VS + PS + GS

最近一直有人问我，我的书章节有没有顺序。我本人读书，很不喜欢按部就班，一点一点往下看这个套路，所以我也就没有规定章节。在我看来，除了教材按部就班以外，很多书，都应该是单独章节的，想看什么就是什么，想看材质就看材质，想看阴影就是阴影，而不是说必须先看什么，再到什么。

但是，不得不说，如果非要论顺序，本章节按道理是比较靠前的。原因无他，因为本章节，是大部分章节的基础。如果VS，PS都不熟，看阴影章节，按道理是看不下去的。但是，我写阴影章节的时候，预计会默认已经有了一定的图形学基础。

闲话少说，先来一波解释。VS，一般叫Vertex Shader，中文有个翻译叫做顶点着色器。PS，一般是指Pixel Shader，有时候也叫FS，有个中文翻译叫像素着色器。GS，也就是Geometry Shader，中文一般翻译做集合体着色器。

如果自己写过软光栅，其实比较好理解这些着色器的各个阶段。没写过的话，强烈推荐写一次。实在没有写过又想往下看，我勉为其难解释一下。

假设在CPU做一个软光栅，不考虑多线程软光栅的话，大概是这样：

For(int i = 0; i < 三角形数量; i++)

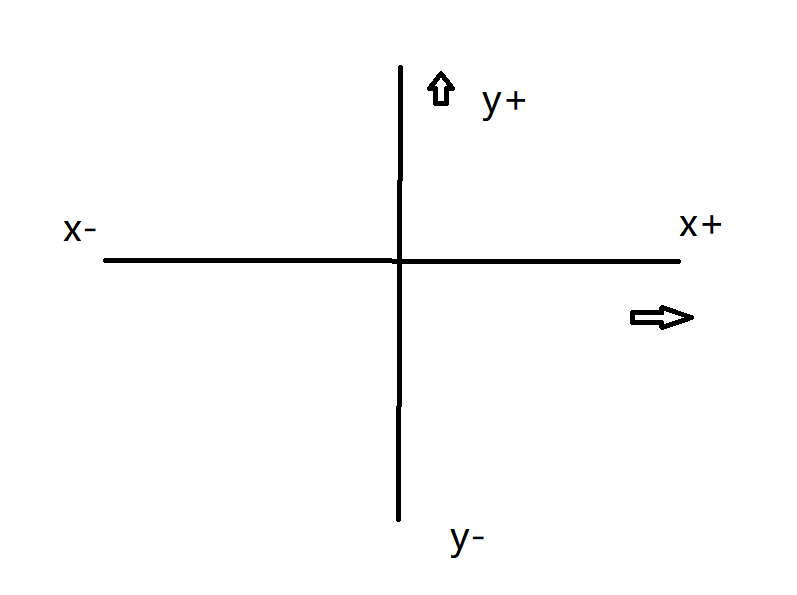
{

RenderTriangle(); // 渲染三角形

}

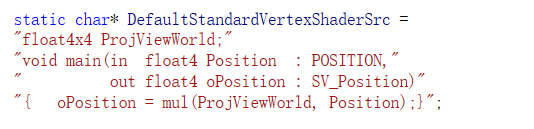
RenderTriangle这个函数大概是怎么弄呢？你渲染的时候，总得先判断这个三角形在不在屏幕上，在屏幕的坐标啊。这部分，叫做坐标变换，也就是VS的主要工作之一。

有一个矩阵，叫做WorldViewProjection，简称WVP。你用WVP \* Position，就能够得到屏幕坐标。当然了，实际上要稍稍复杂一点，因为还需要转换到屏幕坐标系。这部分内容，我记得以前应该讲过，这里还是简单复习一下，完全熟悉这块的略过。普通的坐标系，书本上叫做平面直角坐标系，大概是这样：



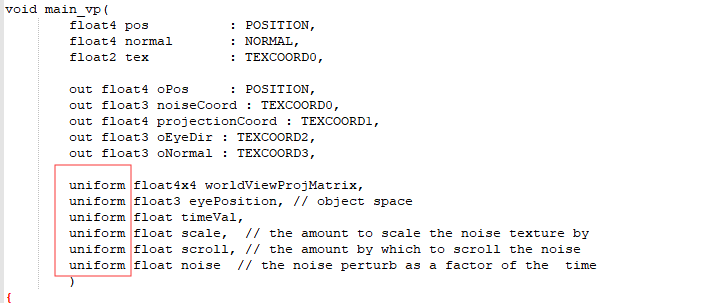
但是，屏幕坐标，你懂的，左上角是0，0，右下角是1920，1080。所以，这里有个坐标变换，而这个，是不需要你自己在VS里面做的，硬件就会帮你做了。但是，写软光栅，这部分需要自己做。本来想找找自己软光栅的代码贴一下，一下子找不到，算了。

所以，VS的核心功能，其实就是把三角形变换到屏幕上。其他都是次要功能。大多数简单的VS，大概率就是这样一行代码：



那么，这么简单的话，VS要来何用？当然了，很多时候，我们必须用到VS。例如，我们最常见的水面波动，就可以用VS来做。

怎么做？我开始折腾VS，就是练习自己写水面。最简单的做法，你可以传入一个时间，然后在VS里面花式sin一下。你可以sin一下position的高度，可以sin（t \* pos.y），也可以sin(t \* pos.length)，反正就是各种算，然后，你会慢慢摸索并且理解VS，尽管你做的效果很糟糕。

这里涉及到一个问题。无时无刻，你要切记：VS、PS、GS，都是执行在GPU中的程序，而引擎里的代码，基本上都是执行在CPU的，所以这里涉及到CPU跟GPU的同步，例如，这个时间，GPU是不知道的，而CPU可以轻易通过系统的API获得，例如典型的c函数time(NULL)，例如GetTickCount……。所以，这里需要CPU把时间传入到GPU。怎么传入？不同的shader语言其实有不同的做法，一般的书里都是介绍用Uniform变量，例如这种：

我一般不这样介绍，不同语法有不同的做法，例如典型的dx11里面，可以用const buffer。Const buffer好像最大是1024还是2\*1024，可以查一下MS的文档，我大概率不会记错。这个必须16字节对齐，然后自己memcpy到这个里面就可以了。不同的图形API，例如metal，GL，Vulkan等等，都可能是不同的，理解这个原理就好了，不同的API，查查文档即可。

所以切记，帧的概念。每一帧，CPU都需要更新每一个Mesh的传入GPU的变量。大概率，引擎里面，都会计算这个Mesh的WVP，然后传入到GPU。模型越多，这个效率自然也就越低。

这么简单的介绍，是不是有VS不外如是的感觉？可以有这种感觉，没有错。但是，当你非常纯熟，炉火纯青的时候，VS有很多优化，提升性能的妙用。

举个例子，现在有一大片草地，你有1000棵草，这些草随风而动。你可以这么干：大概计算一个风力，方向，然后计算这个草的偏移，大概可以这样：

For(int i = 0; i < 1000; i++)

{

//计算每一棵草的顶点偏移。

GrassYaw(i);

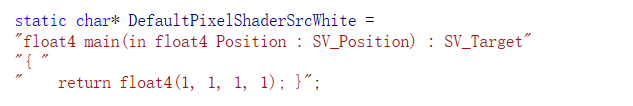
}

这其实是用CPU来算，我没记错的话，早期OGRE的Grass的DEMO，就是这么干的。但是显然，这么干效率低下，可以用GPU做优化。GPU优化听起来高大上，无非就是把这个计算，放在VS里面干。

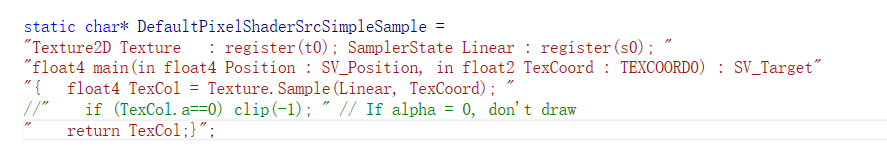
为什么GPU做这种事更快？强调一千遍，GPU是并行的，在CPU里，你需要循环1000次，GPU直接开1000个线程执行一次，哪个更快是显而易见的事。

所以，对于刚入门的人来说，VS就是一个简单的空间变换。但是，写得多了，什么东西该在VS里面，什么东西该在CPU里做，什么时候要上CS，轻易就能找到最优方案，这就是能力的不同。

回归正题，VS做了坐标变换之后，已知某个三角形要显示在某个像素上，那么，现在需要计算这个像素的颜色，这部分工作，其实就是PS来做的。这部分，在很原始的阶段，主要就是做一个像素采样。所谓像素采样，就是根据UV坐标，计算这个像素该显示图片上的什么颜色。这部分，我应该已经在之前的材质章节讲过。不打算再详细讲了。这里，主要讲讲PS能做些什么东西，一般用来做什么。



这个是最简单的PS，直接返回一个白色。无论你玩什么花样，传入什么参数，都没用，直接返回白色。

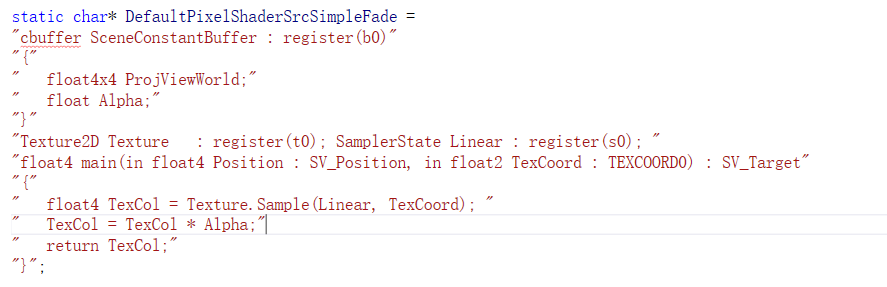


这个，就是最简单的纹理采样，根据UV坐标，采样纹理。有大佬说，怎么我看到的跟你这里的不一样？当然可能不一样，例如dx9的hlsl，大概率是这样一个函数：

tex2D();

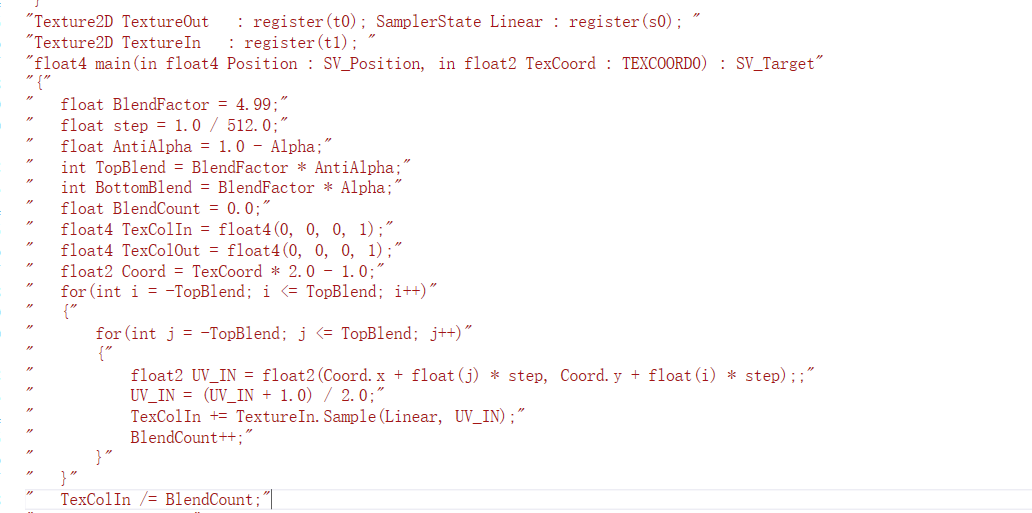
采样的函数有很多，不同的图形API，不同的版本，都可能不同，但是基本原理是一样的。这类函数一般有一个系列，例如texCube(), tex2DLod()，随便去搜一个hlsl说明文档，你能看到一大堆。这类函数有区别吗？还是有区别的，采样cube图，或者lod此类的，各有不同。我不打算在这里详细讲，需要的话，可以看文档。没记错的话，我在纹理章节里，应该有涉及到一些，例如就是纹理LOD的，其实就是纹理的mipmap，所谓的双线性采样，三线性采样，如何求偏导数，应该都有讲过。

那么，PS里能干什么？但凡跟屏幕颜色相关的，都可以做。举例：做一个简单的渐变效果，例如一个人物，在场景里面慢慢的显示出来或者消失，就可以通过一个alpha来做一个渐变，在PS里面实现。大概是这样即可：



你可以在CPU里面，把Alpha的值，在3秒的时间内，从0到1，或者从1到0，就可以实现这类渐变。你还可以这里玩各种花样，例如人物穿一身衣服，渐变切换到另外一身衣服，诸如此类的东西，都极其简单。多写几次就会了，没什么难度。

除了渐变，常见的，还可以做模糊。最简单的模糊怎么做？例如你要采样的坐标是UV，那么你采样的时候，把周边的像素都采样了，求个平均值，这就是最简单的模糊算法。会显示一个虚像，整个画面变得模糊。大概可以这样干：



看到了吗？其实就是采样了周边的像素，加权求平均，就是如此简单。后续讲阴影的时候，会有专门讲到，阴影的边缘是如何做模糊的，其实最简单的做法，也是这么干，就能实现阴影边缘的锯齿变模糊。

PS还有一个非常常见的应用。这个应用，有一个比较奇怪的名字，叫“后处理”。我不知道正规的书籍是不是这么翻译的，不同的引擎这个叫法不同，例如在OGRE里面，这个叫做Compositor。而在UE4里面，这个叫做Post Process Volume。主要做些什么呢？例如DOF（Depth Of View，景深），Blur（模糊），HDR……诸如此类的种种效果，都是用的所谓的“后处理”。说起来玄乎，其实很简单。先正常渲染，得到一张图片，然后对这张图片重新处理一下，实际上就是再在屏幕上画一个矩形，刚好满屏，就是（-1，1）之间即可，然后在PS里实现各种效果，例如可以调色，可以理解为photoshop里面的滤镜。很多奇奇怪怪的效果，都是这样实现的。例如游戏里，一刀砍过去，整个屏幕一阵抖动，看起来很玄乎，都是类似的应用。这个应用有一个大硬伤，就是不支持传统的AA（抗锯齿），例如MSAA。这里，扩展讲一下AA，AA其实可以单独开章节讲的，毕竟内容很多，方案很多，光是一个单独的TXAA就很多内容。不过好像资料已经很多了，抄书也没什么意思。传统FSAA的原理，其实就是放大渲染，例如你现在渲染1920 \* 1080的窗口，那么4倍抗锯齿，就是渲染一个4 \* 1920， 4 \* 1080的大图片。自己写过软光栅就知道，每个像素点都是需要计算光栅化的，像素点越多，渲染效率越低，这就是为什么现在4K，8K流行不起来的重要原因。所以说，AA非常的占资源。MSAA改进了一点，只把边缘部分放大渲染，然后再缩小。这部分，是在硬件里面实现的。也就是说，你只需要开启NV的选项即可，什么都不用干。这就带来一个问题，你渲染到纹理的时候，是没有抗锯齿的，所以，早期实现什么HDR之类的，都是牺牲了抗锯齿为代价。不过后来，有一些新式的抗锯齿方式，例如FXAA，TXAA，这类技术，是不需要像MSAA一样的。不过FXAA的效果，讲真，一言难尽。反正我很不喜欢，基本不用。效果比较模糊不说，在一些场景，例如森林，树叶比较多的场景，边缘检测实在是有点糟糕。最近两年AA技术有没有什么新东西我不知道，早几年，UE4只有两种AA可选，就是FXAA跟TXAA。主要是早几年延迟渲染大行其道，而延迟渲染其实也是MRT（multi render target）的后处理，其实就是一次不是渲染一张图片，而是渲染好几张图片，把颜色，坐标、法线等等渲染出来，统一再在PS里面处理一遍，跟后处理的区别只是渲染图片数的不同。

PS复杂的玩法还有很多，例如光照计算，例如所谓的PBR材质计算，例如阴影的计算……基本都是在PS里面完成。这里，我不打算一一介绍。我这里只打算科普一下PS是个什么，怎么用，而不是打算写一个《PS应用实战案例分析》。PS的实战，需要漫长的时间，一点一点的积累，一点一点的磨，才能有所收获。这也是图形学太过于枯燥跟难以掌握的原因之一。

GS，几何体着色器。

以上，整个渲染流程都已经有了，为什么还需要GS？GS能干什么？其实，在AI大火之前，Nvidia一直很艰难。市值一直只有80亿就是明证。知乎上说起来高工资，都是说FLAG，从来没听说AMD，NV入列。那个时候，nvidia基本上all in游戏，一直致力于解决游戏里碰到的，然后CPU不好解决的问题。我认为GS的出现，也是这个原因。

举例：假设游戏里，你需要漫天的雪花掉落，这是不是一个很常见的需求？

漫天的花瓣掉落，是不是一个很常见的需求？

早期，你需要这么干：有一个粒子发射器，然后计算发射了多少个粒子，然后每个粒子画一个矩形，贴上贴图。假设某帧创建了100个花瓣，你需要这样：

// 申请一块显存/内存。其实一般是写入内存，写好了再memcpy传入显存。

AllocVideoMemory();

For(int i = 0; i < 100; i++)

{

// 画Quad

CreateQuad();

// 一般花瓣不会只有一个材质，太单调，可以随机一下材质

RandTexture();

// 把计算结果cpy到内存，后面再统一到显存

CpyToMemory();

}

TransferToVideoMemory();

这里，就有优化空间了。例如这个计算Quad，本身是大量并行的重复的操作，你要算vertex，index，比较麻烦。这部分工作，就可以交给GPU来做，这就是GS的用途。

也可能GS有其他用途，反正我就用来做过粒子的优化，其他的我没用过。

一句话总结：GS就是在GPU里生成几何体。你可以把生成几何体，当作是一个函数，传入一些参数，得到几何体。那么，传入的参数，肯定是CPU传入。而计算部分，放到GPU，充分利用了GPU多线程的优势。

那么GS是怎么做的呢？我的做法是：只需要传入一个顶点，我直接在GS里面计算Quad，圆形等等。我的代码大概是这样的：

struct Quad

{

float4 Pos[6];

};

Quad BuildQuadFromPoint(float4 P)

{

Quad Q;

Q.Pos[0] = P;

Q.Pos[0] = Q.Pos[0] + float4(-0.2f, -0.2f, 0, 0);

Q.Pos[1] = P;

Q.Pos[1] = Q.Pos[1] + float4(-0.2f, 0.2f, 0, 0);

Q.Pos[2] = P;

Q.Pos[2] = Q.Pos[2] + float4(0.2f, 0.2f, 0, 0);

Q.Pos[3] = P;

Q.Pos[3] = Q.Pos[3] + float4(0.2f, 0.2f, 0, 0);

Q.Pos[4] = P;

Q.Pos[4] = Q.Pos[4] + float4(0.2f, -0.2f, 0, 0);

Q.Pos[5] = P;

Q.Pos[5] = Q.Pos[5] + float4(-0.2f, -0.2f, 0, 0);

return Q;

}

[maxvertexcount(18)]

void MainGS(triangle in VertexShaderOutput vertexData[3], inout TriangleStream<VertexShaderOutput> triStream)

{

for (int i = 0; i < 3; i++)

{

Quad Q = BuildQuadFromPoint(vertexData[i].Position);

VertexShaderOutput VSO[6];

for (int j = 0; j < 6; j++)

{

VSO[j].Position = Q.Pos[j];

}

triStream.Append(VSO[0]);

triStream.Append(VSO[1]);

triStream.Append(VSO[2]);

triStream.RestartStrip();

triStream.Append(VSO[3]);

triStream.Append(VSO[4]);

triStream.Append(VSO[5]);

triStream.RestartStrip();

}

}

GS里面有一些规则，例如需要定义输入的是什么数据，输出的是什么数据，诸如此类的东西。我这里不打算深入讲解了，我当初是去MS的官网上看的，但凡理解了原理，看这类规则，都是极其简单的事，我都是一遍过就能随便写了。

GS比较核心的语法，主要是两个，一个是输入参数，一个是输出参数。还有一点，是GS应该是不支持Vertex，Index的套路的，也就是说，只支持顶点直接构建三角形，所以上面我的代码，一个QUAD其实是计算了6个顶点，而不是4个。还有一个是RestartStrip()这个函数，好像一个三角形之后必须这么来一下。细节不大记得了，自己测试一下或者找找文档即可。

摸清楚这些规则的时候，可以很简单的测试一下。例如你传入三个顶点，或者传入一个三角形，每个顶点再画一个QUAD，一天的时间足够摸清楚这些规则。如果这点功夫都不愿意花，只希望伸手，放弃吧，搞图形学是没有前途的。

有一说一，GS其实已经在了淘汰的边缘。早期刚出GS的时候，还仅仅只有VS+PS。GS仅仅是作为这两个的补充，是管线的一小部分。但是，这个东西作用很受限，能做的东西也不算多。随着CS的推出，这东西不可避免的要淘汰了。相比于GS，CS更加的灵活。你能做的事情，我都能做，我能做的，很多你都不能做，大概这意思。例如，我可以用CS计算粒子的轨迹，实现完全体的GPU粒子；例如，最近大火的光线跟踪，其实也是底层的类CS实现。为什么没有直接淘汰？我猜测是为了兼容以前的使用了GS的老游戏之类的吧。

此外，在Tessellation中，也用到了GS，也许后面还会不会用？天知道。

以上所有代码，除了那个uniform的那个是抄的，其他全部是我手写，并且都是经过验证的，能用的。但是基本都是N年前的代码了，长时间没有测试过也没有跑过了，不保证是不是给改坏过了。但是大概率能跑。

江湖越老，人越懒，章节越写越短。不喜可喷。