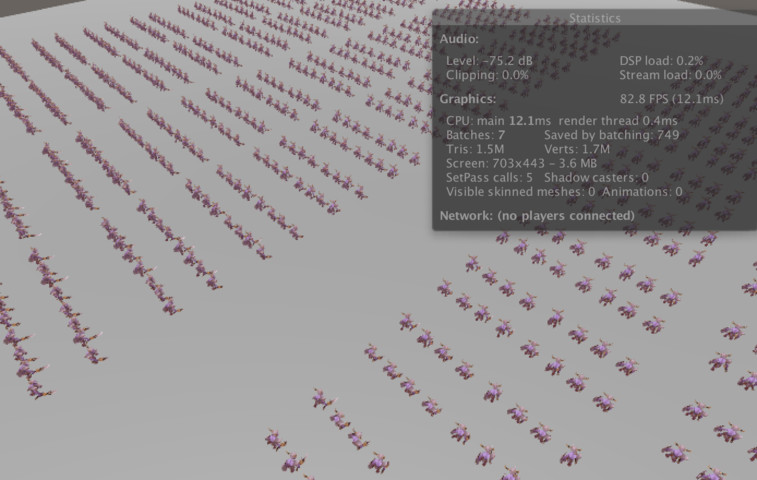
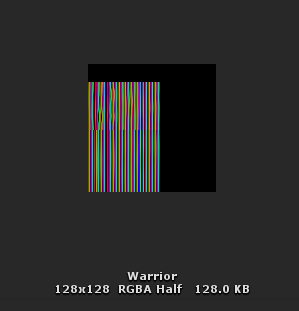
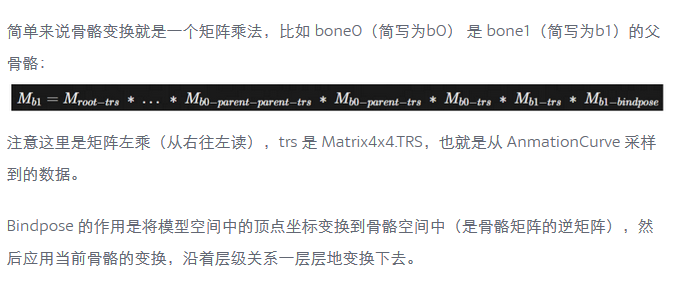
# **gpu instancing animation代替骨骼动画的做法**

作者：日音

链接：<http://gad.qq.com/article/detail/287613>

最早是在Unity推出gpuinstancing后，马上有人做了一个顶点动画代替骨骼动画的方案，当时自己也测试了一下，红米2一千人可以跑60帧，确实非常不错。后来发现UWA群里也有人在讨论这个东西的做法，当时M神说可以用烘焙骨骼的方式代替烘焙顶点，这样子烘焙出来的贴图大小只和骨骼数相关。而小米超神也说是通过烘焙顶点，不过为了减少烘焙文件的大小，使用了类似RGBM的方式存储数据。  
  
    我整合了主流的几种做法，做了一个插件。  
  
    首先展示结果：  
  
    
  
   场景中可见大概750个角色，batches只有7，去掉地面和天空盒，其实这么多人只有5个Batches.  
  
   贴图大小：  
  
   
  
  115帧的动画，4秒不到一点，128k，而且看到图中还有剩余，即使动画文件更大一些，依然可以用这张贴图放下。可能现在还看不出来它足够小，等后面和烘焙顶点的做法比较一下，就知道这样做的优势了。  
  
  让我们从头开始。  
  
  一切都必须是opengl 3.0以上。  
  
  unity自带的gpuinstancing可以很好的工作在静态物体上，例如草，树。但遗憾的是暂时还无法对骨骼动画使用这个特性。而我们游戏经常使用上百个小兵单位作战，如果可以让小兵使用这个特性，那么对于性能的提升无疑是很可观的。于是有人提出了将动画信息烘焙到贴图中，在shader里面根据贴图设置顶点位置，也就是我们的顶点动画。这样的话，模型就既可以像骨骼动画那样播放动作，又可以使用gpuinstancing合批了。做法也非常简单，可以参考：https://www.cnblogs.com/murongxiaopifu/p/7250772.html  
  
  本来这样就可以了，但实际使用过程中却发现了几个问题。  
  
  1.烘焙的贴图过大，因为为了存储浮点数，必须使用rgbahalf的格式，这个格式每个像素有64个字节，是真彩色的两倍。假设一个小兵有1000个顶点，那么1s的动作就需要1000\*64,也就是64000个字节，而正常情况下，我们小兵在2000个顶点左右，动画在5s以上，那么每个动画贴图大概就在2M以上，甚至有可能是4M。而我们有60多个兵种，这样一算竟然有240M。虽然小米超神使用了RGMB来减少每个像素的大小，但那也高达120M的动画贴图了。而我们知道，原始的骨骼动画数据其实只有几百k左右。  
  
  2.无法计算光照，因为法线始终保持T-pos形态，在shader里面改变顶点位置的时候，无法重新计算法线。为了能够使用正常的光照计算，必须将法线也一起烘焙。幸运的是法线都是单位向量，可以采用rgba存储，但也需要大概1M左右的空间。  
  
  3.没有动画之间的blend，为了实现blend，必须对两个动作的贴图进行采样，然后lerp。这样会导致shader里放两张4M的贴图，对手游来说还是不小的开销。  
  
   综上所述，我最终还是采纳了M神的建议，使用了烘焙骨骼信息的方案。  
  
   来看看原理，烘焙顶点很好理解，就是把位置的值存到贴图中。那么如何烘焙骨骼信息，然后得到顶点位置呢？首先我们要理解骨骼动画的原理，这里引用UWA博客里面的一段话：  
  
   
  
     当然上面的描述很简单，如果想要了解更加详细的推倒过程，可以看Milo大神的书《游戏引擎架构xxx》里面的蒙皮的数学这一章。  
  
     总之，结论就是从当前骨骼的bindpos一直左乘到根骨骼。  
  
     代码也非常简单：  
  
    

for (int j = 0; j < bones.Length; j++)

｛

GPUSkinningBone currentBone = bones[j];

Matrix4x4 lastMat = currentBone.bindpose;

while (true)

｛

if (currentBone.parentBoneIndex == -1)

｛

Matrix4x4 mat = Matrix4x4.TRS(currentBone.transform.localPosition, currentBone.transform.localRotation, currentBone.transform.localScale);

if(rootBone.transform != go.transform)

｛

mat = Matrix4x4.TRS(Vector3.zero, Quaternion.identity, go.transform.localScale) \* mat;

｝

lastMat = mat \* lastMat;

break;

｝

else

｛

Matrix4x4 mat = Matrix4x4.TRS(currentBone.transform.localPosition, currentBone.transform.localRotation, currentBone.transform.localScale);

lastMat = mat \* lastMat;

currentBone = bones[currentBone.parentBoneIndex];

｝

｝

animMap.SetPixel(j \* 3, k + 1, new Color(lastMat.m00, lastMat.m01, lastMat.m02, lastMat.m03));

animMap.SetPixel(j \* 3 + 1, k + 1, new Color(lastMat.m10, lastMat.m11, lastMat.m12, lastMat.m13));

animMap.SetPixel(j \* 3 + 2, k + 1, new Color(lastMat.m20, lastMat.m21, lastMat.m22, lastMat.m23));

if (k == startFrame)

｛

animMap.SetPixel(j \* 3, k, new Color(lastMat.m00, lastMat.m01, lastMat.m02, lastMat.m03));

animMap.SetPixel(j \* 3 + 1, k, new Color(lastMat.m10, lastMat.m11, lastMat.m12, lastMat.m13));

animMap.SetPixel(j \* 3 + 2, k, new Color(lastMat.m20, lastMat.m21, lastMat.m22, lastMat.m23));

｝

else if(k == curClipFrame1 + startFrame - 3)

｛

animMap.SetPixel(j \* 3, k + 2, new Color(lastMat.m00, lastMat.m01, lastMat.m02, lastMat.m03));

animMap.SetPixel(j \* 3 + 1, k + 2, new Color(lastMat.m10, lastMat.m11, lastMat.m12, lastMat.m13));

animMap.SetPixel(j \* 3 + 2, k + 2, new Color(lastMat.m20, lastMat.m21, lastMat.m22, lastMat.m23));

｝

｝

最重要的部分就是生成矩阵的那里。这里有几个注意点，一个是根骨骼可能有多个，那么你只需要将他们共同的父亲放到根节点，把这个其实没有骨骼的节点处理成默认矩阵的情况就可以。第二个是因为贴图采样有可能采样到边缘，为了防止精确度不够引起动画抖动，我前后各多增加了一帧，防止抖动。  
  
然后是shader部分：

v2f vert(appdata v)

｛

UNITY\_SETUP\_INSTANCE\_ID(v);

float start = UNITY\_ACCESS\_INSTANCED\_PROP(Props, \_AnimStart);

float end = UNITY\_ACCESS\_INSTANCED\_PROP(Props, \_AnimEnd);

float off = UNITY\_ACCESS\_INSTANCED\_PROP(Props, \_AnimOff);

float speed = UNITY\_ACCESS\_INSTANCED\_PROP(Props, \_Speed);

float \_AnimLen = (end - start);

float f = (off + \_Time.y \* speed) / \_AnimLen;

f = fmod(f, 1.0);

float animMap\_x1 = (v.uv2.x \* 3 + 0.5) \* \_AnimMap\_TexelSize.x;

float animMap\_x2 = (v.uv2.x \* 3 + 1.5) \* \_AnimMap\_TexelSize.x;

float animMap\_x3 = (v.uv2.x \* 3 + 2.5) \* \_AnimMap\_TexelSize.x;

float animMap\_y = (f \* \_AnimLen + start) / \_AnimAll;

float4 row0 = tex2Dlod(\_AnimMap, float4(animMap\_x1, animMap\_y, 0, 0));

float4 row1 = tex2Dlod(\_AnimMap, float4(animMap\_x2, animMap\_y, 0, 0));

float4 row2 = tex2Dlod(\_AnimMap, float4(animMap\_x3, animMap\_y, 0, 0));

float4 row3 = float4(0, 0, 0, 1);

float4x4 mat = float4x4(row0, row1, row2, row3);

float4 pos = mul(mat, v.vertex);

float3 normal = mul(mat, float4(v.normal, 0)).xyz;

v2f o;

UNITY\_TRANSFER\_INSTANCE\_ID(v, o);

o.uv = TRANSFORM\_TEX(v.uv, \_MainTex);

o.vertex = UnityObjectToClipPos(pos);

o.color = float4(0, 0, 0, 0);

o.worldNormal = UnityObjectToWorldNormal(normal);

float3 normalDir = normalize(mul(float4(normal, 0.0), unity\_WorldToObject).xyz);

float frezz = UNITY\_ACCESS\_INSTANCED\_PROP(Props, \_Frezz);

float3 normalWorld = o.worldNormal;

fixed dotProduct = dot(normalWorld, fixed3(0, 1, 0)) / 2;

dotProduct = max(0, dotProduct);

o.color = dotProduct.xxxx \* frezz;

return o;

｝

主要就是顶点着色器部分，我们把4x4的骨骼旋转偏移矩阵存在贴图里，因为最后一行是flaot4(0,0,0,1)，为了节省空间，我们只存了3x4大小的矩阵，最后一行在shader里补上。然后直接将矩阵和顶点相乘，就可以得到蒙皮后的顶点位置。而且我们看到，法线也可以这么处理，就可以得到蒙皮后正确的法线。这里还有一个我没有做的功能，就是骨骼权重，其实我将骨骼权重存进了顶点的uv2中，uv2.xy是第一根骨骼的索引和权重，uv2.zw是第二根骨骼的索引和权重，理论上需要将两个骨骼结算的结果加权平均一下，但因为我测试发现精度够了，就少采样一次，节省点消耗。如果有需要，可以自己加上这个加权平均。  
  
还有一个未来需要做的，就是动画之间的blend，需要额外增加一个变量控制blend的程度，对两个时刻的动作分别采样计算，然后lerp一下就可以了。  
  
我们看看用贴图存储骨骼需要的大小，假设一个小兵有25个骨骼，那么一个骨骼需要4x3个浮点数，也就是3个像素，那么需要75个像素，一个1s的动画，也只需要75\*64,大概4800字节而已。而且重要的是我们不受到顶点数的限制，而一个小兵的骨骼正常情况下就是30以内，我们得到了一个可控的合理的结果。  
  
最后献上商店地址：  
  
https://www.assetstore.unity3d.com/en/?stay#!/content/130516  
---------------------