要不断的升级维护。并且随着指令集的提升，你不用SIMD都被淘汰了。
3. 、内存占用很大，并没有做分块，也就很难利用GPU的并行能力。也就没有硬件级别的采样支持。压缩只是压缩，终究是要解压的。
4. 、应用场景太少了，例如颜色超过256就坑逼了。
5. 、纯2D游戏越来越少了。就算是纯2D的游戏，也是3d做正交投影了，早期的direct2D都不用了，更加不要说其他。现在很多的2D游戏，都是3D渲染，2D贴图。

基于那么多的原因，这个算法慢慢被淘汰也就可以理解了。

1. PNG压缩。一般现在做游戏的，碰到这种，貌似都是用的PNG压缩？然后压缩个16位？我不确定有没有PNG16位的，但是PNG32位的肯定没问题的。当这类资源本身不是太多的时候，用PNG是比较合理的选择。
2. 使用DDS压缩。这个压缩，是MS跟NV力推的方案，支持3D图片，有硬件支持，显存占用少。显存占用少的意思是：当你用PNG之类的算法，无论你压缩到了多小，真正使用的时候，必须解压缩的。也就是说，一张PNG跟一张BMP，在显存里占用的空间是一样的。但是，DDS硬件支持，不需要解压缩，硬件可以直接采样！所以仔细看看各种大制作，清一色的DDS。然而，没有东西是完美的，这个格式有几个问题。第一个，熟悉DXT格式的都知道，这个是一个16位的颜色，颜色数少了一些。第二个，不能支持跨平台。不过我不知道移动端是不是有什么相似的方案。移动端我不熟悉。（附：我记得早期，大概十几年前，JPG也是不支持硬件直接解码的，当初我记得JPG也是需要解压到显存，导致显存占用非常大。但是现在，我记得是可以直接硬解了？因为JPG本身也是分块压缩，只是算法思路完全不一样。这块由于时间不断的变化，会导致一些写的内容可能出错，理解大概思路即可，细节的东西，需要自己去查现在的文档。我是懒得去查了，现在用不上。）

DXT格式，英文叫做Block Compression。简称BC1、BC2、BC3……BC7。MS官网上有专门的各种介绍，例如这个链接：

<https://docs.microsoft.com/en-us/windows/win32/direct3d10/d3d10-graphics-programming-guide-resources-block-compression>

这个链接下面各种链接，压缩的各种算法，文件如何保存……全部都有。我有认真看过的。有兴趣的也可以自己研究一下，还是有点意思的。

这个算法，核心部分，其实还是个调色板，英文叫Color table，也有翻译做颜色表的。所以说，虽然计算机图形学算法极多，但是很多算法真的是随处可见。这个调色板的操作，在vertex/index的套路里有，在图片压缩里有，连这里也有。我粗略讲一下这个过程。

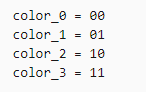
1. 第一步，先把文件分块，一块是4 \* 4。所以，你看到的dds格式，基本分辨率都是2的N次方。不是的时候怎么办呢？我记得应该就是强制弄成这样的，然后后面增加部分其实是浪费掉的。例如125给你弄到128，那个后续三个像素就是废的。也就是说，强制4像素对齐（这里的强制对齐除了DXT格式算法的原因，GPU里的图片，非DXT格式，也有这个对齐。当你mapped，然后memcpy的时候，必须要注意这个点，我就被坑过。而dds格式的图片，需要考虑到mipmap，基本上必须2的N次方，不是仅仅对齐一下那么简单，非要抬杠说我见过很多dds格式的非2的N次方的图片，那我也没办法）。
2. 现在，我们针对每个小块做压缩。先把24位、32位颜色，统统给你搞到16位。有的dxt格式带alpha，有的不带。为什么要这样呢？这不是压缩了精度吗？我认真思考过这个问题。我的答案是：首先，3D渲染的时候，是有光照的，光照的影响非常大，现在损失的这点精度，在光照计算之后，差异微乎其微。其次，DXT的压缩算法，用的是调色板+索引，还不是直接记录的调色板，是用像素差+插值（这个下面再讲），这么干，其实就影响很大了。如果用32位的颜色，这么搞一个插值，说不定差异更大。这个原因，可以在后续的算法讲解里能理解到。
3. 第三步，以BC1无Alpha为例。现在有16个像素，理论上最坏的情况下，有16种颜色。我不可能去记录16种颜色，这么搞，还压缩个屁啊。非压缩的时候，占用空间是16 \* 16 / 8 = 32字节，单个Block。压缩的时候怎么做呢，调色板就记录两个颜色，这个block里的最大值跟最小值！只记录了两个16位色，一个叫color\_0，一个叫color\_1。自然只占了16 \* 2 / 8 = 4个字节。然后，根据最大值跟最小值，线性插值，可以得到中间的两个颜色，大概是这样：



上图来自于那个链接。

现在得到了四个颜色。然后，16个像素吗，理论最大有16个颜色，现在硬生生给弄成了四个，其他的，自然都是给忽略掉了。

现在四个颜色做索引，就简单了，不就是四个索引吗？我用2位就能搞定。大概这样：



这个看不懂？那就是二进制不熟悉了，只能回头去恶补二进制了，没有其他法子。

1. 第四步，储存16个像素。16个像素，我现在变成了只需要记录16个2bit的索引，变成了16 \* 2 / 8 = 4字节。所以，我现在一个block，只需要8个字节的储存空间即可。牛逼啊，原本24位，16个像素，24 \* 16 / 8 = 48个字节的空间占用，硬生生给我压缩成了8字节。

其实，这是赤裸裸的有损压缩，并且损失还非常大。所以，这个格式一般也就是3D渲染的时候用一下。要知道，很多贴图，都是一些颜色块，效果靠光照，这种情况下，颜色块有小小偏差，也就无所谓了。另外，这个肯定还跟硬件采样有关。这个算法，很适用于显卡的平行运算，采样。所以，现实中满世界的JPG，3D世界满世界的DDS，只能说，合适的东西做合适的事。(JPG也是分块，但是算法上不大一样。)

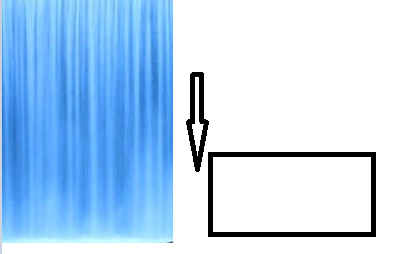
好了，序列帧没太多介绍的技术点，就介绍到这里。下面，介绍一下UV动画

UV动画：

UV动画主要用在哪些地方？例如，一个瀑布，水流不断往下，可以用一个简单的UV动画循环；一条河流，水面可以用一个简单的UV动画，做出好像泡沫顺流而下的效果；一把权杖，也可以用UV动画实现一种流光溢彩的感觉。

UV动画一般可以有U方向，可以V方向，可以UV方向……诸如此类，一般用shader实现就可以了。

大概这样解释一下UV动画的原理：



看图，你画右边这样一个面片，然后把左边的贴图贴上去，贴的时候，不能用1‘1这样贴，这样就全贴上去了，造成了压缩。然后在shader里面，把V坐标根据时间不断的放大，就能营造出一种瀑布不断流下的感觉了。如果贴图做成无缝拼接的，那么就可以了。

大概shader如下，做UV同时计算：

Float4 PS(Texture, UV, uniform float timeStamp)

{

UV = UV + UV \* timeStamp;

Float4 Col = Texture.sample(UV);

Return Col;

}

如果只是U动画，只需要U加入时间计算。如果是V动画，只需要V加入时间计算即可。

UV动画使用的时候不是那么多，稍稍了解一下即可。

下面讲一个也是非常常见的插值动画，英文里一般直接叫Animation，关键字：transform、translate、rotate、scale。

这个动画，我也不知道中文一般怎么叫。游戏里，一条鱼在水里游来游去，镜头在场景里面自动漫游，鸟在天上飞来飞去等等，基本都是这个动画。当然了，也有骨骼动画。但是这里先不讲到骨骼动画。

这类动画一般是怎么实现的？一般是记录多个关键点，然后做插值。大概这样：

Animation:

KeyFrame Position:x = 0, y = 0, z = 0;

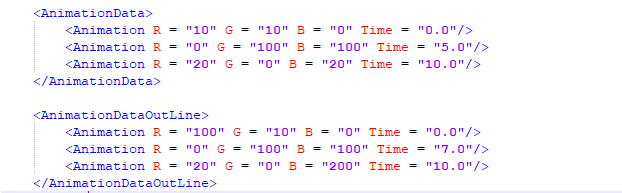
Rotate: x = 0, y = 0, z = 1, w = 1;

Scale: x = 1, y = 1, z = 1;

Time = 0.0;

每一帧，其实关键是记录了平移、旋转、缩放、时间。

OGRE里面，无论是ofusion还是OGREMAX导出插件，都是这么记录的。我自己写的一个DMX512灯光动画，也是这么写的，看图：



解释的时候怎么弄？其实很简单，做插值即可。代码大概这样：

线性插值：

Vector3 lerp(vector3 v1, vector3 v2, float time)

{

Return v1 \* time + v2 \* (1 - time);

}

B样条插值：

这个主要是多个顶点，做一个样条线插值。这个插值用得比较多，也比较平滑。一般的这类动画都是用的样条线插值而不是线性插值。毕竟线性插值太生硬了，不适合动画。

在3D里面，插值的方法很多，贝塞尔曲线，B样条线，各种变种至少十几种。一般最多的还是线性插值跟B样条线。好处是不言而喻的。贝塞尔曲线不能局部修改，只能全部重构，这是硬伤。这里，就不打算抄代码了，OGRE这样定义引擎里面都有B样条线插值，有兴趣的自己去看看就有。

方向的插值复杂一些，一般用四元数插值，slerp。大概这样：

Quaternion lerp(Quatnion q1, Quaternion q2, float time)

{

Return Quaternion::Slerp(q1, q2, time);

}

逻辑代码也简单，大概这样：

Void Tick(float Delta) // 每帧更新，传入两帧时间间隔

{

Static float CurrentTime = 0.0f;

CurrentTime += Delta;

// 现在要判断CurrentTime在哪一帧跟哪一帧之间

For (int i = 0; i < NumFrames - 1; i++)

{

If(mFrameInfo[i].time < CurrentTime && CurrentTime <= mFrameInfo[i+1].time)

{

// 刚好两帧之间，那么计算在两帧之间的比例

Float time =( CurrentTime - mFrameInfo[i].time) / (mFrameInfo[i+1].time - mFrameInfo[i].time);

// 然后插值即可

Lerp();

Lerp();

……

Break;

}

}

}

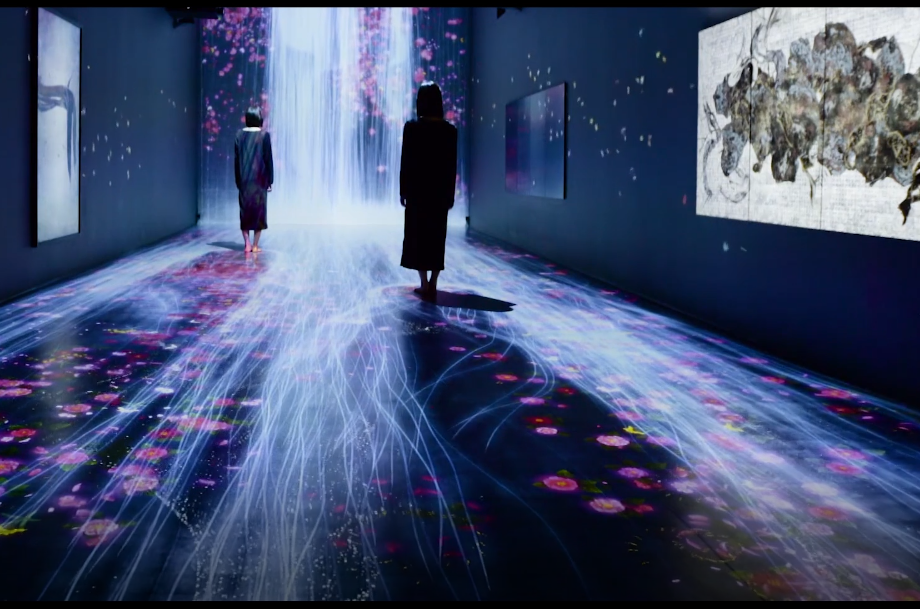
上面的代码，是比较简单并且无优化的代码，但是原理是正确的。正确的做法，不需要每帧全部重新遍历的，但是这是解释原理的代码，就不要加入优化了，加入优化很容易让新人一头雾水？

就如此简单？其实基本就如此简单。只是有不少的细节需要考虑到。例如时间播放完毕了，是否循环。例如一条鱼游来游去，起点需要跟终点重合，这样就不会穿帮。再例如动画会不会有什么触发条件导致开启或者关闭，编辑的时候怎么方便。诸如此类的细节问题。

但是，为什么UE里面的Matinee那么复杂呢？因为这不是一个简简单单的动画，而是一大堆动画的混合体。例如，你可能镜头移动到了一个峡谷，两边出来一大堆烟雾。例如你镜头动画的过程中，出现了镜头抖动效果。例如你镜头飞进一个山洞，画面渐隐……诸如此类的，全部支持下来，自然变成了一个庞然大物。

这里，我不打算详细讲这些实现，所有的庞然大物，其实都是由一个一个小的细节生成的。有兴趣的，可以自己看，或者留言问我。

关于这个动画，其实应用很多，例如那个知名的teamlab的瀑布，我们就用这个东东实现了一个差不多的。大概截图如下：



发这个图是想说，你所看到的各种惊艳效果，都是可以由一些小小的功能组成，打好基础才是王道。

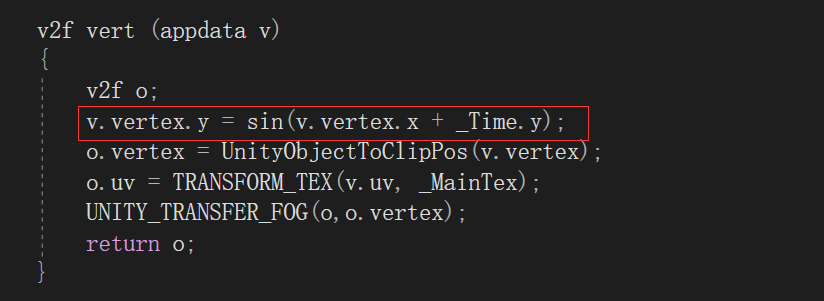
顶点动画。

顶点动画的应用非常多。例如最常见的水面波动，就是顶点动画。顶点动画的一般定义，是指在Vertex Shader里面实现的顶点变动。例如水面波动，小草的随风摆动……。

这里，为了简单起见，我用unity非常简单的写了一个顶点动画，作为例子。

步骤非常简单：

1. 打开unity，创建一个新的3d工程。
2. 创建一个Plane。
3. 为Plane创建新的材质，Shader。Shader用最简单的Unlit。
4. 把新建的材质，shader赋予给新的plane。
5. Shader新增一行代码，如图：



其实就是把这个顶点的y值，实时计算一下，根据时间，在x方向做一个sin()即可。这么一来，一个极其简单的正弦波浪就生成了。

这个，只是一个简单的描述这个顶点动画的例子。原理很简单：在顶点着色器中，计算Y轴的动画。我们常见的水面波动，都是这么来的。但是，不会是这么简单的做法。现在比较主流的做法，是Perline Noise或者FFT。这里我不打算详细解释，网上资料很多。

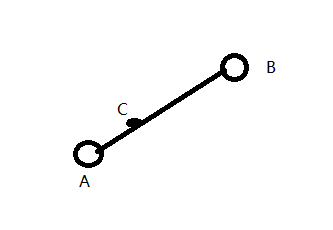
其实早期游戏的地形也是这种生成方法，只是不是直接计算生成，而是用的地形编辑器，人工生成，先生成一个512 \* 512的格子，然后人工不断的编辑，拉伸各个格子，编辑好地形，然后再摆放物体，生成遮挡等等。

顶点动画有一个问题：一个面片，如果顶点过于稀疏，中间的插值由于是线性插值，可能会导致看起来比较糟糕。解决方案，一般都是用LOD。例如地形和水面，一般用四叉树，近的地方顶点多，远的地方顶点少。其他的，例如用GS实时生成三角形的，用tessellation的，各显神通。

好了，接下来讲一讲3D动画里面，最复杂的，最不好理解的骨骼动画，英文一般叫做Skeleton/skeletal Animation。

我们先看一个最简单最简单的骨骼，然后先知道怎么软件实现，再考虑其他优化，例如硬件蒙皮，例如硬件蒙皮有哪些可能的优化方式等等。一旦连个最简单的骨骼动画软件实现都没有理解，谈什么硬件实现，什么优化？

先看看这个最简单的图：



假设A、B分别是骨骼上的一个点，C是需要渲染的Mesh上的一个点。专业术语上，一般A和B，会被记作Bone。那么，我现在先做一个简单的组织架构，我把B记作A的父节点，A是B的子节点。

这个时候，很容易理解：假设A动了，或者B动了，C一定动了。那么思考题：假设B动了，A动了吗？

当然动了啊，因为B是A的父节点！一棵树，哪里有树根动了，树叶没动的？

那么问题来了，为什么要搞成父节点、子节点这种组织架构？很简单，因为人物、动物骨骼，都是这样的。例如你可以尝试一下，你手臂动了，手指完全不动这种骚操作？大佬真的可以？我靠，我觉得你适合去杂技团，那才有大好前途，写什么代码？屈才了。

假设大佬也认识到父子节点才是骨骼动画的根本，那么，继续往下看。

现在，我需要描述A、B的移动、旋转、缩放。这里，可以用一个矩阵来描述，这个，是3d图形学的基础。如果连这个矩阵都不知道怎么算的话，回头看看基础吧，你的基础不适合往下看。

假设现在已知矩阵A，记作MA，矩阵B，记作MB，那么如何算C的坐标呢？

这里，有一个权重的概念。其实顶点坐标的格式里，一般有Vertex、Index、TextureCoord、Normal，还有一个，就是权重，一般叫做BlendWeight。这个等于，一个Mesh，假设带骨骼动画，那么，每一个顶点，增加一个权重的数据，用于作骨骼动画计算。常见的权重，是float2，也就是双权重。例如上图，C点的坐标，只跟A跟B有关，跟其他骨骼点无关。多权重什么的，也是有的，这里不作太多讨论，原理上应该是差不太多的，我也没搞过。

假设C点的双权重是0.7，0.3，那么，C点的坐标计算，需要两个步骤：

1. 分别计算得到A跟B的世界矩阵。由于B是父节点，A是子节点，那么A节点的矩阵，需要根据B节点跟自身的平移、旋转、缩放计算得到。
2. 假设已经得到B的世界矩阵WMB，A的世界矩阵WMA，那么，C点的计算就是：float3 C = WMB \* C\_Vertex \* 0.3 + WMA \* C\_Vertex \* 0.7;

C\_Vertex也就是C点本身的顶点坐标。如果不带骨骼动画，C\_Vertex的计算，实际上就是WorldMatrix \* C\_Vertex即可。

是不是很简单？骨骼动画就这点东西？瞬间觉得图形学也就太容易了吧？确实，这个看似简单，其实不那么简单。骨骼动画最大的问题，在于优化。如果你一个游戏，就那么一两个小小的骨骼动画在里面扭动一下，例如2d休闲游戏，那么，真的，挺简单的，怎么做都是对的，也没有任何的性能瓶颈。问题出在哪里？出在复杂的游戏场景需求！我简单总结一下，有这样一些不大好处理的问题：

1、多人物，多骨骼的优化问题。

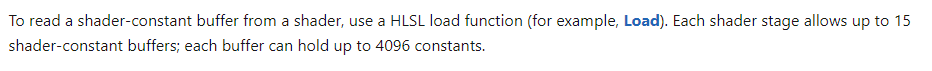
假设，一个人物有100个骨骼节点（之前的游戏一般没那么多，低于64。AAA大制作估计就不止了。）那么，上面的套路计算的话，就有100个矩阵。假设游戏里有100人同屏，每秒渲染60帧，这个计算量多大？那是：

60 \* 100 \* 100 = 600000次矩阵计算。更绝的是，每一个子节点的计算，都跟父节点有关，计算一个骨骼动画，就变成了先计算父节点，再计算这个父节点的子节点，一级一级的往下计算……。所以还不是60万次矩阵乘法那么简单。如果是软件蒙皮，更可怕了。假设一个人物模型有5000个顶点，不算太多吧，每个顶点还要来两次矩阵乘法，做一个混合。这么大的计算量，CPU根本顶不住。

怎么办？无非就是那么几个办法：

1. 、国产的游戏，基本上骨骼点少，降低骨骼量，例如用30个骨骼点。
2. 、使用硬件蒙皮，其实就是把插值计算部分，放进顶点着色器里面算，而不是在CPU里面算。这个，会引出其他一些问题。
3. 、不用矩阵计算，不考虑骨骼动画缩放，那么就变成了一个Translate，一个Rotate。一个向量，一个四元数即可，不用一个4 \* 4的矩阵。
4. 、降帧率，降同屏人数，国战游戏，同屏人数过多的时候，甚至推出例如娃娃模式，所有人物都是同一个娃娃模型，同一套少数的骨骼，退出模式再变回原来的人物（这个是某赚钱游戏的优化方式）。

这里，重点讲讲硬件蒙皮。即使使用了硬件蒙皮，而前面的矩阵计算，还是需要CPU来算的，这是矩阵一级一级的计算下去的，并不能并行，所有没有什么优化空间，顶多也就是SIMD指令级别的优化。硬件蒙皮的意思是：把这些矩阵传入Vertex Shader，然后在Vertex Shader里面计算实际顶点坐标。这个做法理论上没什么问题，问题在于早期的硬件，传入的参数是有限的（即使现在，也是有限的），例如SM2.0时代，或者openGL2.0时代，使用了硬件蒙皮的话，一般都是三几十个骨骼点，不能太多。具体数字，已经很老的东西了，我也懒得去查。现在到了DX11时代，这块其实就宽裕了很多，现在可以传入15个constant buffer，每个可以有4096字节。我算了一下，每个可以传入4096 / 64 = 64个4\*4矩阵。这个可以在MS官网上看到：



链接：<https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-resources-buffers-intro>

所以，早期无论是软件蒙皮，硬件蒙皮，都有各自的问题。但是总体来说，硬件蒙皮是必须的，因为这部分本身就是并行计算，硬件蒙皮能实现效率上的极大提升。估计现在已经没有什么引擎使用软件蒙皮了吧？

这里，我一直纠结，要不要抄一些代码来凑凑数。例如计算各个Bone的矩阵啦，例如顶点着色器如何做计算这个混合啦。想想还是算了，这类代码已经烂大街了，自己写又写不出彩，抄又太掉价。想看的，自己随便去搜搜就有了，就不抄了。

2、第二个问题，多个动画之间的切换。

例如，你一刀往前砍，砍到一半，别人刚好一刀把你砍死了，这个时候，你前一个动画没播放完成，现在又要播放死亡动画了，怎么破？

最简单的做法，我不做混合。例如我就等这一刀动画播放完了，再播放死亡动画。估计很多人不屑一顾：这不是坑爹吗，效果太差了。老实说，好好设计一下，动画不要那么长，跨度不要那么大，还是凑合的。早期的游戏，一大堆这样的，再怎么说，也比回合制游戏强吧？回合制不是更加离谱吗？完全不会有这类问题，然而不还是那么多人玩？还有2D游戏，全是序列帧，也没什么多问题，玩的人还贼多。国内的玩家不挑的：）。

复杂点的，做插值，做混合。怎么做？假设你现在劈砍动画，已经到了时间t，这个时候，你每个Bone，都是已经算出来一个矩阵的。这些矩阵，全部加入后续死亡动画的计算，增加一个混合权重。合理的做法，例如死亡动画2s，可以设定一个混合时间，例如混合0.5s，之后，劈砍动画的矩阵权重为0，也就是说，不再参与混合计算。

这么做就可以了吗？我认为这是不大合适的。因为这么做的话，其实就是一个线性插值。线性插值，最大的问题是会僵硬。以我自己的理解，我认为这里分解成Translate跟Rotate，用向量和四元数做样条曲线的做法会比较合理。（高能：这块我并没有实操过，早期我做游戏的时候，做的是没有混合的。至于现在，不做游戏好多年，更加没有做过了。所以一切事我自己的推敲，按道理没太大问题？这块可以轻喷。）

1. 第三个问题，一些其他复杂的问题。例如最典型的，使用双四元数而不是矩阵，来计算最终的vertex shader插值。问题出在哪里？当然还是因为线性插值的局限。一个很简单的小例子，就能说明问题：拿一根双汇火腿肠，双手抓住两头，一头左扭，另一头右扭，你会发现，中间立马塌陷了。这就好比骨骼动画，两个点，A和B，分别两个方向稍稍旋转一下，A和B插值的中间点，很容易就塌陷了。双四元数，也有叫对偶四元数，也有叫共轭四元数，英文叫Dual Quaternion。那么，Dual Quaternion是如何解决这个问题的？很简单，对偶四元数的性质，计算这个插值的时候，并非是直接的线性插值，而是路径插值。早前我讲过四元数的插值，一般是Slerp，插值就非常圆滑。

对偶四元数，本质上可以理解为：不使用矩阵计算，而是使用旋转Quaternion + 平移Vector3的方式，这种方式下，双权重插值，就不是普通的线性插值，而是两个四元数的Slerp，能有效的防止这种塌陷问题。然而，可想而知，缺陷也是存在的。缺陷就是忽略掉了矩阵的scale。意思是，假设骨骼动画存在缩放，那么，这个就不能用了。一般来说，存在缩放的骨骼动画很少。骨骼动画的来源，要么是动作捕捉仪，要么是美术K动画，动作捕捉仪是没有缩放的，美术K动画使用了缩放的话，貌似不大可控。所以，据我所知，一般的骨骼动画都是没有使用缩放的。

双四元数的算法，我这里就不贴了，贴了也是抄，网上一大堆。即使各个引擎，也是抄的算法，一般各大引擎里这部分都会贴出处链接。有兴趣的可以自己随便找个引擎就能看到。