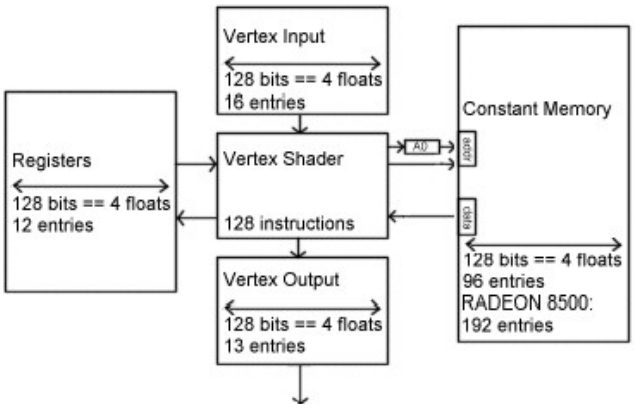
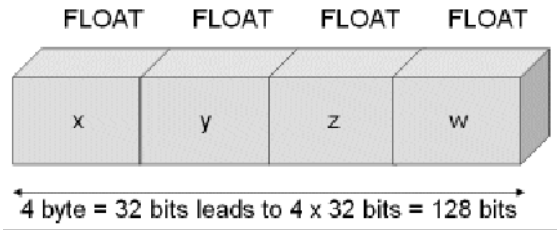
龙渊剑

Vertex Shader 结构

我们将通过下面这个Vertex Shader结构的图形模型来进入Vertex Shader的世界：



在Vertex Shader中的所有数据都被表现为128-bit的四元数(4 x 32-bit):



因为使用一个指令但处理一组数据，一个Vertex Shader的硬件可以被看成是一个典型的SMID（单指令多数据）处理器。Vertex Shader使用的这种数据格式非常的有用，因为大多数的转换和光线计算的进行都需要使用4 x 4的矩阵或者 四元组。

Vertex Shader指令非常的简单和容易理解。因为Vertex Shader不允许任何的循环，跳转和条件分支，这意味着它仅仅是线性的执行程序，一个指令接着一个指令。Vertex Shader的程序在Directx 8.1中最长为128个指令。我们可以结合几个Vertex Shader使用, 一个计算转换，一个计算光照。但是在同一时候仅仅只有一个Verthex Shader可以被激活，并且激活的Vertex Shader必须要计算每个顶点所有需要的输出数据。

一个Vertex shader使用16个输入寄存器（v0-v15，每一个寄存器都由128位的四元浮点数构成）来读取输入的数据。通过输入寄存器，Vertex shader可以非常容易的表示一个典型顶点的数据：位置坐标，法线，漫反射颜色和镜面反射颜色，雾坐标和贴图大小信息。

常量寄存器在vertex shader开始执行指定程序之前被CPU加载。常量寄存器是只读的， 一般用于储存例如光源位置、材质、特殊动画所需数据等参数。常量寄存器可以通过地址寄存器a0.x来间接寻址。常量寄存器除了在vertex shader中还可以在程序中被使用，但是在每一条指令中仅仅可以引用一个常量就寄存器。如果一条指令需要引用超过一个的常量寄存器，它只能通过暂存寄存器来引用。一般的常量寄存器为c0-c95，但在ATI RADEOM 8500中是c0-c191。

暂存寄存器由12个寄存器组成，是可读写的，可以用于数据的存储和读取。它们分别是r0-r11。

根据具体的硬件的不同，有至少13个输出寄存器。每个输出寄存器都以o打头。输出寄存器在光栅化时可以被使用。存在输出寄存器中的最终结果是另外的一个顶点，一个转换入"同源剪裁空间"的顶点。下面的表中列出了所有可用的寄存器:

Registers:	Number of Registers	Properties
Input (v0 - v15)	16	RO1
Output (o*)	GeForce 3/4TI: 9; RADEON 8500: 11	WO
Constants (c0 - c95)	vs.1.1 Specification: 96; RADEON 8500: 192	RO1
Temporary (r0 - r11)	12	R1W3
Address (a0.x)	1 (vs.1.1 and higher)	WO (W: only with mov)

Vertex Shader编程概览

因为在同一个时候仅仅有一个Vertex shader可以被激活，为每一个基本的功能块编写一个vertex shader是一个不错的主意。一般来说在不同的vertex shader之间切换的性能消耗要比变换一个贴图的性能消耗都要小。所以如果一个物体需要一种特殊的转换或者灯光，最好就在它的任务中给它一个恰当的vertex shader。让我们看看下面的例子：

你在一个外星球遇难了，身上穿着正规军的盔甲，但仅仅装备着一个锯子。当你在一个烛光照耀着的地下室穿行时，一个怪物出现了，然后你就躲到了一个在任何星球都很常见的箱子后面。在考虑你作为一个使用锯子拯救这个世界的英雄的命运同时，我们开始计算这个场景所需要的vertex shader数目。

首先需要一个vertex shader作为怪物的动画需要，光照渲染和可能存在的环境反射渲染。其他的vertex shader将分配给地板，墙，箱子，视角，烛光和你的锯子。或许地板，墙，箱子和锯子可以使用同一个shader，但是烛光和视角必须要有不同的shader。这依赖于你的设计和特定图形硬件的性能。

每一个vertex shader驱动的程序都必须要有下面的几个步骤：

- 通过检查D*3DCAPS8::VertexShaderVersion*来确定 vertex shader有无被支持。
- 使用D3DVSD_*宏来定义 vertex shader，使 vertex shader 的流映射到输入寄存器
- 使用*SetVertexShaderConstant()* 来设定vertex shader常量寄存器
- 使用*D3DXAssembleShader*()* 编译刚才所写的vertex shader（或者，你可以预编译vertex shader 使用一个shader编译器）
- 使用*CreateVertexShader()* 创建一个vertex shader句柄
- 使用*SetVertexShader()*将vertex shader与一个特定的物体相连
- 使用*DeleteVertexShader()* 删除vertex shader

检查Vertex Shader的支持状况

检查如果缺少一些特殊功能的支持，程序应该使用默认的行为(例如使用T&L)或者给用户一个提示，使用户做一些使这些特殊功能得以支持的事(就是提示他们可以购买新的显卡了^_^）。下面的代码段检查vertex shader 1.1是否被支持了：

```
if( pCaps->VertexShaderVersion < D3DVS_VERSION(1,1) )
  return E_FAIL;
```

在程序启动阶段，必须通过GetDeviceCaps()函数来得到一个D3DCCAPS8的结构caps。如果你使用Directx 8.1 SDK中提供的Directx框架来搭建你的应用程序，这个会被自动完成。如果检查后发现你的硬件不支持你需要的vertex shader版本，你必须通过设置*D3DCREATE_SOFTWARE_VERTEXPROCESSING* 属性调用CreateDevice()来切换使用软件vertex shader。这时将由对Intel和AMD不同CPU进行优化过的软件接口来进行vertex shader执行。

下面是不同版本Directx支持的vertex shader的版本：

Version:	Functionality:
0.0	DirectX 7
1.0	DirectX 8 without address register A0
1.1	DirectX 8 and DirectX 8.1 with one address register A0
2.0	DirectX 9

在1.0和1.1之间的唯一区别就是对于a0寄存器的支持。Directx 8.0和Directx 8.1对应的光栅化器(rasterizer)和Inter、AMD为他们各自CPU所写的软件模拟接口都支持1.1版本。在本文写成之时，市面上支持1.1的硬件只有GeForce3/4TI、RADEON 8500，同时要注意的是并没有只支持1.0版本的显卡，支持1.0的也肯定支持1.1，1.0只是一个过渡版本。

Vertex Shader定义

你必须在使用Vertex Shader前定义它。它的定义可以通过一个静态的外部接口来完成。看起来有可能是这个样子：

```
float c[4] = {0.0f,0.5f,1.0f,2.0f};
DWORD dwDecl0[] = {
  D3DVSD_STREAM(0),
  D3DVSD_REG(0, D3DVSDT_FLOAT3 ),     // 输入寄存器 v0
  D3DVSD_REG(5, D3DVSDT_D3DCOLOR ),   // 输入寄存器 v5
                                      // 设置几个常量寄存器
  D3DVSD_CONST(0,1),*(DWORD*)&c[0],*(DWORD*)&c[1],*(DWORD*)&c[2],*(DWORD*)&c[3],
```

```
    D3DVSD_END()
};
```

上面的Vertex shader定义使用*D3DVSD_STREAM(0)*来设置它成为0号数据流。在以后*SetStreamSource()* 将会通过这个声明绑定一个顶点buffer到设备数据流。你可以通过这种方法提供Direct3D渲染引擎不同的数据流。

> *举个例子，我们可以用第一个数据流表示位置和法线，第二个数据流来表示颜色和贴图坐标。它也可以使在单材质渲染和多材质渲染之间的切换变得非常容易：只要使有第二套材质坐标的的数据流失效就可以了。*

对于哪一个顶点属性或者输入的顶点数据被映射到哪一个输入寄存器，你也必须给出定义。*D3DVSD_REG* 将一个顶点寄存器和一个顶点数据流中的顶点元素(或者属性)加以绑定。在我们上面的例子中, *D3DVSDT_FLOAT3* 将被放入第一个输入寄存器中而*D3DVSDT_D3DCOLOR* 将被放入到第6个输入寄存器中。举个另外的例子，通过*D3DVSD_REG(0, D3DVSDT_FLOAT3)* 定义关于位置的数据可以被0号输入寄存器(v0)处理，而通过 *D3DVSD_REG(3, D3DVSDT_FLOAT3)*的定义，法线数据可以被3号输入寄存器(v3)处理。

如果一个人想使用N-Patches，开发者如何将输入的顶点属性映射入不同的寄存器比较的重要，因为N-Patch的斑格纹需要它的位置数据放在v0而法线数据放在v3。否则，开发者可以自由的映射到自己看上去适合的寄存器。例如，通过*D3DVSD_REG(0, D3DVSDT_FLOAT3)* 定义使0号输入寄存器(v0)处理关于位置的数据，而通过 *D3DVSD_REG(3, D3DVSDT_FLOAT3)*的定义，使3号输入寄存器(v3)处理法线数据。

> 与此形成对比的是在fixed-function渲染管道中，映射入不同寄存器的数据是固定的。d3d8types.h中有一张关于fixed-function管道渲染输入的数据的预定义。特定的顶点元素例如位置必须被放置在位于顶点输入内存中的特定寄存器。例如，顶点的位置被 D3DVSDE_POSITION 限定放于0号寄存器，漫反射光颜色被 D3DVSDE_DIFFUSE 限定放于3号寄存器。下面是d3d8types.h中的整张列表:

```
#define  D3DVSDE_POSITION      0
#define  D3DVSDE_BLENDWEIGHT   1
#define  D3DVSDE_BLENDINDICES  2
#define  D3DVSDE_NORMAL        3
#define  D3DVSDE_PSIZE         4
#define  D3DVSDE_DIFFUSE       5
#define  D3DVSDE_SPECULAR      6
#define  D3DVSDE_TEXCOORD0     7
#define  D3DVSDE_TEXCOORD1     8
#define  D3DVSDE_TEXCOORD2     9
#define  D3DVSDE_TEXCOORD3     10
#define  D3DVSDE_TEXCOORD4     11
#define  D3DVSDE_TEXCOORD5     12
#define  D3DVSDE_TEXCOORD6     13
#define  D3DVSDE_TEXCOORD7     14
#define  D3DVSDE_POSITION2     15
#define  D3DVSDE_NORMAL2       16
```

D3DVSD_REG 中的第二个参数表示了纬度和算法数据类型。下面的是定义在d3d8types.h中的值:

```
// bit declarations for _Type fields
#define D3DVSDT_FLOAT1 0x00 // 1D float expanded to (value, 0., 0., 1.)
#define D3DVSDT_FLOAT2 0x01 // 2D float expanded to (value, value, 0., 1.)
#define D3DVSDT_FLOAT3 0x02 // 3D float expanded to (value, value, value, 1.)
#define D3DVSDT_FLOAT4 0x03 // 4D float

// 4D packed unsigned bytes mapped to 0. to 1. range
// Input is in D3DCOLOR format (ARGB) expanded to (R, G, B, A)
#define D3DVSDT_D3DCOLOR 0x04


#define D3DVSDT_UBYTE4 0x05 // 4D unsigned byte
// 2D signed short expanded to (value, value, 0., 1.)
#define D3DVSDT_SHORT2 0x06
#define D3DVSDT_SHORT4 0x07 // 4D signed short
```

注意，GeForce3/4TI 并不支持D3DVSDT_UBYTE4，它在 D3DVTXPCAPS_NO_VSDT_UBYTE4属性中表示出来。

D3DVSD_CONST 将常量加载进vertex shader常量内存。它的第一个参数是填充有常量数据的数组的起始地址。数值的范围是0到95，如果是RADEON 8500，数值范围是0到191。在这里，我们从0开始。第二个参数指的是加载的常量向量（四元浮点数）的数量。一个向量是128bit，所以我们一次加载4个32bit的浮点数。如果你想加载一个4x4的矩阵，你可以用下面的代码段，加载4个128bit的四元浮点数到c0dd到c3寄存器:

```
float c[16] = (0.0f, 0.5f, 1.0f, 2.0f,
               0.0f, 0.5f, 1.0f, 2.0f,
               0.0f, 0.5f, 1.0f, 2.0f,
               0.0f, 0.5f, 1.0f, 2.0f);
D3DVSD_CONST(0, 4), *(DWORD*)&c[0],*(DWORD*)&c[1],*(DWORD*)&c[2],*(DWORD*)&c[3],
```

```
                    *(DWORD*)&c[4],*(DWORD*)&c[5],*(DWORD*)&c[6],*(DWORD*)&c[7],
                    *(DWORD*)&c[8],*(DWORD*)&c[9],*(DWORD*)&c[10],*(DWORD*)&c[11],
                    *(DWORD*)&c[12],*(DWORD*)&c[13],*(DWORD*)&c[14],*(DWORD*)&c[15],
```

D3DVSD_END 产生一个结束的标志表示vertex shader定义的结束。下面给出vertex shader的另一个定义:

```
float   c[4] = {0.0f,0.5f,1.0f,2.0f};
DWORD dwDecl[] = {
  D3DVSD_STREAM(0),
  D3DVSD_REG(0, D3DVSDT_FLOAT3 ), //input register v0
  D3DVSD_REG(3, D3DVSDT_FLOAT3 ), // input register v3
  D3DVSD_REG(5, D3DVSDT_D3DCOLOR ), // input register v5
  D3DVSD_REG(7, D3DVSDT_FLOAT2 ), // input register v7
  D3DVSD_CONST(0,1),*(DWORD*)&c[0],*(DWORD*)&c[1],*(DWORD*)&c[2],*(DWORD*)&c[3],
  D3DVSD_END()
};
```

在上面的例子中，*D3DVSD_STREAM(0)*设置它为0号数据流。位置值被放入v0，法线值放入v3，漫反射光颜色放入v5，一个材质坐标被放入v7。常量寄存器c0放置了一个128bit的值。

编写和编译Vertex Shader

在我们可以编译一个vertex shader之前，我们必须要写一个vertex shader...（古老的智慧 :-))。我首先将给你们一个关于指令的大概了解，然后再在下一章中介绍关于vertex shader编程的更多细节。

每条指令的语法结构是

操作名目标，[-]源1 [,[-]源2 [,[-] 源3]] ;注释

例如:

```
mov r1, r2

mad r1, r2, -r3, r4 ;r3的内容取反

 OpName dest, [-]s1 [,[-]s2 [,[-]s3]] ;comment
 e.g.
 mov r1, r2
 mad r1, r2, -r3, r4 ; contents of r3 are negated
```

这里共有17种不同的指令:

指令	参数	用途
add	dest, src1, src2	将src1加到src1(可通过在src2前加-,选择为减)
dp3	dest, src1, src2	三维点的乘积 dest.x = dest.y = dest.z = dest.w = (src1.x * src2.x) + (src1.y * src2.y) + (src1.z * src2.z)
dp4	dest, src1, src2	四维点的乘积 dest.w = (src1.x * src2.x) + (src1.y * src2.y) + (src1.z * src2.z) + (src1.w * src2.w); dest.x = dest.y = dest.z = unused dp4和mul的区别是什么呢？dp4产生一个标量的结果，而mul是一个分量乘分量的向量乘积。
dst	dest, src1, src2	Dst指令将这样工作:第一个源操作数(src1)将被看作是这样一个向量(忽略,d*d,d*d,忽略)，第二个源操作数(src2)将被看作向量(忽略,1/d,忽略,1/d). Calculate distance vector: 计算结果向量: dest.x = 1; dest.y = src1.y * src2.y dest.z = src1.z dest.w = src2.w

		在计算标准衰减的时候dst非常的游泳。下面是计算一个点光源的衰减的代码段： ; r7.w = distance * distance = (x*x) + (y*y) + (z*z) dp3 r7.w, VECTOR_VERTEXTOLIGHT, VECTOR_VERTEXTOLIGHT ; VECTOR_VERTEXTOLIGHT.w = 1/sqrt(r7.w) ; = 1/||V|| = 1/distance rsq VECTOR_VERTEXTOLIGHT.w, r7.w ... ; Get the attenuation ; d = distance ; Parameters for dst: ; src1 = (ignored, d * d, d * d, ignored) ; src2 = (ignored, 1/d, ignored, 1/d) ; ; r7.w = d * d ; VECTOR_VERTEXTOLIGHT.w = 1/d dst r7, r7.wwww, VECTOR_VERTEXTOLIGHT.wwww ; dest.x = 1 ; dest.y = src0.y * src1.y ; dest.z = src0.z ; dest.w = src1.w ; r7(1, d * d * 1 / d, d * d, 1/d) ; c[LIGHT_ATTENUATION].x = a0 ; c[LIGHT_ATTENUATION].y = a1 ; c[LIGHT_ATTENUATION].z = a2 ; (a0 + a1*d + a2* (d * d)) dp3 r7.w, r7, c[LIGHT_ATTENUATION] ; 1 / (a0 + a1*d + a2* (d * d)) rcp ATTENUATION.w, r7.w ... ; Scale the light factors by the attenuation mul r6, r5, ATTENUATION.w

expp	dest, src.w	E10位精度指数：
		
		---------- 	
		float w = src.w;
		float v = (float)floor(src.w);
		
		dest.x = (float)pow(2, v);
		dest.y = w - v;
		
		// Reduced precision exponent
		float tmp = (float)pow(2, w);
		DWORD tmpd = *(DWORD*)&tmp & 0xffffff00;
		
		dest.z = *(float*)&tmpd;
		dest.w = 1;
		---------- 	
		Shortcut:
		
		dest.x = 2 **(int) src.w
		dest.y = mantissa(src.w)
		dest.z = expp(src.w)
		dest.w = 1.0

lit	dest, src	从点的乘积和一个幂中计算光的系数 ---------- 为了计算光的系数，如下设置寄存器： src.x=N*L ;法线和光方向的乘积 src.y=N*H ;法线和不完全向量的乘积 src.z=忽略 ; src.w=镜面反射的幂 ;它的值必须在28.0和128.0之间 ---------- usage: 使用方法：

		dp3 r0.x, rn, c[LIGHT_POSITION]
		dp3 r0.y, rn, c[LIGHT_HALF_ANGLE]
		mov r0.w, c[SPECULAR_POWER]
		lit r0, r0
		---------- 	
		dest.x = 1.0;
		dest.y = max (src.x, 0.0, 0.0);
		dest.z= 0.0;
		if (src.x > 0.0 && src.w == 0.0)
		  dest.z = 1.0;
		else if (src.x > 0.0 && src.y > 0.0)
		  dest.z = (src.y)^src.w
		dest.w = 1.0;

logp	dest,
	src.w
		10位精度对数log2(x)
		---------- 	
		float v = ABSF(src.w);
		if (v != 0)
		{
		  int p = (int)(*(DWORD*)&v >> 23) - 127;
		  dest.x = (float)p; // exponent
		
		  p = (*(DWORD*)&v & 0x7FFFFF) | 0x3f800000;
		  dest.y = *(float*)&p; // mantissa;
		
		  float tmp = (float)(log(v)/log(2));
		  DWORD tmpd = *(DWORD*)&tmp & 0xffffff00;
		  dest.z = *(float*)&tmpd;
		
		  dest.w = 1;
		}
		else
		{
		  dest.x = MINUS_MAX();
		  dest.y = 1.0f;
		  dest.z = MINUS_MAX();
		  dest.w = 1.0f;
		}
		---------- 	
		Sortcut:
		dest.x = exponent((int)src.w)
		dest.y = mantissa(src.w)
		dest.z = log2(src.w)
		dest.w = 1.0

mad	dest, src1, src2, src3	dest = (src1 * src2) + src3
max	dest, src1, src2	dest = (src1 >= src2)?src1:src2
min	dest, src1, src2	dest = (src1 < src2)?src1:src2
mov	dest, src	move 优化提示:在每一次使用mov前都自己问一下，是否必须使用，因为这里经常会有直接通过源寄存器和需要输出的输出寄存器执行希望操作的方法。
mul	dest, src1, src2	dest是src1和src2的乘积 ; To calculate the Cross Product (r5 = r7 X r8), ; r0 used as a temp mul r0,-r7.zxyw,r8.yzxw mad r5,-r7.yzxw,r8.zxyw,-r0
nop		do nothing 什么也不做

rcp	dest,
	src.w
		取倒数
		
		if(src.w == 1.0f)
		{
		  dest.x = dest.y = dest.z = dest.w = 1.0f;
		}

		else if(src.w == 0)
		{
		  dest.x = dest.y = dest.z = dest.w = PLUS_INFINITY();
		}
		else
		{
		  dest.x = dest.y = dest.z = m_dest.w = 1.0f/src.w;
		}
		
		Division:
		; scalar r0.x = r1.x/r2.x
		RCP r0.x, r2.x
		MUL r0.x, r1.x, r0.x

rsq	dest, src	src平方根的倒数(比平方根有用的多)
		
		float v = ABSF(src.w);
		if(v == 1.0f)
		{
		  dest.x = dest.y = dest.z = dest.w = 1.0f;
		}
		else if(v == 0)
		{
		  dest.x = dest.y = dest.z = dest.w = PLUS_INFINITY();
		}
		else
		{
		  v = (float)(1.0f / sqrt(v));
		  dest.x = dest.y = dest.z = dest.w = v;
		}
		
		Square root:
		; scalar r0.x = sqrt(r1.x)
		RSQ r0.x, r1.x
		MUL r0.x, r0.x, r1.x

sge	dest, src1, src2	dest = (src1 >=src2) ? 1 : 0 用于模拟条件判断非常有用: ; compute r0 = (r1 >= r2) ? r3 : r4 ; one if (r1 >= r2) holds, zero otherwise SGE r0, r1, r2 ADD r1, r3, -r4 ; r0 = r0*(r3-r4) + r4 = r0*r3 + (1-r0)*r4 ; effectively, LERP between extremes of r3 and r4 MAD r0, r0, r1, r4
slt	dest, src1, src2	dest = (src1 < src2) ? 1 : 0

你可以从http://www.shaderx.com/下载这个列表的Word文件。如果想得到更多的信息，请参看SDK文档。

Vertex Shader运算器是一个处理四元浮点数的多线程处理器。它有两个功能模块 .SIMD(单指令多数据,Single Instruction Multi Data)向量模块对应着mov,mul,add,mad,dp3,dp4,dst,min,max,slt,sge指令。还有一个是特殊功能模块，对应着rcp,rsq,log,exp和lit指令。大部分指令的执行都只要一个周期，rcp和rsp在特殊情况下需要多于一个周期的时间。他们仅仅使用一条总线，这就使得当需要立即使用结果时，指令需要多于一个的周期，因为有一个寄存器延迟。

程序提示

Rsq主要用于正则化将用于光照等式中的向量。指数指令expp可以用于雾效果，噪声生成(参看NVIDIA Perlin Noise例子)，在一个粒子系统中的粒子行为(参看NVIDIA Perlin System例子)或者表现在游戏中一个物体是如何被损坏的。当需要一个快速变换的功能时你将会在许多地方使用到它。相反当需要一个非常慢的表现时(即使他们在开始的时候变换非常快)，对数功能lopg将会非常的游泳。对数功能是指数功能的对立面，这意味着logp指令可以用于撤销expp指令。

光照指令默认被方向光处理。它给予N*L,N*H和镜面反射幂来计算漫反射和镜面反射因素。计算结果并不含有衰减，但是你可以通过使用dst指令来计算个别的衰减等级。这对于构造点光源和面光源的衰减因数非常的游泳。

min和max指令允许截取和绝对值计算。

在Vertex Shader中的复杂指令

还有一些被vertex shader支持的复杂指令。虽然，它有点类似于宏，但"宏"这个术语并不能用于这些指令，因为它们并不是像C预编译宏一样的简单替换。在使用这些指令前，你必须考虑清楚。如果你使用这些指令，你或许会失去会使你超过128条指令的

限制和可能的优化路径。但在另一方面，Intel或者AMD提供的对于他们的处理器的软件模拟能够提供一个类似于m4x4的复杂指令（或者将会提供）。或者，在将来一些硬件或许会使用门数(gate count)来优化m4x4。所以，如果你需要，例如在你的vertex shader汇编代码中有4个dp4的调用，最好用m4x4来替换他们。如果你决定在你的shader中使用m4x4指令，以后你就不应该再使用dp4来调用相同的数据，因为在结果之间可能会有一些轻微的差别。

宏	参数	行为	时钟数
exp	dest, src1	提供精度至少在$1/2^{20}$ 的2的幂计算	12
frc	dest, src1	返回每一个输入部分的小数	3
log	dest, src1	提供精度至少在$1/2^{20}$ 的log2(x)计算	12
m3x2	dest, src1, src2	计算输入向量和一个3x2矩阵的乘积	2
m3x3	dest, src1, src2	计算输入向量和一个3x3矩阵的乘积	3
m3x4	dest, src1, src2	计算输入向量和一个3x4矩阵的乘积	4
m4x3	dest, src1, src2	计算输入向量和一个4x3矩阵的乘积	3
m4x4	dest, src1, src2	计算输入向量和一个4x4矩阵的乘积	4

你可以通过这些指令来执行所有的转换和光照操作。如果看上去好像还缺少一些指令，那肯定是因为你可以通过存在的指令来实现它们。例如，除法指令可以通过一个倒数和乘法指令来实现。你甚至可以在vertex shader中使用这些指令来实现整个fix-function管道渲染。你可以参看NVIDIA的例子NVLink。

将它们放到一起

现在让我们来看看在vertex shader运算器中这些寄存器和指令如何被典型的运用。

在vs 1.1中，每一个光栅化中，有16个输入寄存器，96个常量寄存器，12个暂存寄存器，1个地址寄存器和13个输出寄存器。没一个寄存器含有4x32bit的值，每一个32bit的值可以通过x,y,z和w来访问。为了访问这些寄存器部件，你必须加上.x,.y,.z和.w在这些寄存器名字的末尾。让我们从输入寄存器开始:

使用输入寄存器

16个输入寄存器可以通过使用它们的名字v0-v15来访问。在输入寄存器中提供的典型的值往往是这些:

- 位置(x,y,z,w)
- 漫反射光颜色 (r,g,b,a) -> 0.0 to +1.0
- 镜面反射光颜色(r,g,b,a) -> 0.0 to +1.0
- 最多8个材质坐标 (each as s, t, r, q or u, v , w, q) 一般为4-6个，具体依赖于硬件支持
- 雾 (f,*,*,*) -> 在雾等式中使用
- 点大小 (p,*,*,*)

你可以访问位置属性的x分量使用v0.x，访问y分量则使用v0.y。如果你需要知道RGBA的漫反射光颜色的R分量，你可以调用v1.y。如果使用雾属性，你设置v7.x为你需要的值，至于v7.y,v7.z,v7.w则将会废弃不用。输入寄存器是只读的，在每一条指令中都只可以访问一个输入寄存器。如果一个输入寄存器没有预先定义，那么x,y,z分量将是0,而z是1.0。在接下来的例子中，v0和c0-c3分别计算乘积并放入oPos中:

```
dp4 oPos.x , v0 , c0
dp4 oPos.y , v0 , c1
dp4 oPos.z , v0 , c2
dp4 oPos.w , v0 , c3
```

这样的一个代码片断通常用于从模型空间到裁减空间的映射。这四个部件的点乘积执行下面的计算。

```
oPos.x = (v0.x * c0.x) + (v0.y * c0.y) + (v0.z * c0.z) + (v0.w * c0.w)
```

如果我们使用单位长度(正则化)向量，很显然，两个向量之间的乘积的值将会在[-1,1]之间。因此,oPos也将会得到一个这个范围内的一个值。我们也可以这样使用:

```
m4x4 oPos, v0 , c0
```

别忘了这些事情，在你的vertex shader中一致的使用这些复杂指令，因为向上面描述的，在dp4和m4x4的结果之间可能会有轻微的差别。同时，你将被约束于在一条指令时只能使用一个寄存器。

所有的输入寄存器数据在整个vertex shade执行过程甚至更长的过程都持续存在。这意味着它们的数据将被保存比一个vertex shader生命周期还长的时间。也就是说，有可能重新使用输入寄存器的数据在下一个vertex shader。

使用常量寄存器

常量寄存器的典型应用包括:

- 矩阵数据:四元浮点数通常为一个4x4矩阵的一列
- 光属性(位置，衰减等等)
- 当前时间
- 顶点插值数据
- 程序使用数据

有96个四元浮点数可以存储常量数据。因此可以存储相当多的矩阵用于诸如，顶点索引混合(indexed vertex blending)，等操作。

常量寄存器在vertex shader中是只读的，尽管在程序中可以读和写常量寄存器。常量寄存器保持它们的数据并不仅仅在一个vertex shader的生存时间，所以可以在下一个vertex shader中重新使用这些数据。这避免了在程序中过多的 *SetVertexShaderConstant()* 调用。如果试图读一个没有设置过的常量寄存器，将会返回(0.0,0.0,0.0,0.0)。

在每一条指令中，你仅仅可以使用一个常量寄存器，但是不限次数。例如：

```
; 下面的指令是合法的
mul r5, c11, c11 ; c11的乘积存于r5中

; 这个不合法
add v0, c4, c3
```

一个更复杂，但合法的例子：

```
; dest = (src1 * src2) + src3
mad r0, r0, c20, c20 ; 将r0和c20相乘，然后加上c20
```

使用地址寄存器

你可以通过a0-an来访问地址寄存器（在vertex shader以后的版本中将会有超过一个的地址寄存器）.在vs1.1中a0的唯一用处就是作为常量寄存器的间接寻址。

```
c[a0.x + n] ; 仅仅在1.1或者以后版本中支持
            ; n是基址a0.x是地址偏移量
```

下面是一个使用地址寄存器的例子:

```
//Set 1
mov a0.x,r1.x
m4x3 r4,v0,c[a0.x + 9];
m3x3 r5,v3,c[a0.x + 9];
```

根据存在暂存寄存器中的值，不同的常量寄存器在m4x3和m3x3中被使用。请注意，寄存器a0仅仅存储整数部分，并且a0.x是a0可以用的唯一分量。而且vertex shader仅仅可以通过mov指令写a0.x。

如果在软件模拟模式下那请小心使用a0.x:它会显著的降低性能。

使用暂存寄存器

你可以通过r0-r11来访问暂存寄存器。下面是一些例子:

```
dp3 r2, r1, -c4 ; 一个三元的乘积: dest.x = dest.y = dest.z = dest.w
                                  = (r1.x * -c4.x) + (r1.y * -c4.y) + (r1.z * -c4.z)
...
mov r0.x, v0.x
mov r0.y, c4.w
mov r0.z, v0.y
mov r0.w, c4.w
```

每一个暂存寄存器有一个写和三个读的访问。因此，一个指令可以读同一个暂存寄存器3次。vertex shader不允许在写一个暂存寄存器之前读它。如果你试图读一个没有数据的暂存寄存器，当你在创建vertex shader的时候*CreateVertexShader()*将会给出一个出错信息。

使用输出寄存器

总共有13个只写的输出寄存器可以被访问。它们将被作为光栅化的输入，并且每一个寄存器的名字前面将会加上一个小写的'o'。输出寄存器被命名以建议它们被pixel shader使用。

名字	值	描述
oDn	2元浮点数	输出颜色值到pixel shader。oD0存放漫反射光颜色，oD1存放镜面反射光颜色。
oPos	1元浮点数	输出在裁减空间中的位置。必须被一个vertex shader所写。
oTn	至多8元浮点数 Geforce 3: 4 RADEON 8500: 6	输出的材质坐标。要求有材质的最大数量 和材质混合场景的范围
oPts.x	1标量浮点数	输出的点大小，仅仅x分量是可用的。
oFog.x	1标量浮点数	用于插值的雾因子，马上就被加入到雾列表中。仅仅x分量可用。

这里是一个典型的例子，显示了如何使用oPos,oD0和oT0寄存器:

```
dp4 oPos.x , v0 , c4 ; 投影的x位置
dp4 oPos.y , v0 , c5 ; 投影的y位置
dp4 oPos.z , v0 , c6 ; 投影的z位置
dp4 oPos.w , v0 , c7 ; 投影的w位置
mov oD0, v5          ; 设置漫反射光颜色
mov oT0, v2 ; 从输入寄存器v2中输出材质坐标到oT0
```

使用4条dp4

指令从模型空间映射到裁减空间已经在上面提到过了。第一个mov指令移动v5输入寄存器的值到颜色输出寄存器，第二个mov指令移动v2输入寄存器的值到第一个材质输出寄存器。

下面的例子演示了如何使用oFog.x寄存器:

```
; 按比例缩放按照雾参数
; c5.x = fog start
; c5.y = fog end
; c5.z = 1/range
; c5.w = fog max
dp4 r2, v0, c2 ; r2 = distance to camera
sge r3, c0, c0 ; r3 = 1
add r2, r2, -c5.x           ; camera space depth (z) - fog start
mad r3.x, -r2.x, c5.z, r3.x ; 1.0 - (z - fog start) * 1/range
                            ; because fog=1.0 means no fog, and
                            ; fog=0.0 means full fog
max oFog.x, c5.w, r3.x      ; 限制雾在我们规定的范围内
```

使用雾距离这个属性使得可以产生比使用位置的z,w值更多的雾效果。在以后的管道渲染中被使用的标准雾等式使用的雾距离值是被插值过的。

每一个vertex shader必须向oPos的一个分量写入值，否则编译器就会返回一个错误。

> *当使用vertex shader时， D3DTSS_TEXCOORDINDEX中的所有D3DTSS_TCI_*属性都会失效。所有的材质坐标都会被映射为数字顺序。*
>
> *优化提示:尽可能早的输出oPos，以触发pixel shader的平行度优化(trigger parallelism)。在写完shader汇编的时候，重新编排一下汇编指令的顺序，以便尽可能早的输出oPox。*

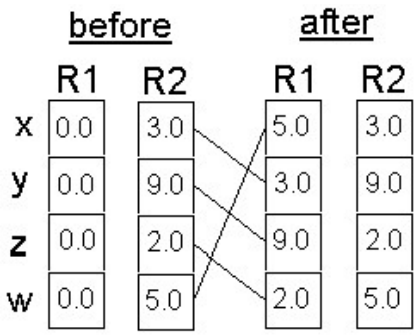
所有提到过的值输出vertex shader的时候都会被限制在[0..1]的范围之内。如果需要在pixel shader中使用有符号数，你必须在vertex shader中标记它们，然后在pixel shader使用_bx2重新扩展它们。

交叉混合和掩码

如果你使用输入、常量和暂存寄存器作为源寄存器，你可以独立的交叉混合每个.x,.y,.z和.w的值。如果使用输出寄存器或者暂存寄存器作为目标寄存器，你则可以使用.x,.y,.z,.w作为写入值的掩码。

交叉混合(仅仅源寄存器:vn,cn,rn)

交叉混合在源寄存器需要旋转过的交叉乘积时，对于效率的提高非常用帮助。交叉混合的另外一个用处是转换常量例如(0.5,0.0,1.0,0.6)到(0.0,0.0,1.0,0.0)或者(0.6,1.0,-0.5,0.6)之类的常量。



2wmov R1, R2.wxyz;

Figure 15 - Swizzling

这里目标寄存器是R1,R可以是一个任一个可写的寄存器例如output(o*)或者任何的暂存寄存器(r)。源寄存器是R2，R可以是输入寄存器(v)，常量寄存器(c)或者暂存寄存器（在指令语法结构中源寄存器的位置在目标寄存器的右边）。

接下来的指令拷贝R2.x的负数到R1.x，R2.y的负数到R1.y和R1.z,拷贝到R2.z的负数到R1.w。可以看到，所有的源寄存器可以在同一时间被交叉混合和取负。

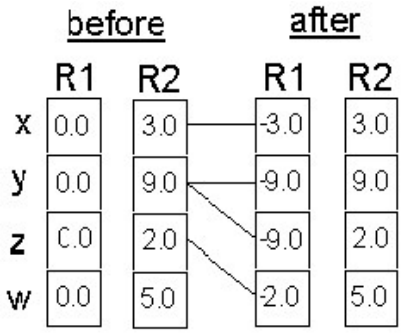


Figure 16 - Swizzling #2

掩码(仅仅目的寄存器:on,rn)

一个目的寄存器可以掩码指出哪个分量被写入。如果你使用R1作为目的寄存器(实际上可以是任何可写的寄存器:o*,r)，R2的每一个分量都会写入R1。但如果你使用这种形式:

```
mov  R1.x, R2
```

将仅仅只有x分量被写到了R1。

```
mov  R1.xw, R2
```

R2的x和w分量将会被写入R1。目标寄存器不支持交叉混合和取负。

下面是一个3维向量的交叉乘积计算:

```
; r0 = r1 x r2 (3-vector cross-product)
mul  r0, r1.yzxw, r2.zxyw
mad  r0, -r2.yzxw, r1.zxyw, r0
```

这在[LeGrand]中有详细的介绍。

下面的这个表格总结了交叉混合和掩码。

修改的分量	描述

R.[x][y][z][w]	目标寄存器掩码
R.xwzy (for example)	源寄存器交叉混合
-R	源取负

因为可以对目标取负，所以减法指令就没有存在的必要了。

编写Vertex Shader的方针

下面我列举了一些在编写vertex shader时你必须注意的东西：

- 至少要输出oPos 的一个分量
- 有128指令长度的限制
- 每一条指令中不得有超过一个的不同常量寄存器。例如，add r0,c3,c4这条指令是一条非法指令
- 每一条指令中不得有超过一个的暂存寄存器
- vertex shader中没有类C的条件判断语句，但是你可以使用sge指令模仿r0=(r1<=r2)?r3:r4之类的指令
- vertex shader输出的转换的值的范围都在[0..1]之间

有一些方法可以优化Vertex Shader,下面是最重要的一些部分:

- 阅读Kim Pallister的关于优化软件vertex shader的论文
- 设置常量寄存器的时候，设法在一个*SetVertexShaderConstant()* 调用中设置所有的值
- 暂停思考使用mov指令；或许你可以避免使用它的
- 尽量选择一次执行多重操作的指令

```
mad   r4,r3,c9,r4
mov   oD0,r4
==
mad   oD0,r3,c9,r4
```

- 在考虑优化前，移去例如m4x4,m3x3之类的复杂指令
- 一条CPU和GPU之间负载平衡的重要规则:在shader中的许多计算可以在外部重新以物体而非点为单位计算。如果你进行一个以物体而非点为单位的计算，那你最好使用CPU中计算，然后将将结果作为一个常量输入到vertex shader。

  压缩你的顶点数据是最有意思的一个优化你程序所需带宽的方法。

现在你对如何编写vertex shader已经有了一个抽象的概念，接下来，我将介绍3种编译vertex shader的方法。

编译Vertex Shader

OpenGL解析的是字符串，而Direct3D使用的是二进制字节。因此，Direct3D开发者需要使用编译器编译vertex shader。这可以帮助你尽早的在开发周期中发现bug，并且它也缩短了load时间。

我了解有3中方法可以编译一个vertex shader:

- 将vertex shader源代码写入到一个单独的ASCII文件中例如test.vsh，然后使用Vertex shader编译器将它编译成一个二进制文件，例如test.vso。编译后的文件将会在游戏开始后被读进来。使用这种方法，就不是每一个人都可以看到你的vertex shader源文件了。

  NVLink可以在运行期将已经编译过的shader段连接到一起。

- 当vertex shader源文件在一个ASCII文件或者以cpp文件中一个字符串形式出现时，你可以在程序启动后用 *D3DXAssembleShader*()* 加载vertex shader。
- 当vertex shader源文件在一个特效文件中，并被一个应用程序打开后。vertex shader可以使用 *D3DXCreateEffectFromFile()*来编译。通过这种方法也可以预编译vertex shader。使用这种方法的vertex shader大部分处理都很简单并且被特效文件的函数所调用。

  还有一个方法就是使用在d3dtypes.h中的操作码，写一个自己的vertex编译/反编译器。

让我们复习一下，看看我们已经学过了什么：

- 使用*D3DCAPS8::VertexShaderVersion* 检查vertex shader是否被支持
- 使用*D3DVSD_** 宏来定义vertex shader
- 使用*SetVertexShaderConstant()* 来设置常量寄存器
- 编写并编译vertex shader

现在让我们来学习如何得到一个vertex shader句柄并调用它。

创建Vertex Shader

CreateVertexShader() 函数用于创建一个vertex shader并使之生效：

```
HRESULT CreateVertexShader(
    CONST DWORD* pDeclaration,
    CONST DWORD* pFunction,
    DWORD* pHandle,
    DWORD Usage);
```

这个函数的第一个参数是指向以前定义的vertex shader的指针，并且返回一个shader句柄在pHandle中。第二个参数pFunction指向使用*D3DXAssembleShader()* | *D3DXAssembleShaderFromFile()* 编译过或者已经用编译器编译过的vertex shader的二进制代码。你可以强制 设置第四个参数为*D3DUSAGE_SOFTWAREPROCESSING* 来强制使用软件vertex模式。如果D3DRS_SOFTWAREVERTEXPROCESSING 被设置成真，第四个参数必须被设置。通过显式的设置软件模拟，vertex shader被CPU通过CPU厂商提供的软件接口模拟。如果有一个可以使用shader的GPU，使用硬件shader将会大大的提高速度。如果在用NVIDIA Shader调试器时，你必须设置这个属性或者使用参考光栅化(reference rasterizer)。

设置Vertex Shader

在调用*DrawPrimitive*()* 绘画一个物体之前，你必须使用*etVertexShader()* 为这个物体设置一个vertex shader.这个函数在两个primitive调用之间被动态的调用.

```
// set the vertex shader
m_pd3dDevice->SetVertexShader( m_dwVertexShader );
```

使用*SetVertexShader()* 调用的 Vertex shader 次数将等同于顶点数。例如，如果你试着用索引的三角形列表模式旋转一个四个顶点的四方形，你可以在NVIDIA Shader调试器中看到，在*DrawPrimitive*()* 函数被调用之前vertex shader运行了4次。

函数要传入的唯一参数就是你使用*CreateVertexShader()*创建的vertex shader句柄。这个函数调用的性能消耗相当的少，甚至比*SetTexture()* 还低，所以可以经常的使用。

释放Vertex Shader资源

当一个游戏结束或者一个设备改变了，vertex shader拥有的资源必须被释放。这通过调用*DeleteVertexShader()* 来完成：

```
// delete the vertex shader
if (m_pd3dDevice->m_dwVertexShader != 0xffffffff)
{
  m_pd3dDevice->DeleteVertexShader( m_dwVertexShader );
  m_pd3dDevice->m_dwVertexShader = 0xffffffff;
}
```

概要

现在我们已经大致了解了vertex shader的创建过程。让我们看一下迄今为止我们所学到的东西：

- 为了使用vertex shader，你必须检查你的最终用户电脑中所安装的软件或者硬件vertex shader接口 ，这可以通过检查*D3DCAPS8::VertexShaderVersion*来实现。
- 你必须要定义好，哪一个顶点属性或者哪个顶点数据被映射到哪个输入寄存器。这个定义由*D3DVSD_** 宏来完成。你还可以使用*SetVertexShaderConstant()* f或者提供的宏来设置vertex shader的常量寄存器。
- 当你准备好了任何东西并且已经写了一个vertex shader，你可以编译它，然后通过*CreateVertexShader()* 得到它的句柄，并最后调用*SetVertexShader()* 执行它。
- 为了释放vertex shader所申请的资源，你必须在游戏结束时调用*DeleteVertexShader()*释放这些资源。

下一章内容

在下一个章节"Vertex Shaders编程"中我们将开始编写我们自己的vertex shader。并且我们将讨论基本的光照算法和如何调用它们。

参考文献

[Bendel] Steffen Bendel, "Smooth Lighting with ps.1.4", *ShaderX*, Wordware Inc., pp ?? - ??, 2002, ISBN 1-55622-041-3

[Calver] Dean Calver, "Vertex Decompression in a Shader", *ShaderX*, Wordware Inc., pp ?? - ??, 2002, ISBN 1-55622-041-3

[Gosselin] David Gosselin, "Character Animation with Direct3D Vertex Shaders", *ShaderX*, Wordware Inc., pp ?? - ??, 2002, ISBN 1-55622-041-3

[Hurley] Kenneth Hurley, "Photo Realistic Faces with Vertex and Pixel Shaders", *ShaderX*, Wordware Inc., pp ?? - ??, 2002, ISBN 1-55622-041-3

[Isidoro/Gosslin], John Isidoro, David Gosselin, "Bubble Shader", *ShaderX*, Wordware Inc., pp ?? - ??, 2002, ISBN 1-55622-041-3

[LeGrand] Scott Le Grand, Some Overlooked Tricks for Vertex Shaders, *ShaderX*, Wordware Inc., pp ?? - ??, 2002, ISBN 1-55622-041-3

[Pallister] Kim Pallister, "Optimizing Software Vertex Shaders", *ShaderX*, Wordware Inc., pp ?? - ??, 2002, ISBN 1-55622-041-3

[Riddle/Zecha] Steven Riddle, Oliver C. Zecha, "Perlin Noise and Returning Results from Shader Programs", *ShaderX*, Wordware Inc., pp ?? - ??, 2002, ISBN 1-55622-041-3

[Schwab] John Schwab, "Basic Shader Development with Shader Studio", ShaderX, Wordware Inc., pp ?? - ??, 2002, ISBN 1-55622-041-3

[Vlachos01] Alex Vlachos, Jörg Peters, Chas Boyd and Jason L. Mitchell, "Curved PN Triangles", ACM Symposium on Interactive 3D Graphics, 2001 (http://www.ati.com/na/pages/resource_centre/dev_rel/CurvedPNTriangles.pdf).

其他资源

A lot of information on vertex shaders can be found at the web-sites of NVIDIA (developer.nvidia.com) and ATI (http://www.ati.com/). I would like to name a few:

Author	Article	Published at
Richard Huddy	Introduction to DX8 Vertex Shaders	NVIDIA web-site
Erik Lindholm, Mark J Kilgard, Henry Moreton	SIGGRAPH 2001 -- A User Programmable Vertex Engine	NVIDIA Web-Site
Evan Hart, Dave Gosselin, John Isidoro	Vertex Shading with Direct3D and OpenGL	ATI Web-Site
Jason L. Mitchell	Advanced Vertex and Pixel Shader Techniques	ATI Web-Site
Philip Taylor	Series of articles on Shader Programming	http://msdn.microsoft.com/directx
Keshav B. Channa	Geometry Skinning / Blending and Vertex Lighting	http://www.flipcode.com/tutorials/tut_dx8shaders.shtml
Konstantin Martynenko	Introduction to Shaders	http://www.reactorcritical.com/review-shadersintro/review-shadersintro.shtml

致谢

I'd like to recognize a couple of individuals that were involved in proof-reading and improving this paper (in alphabetical order):

- David Callele (University of Saskatchewan)
- Jeffrey Kiel (NVIDIA)
- Jason L. Mitchell (ATI)