



*n*VIDIA™

Introduction to Vertex Shaders

Richard Huddy

RichardH@nvidia.com

What you guys have been asking for...

- **Complete control of the transformation and lighting pipeline**
- **Custom vertex lighting**
- **Custom skinning and blending**
- **Custom texgen**
- **Custom texture matrix operations**
- **Insert vertex operation of your choice**



Enter the Vertex Shader

- **Assembly language interface to the transformation and lighting engine**
- **Instruction set to perform all vertex TnL**
- **Constant table to store data (matrices, light position, attenuation, etc)**
- **Registers to save intermediate data**
- **Reads an untransformed, unlit vertex**
- **Creates a transformed and lit vertex**
- **Your opportunity to look different**
- **New for DirectX 8**

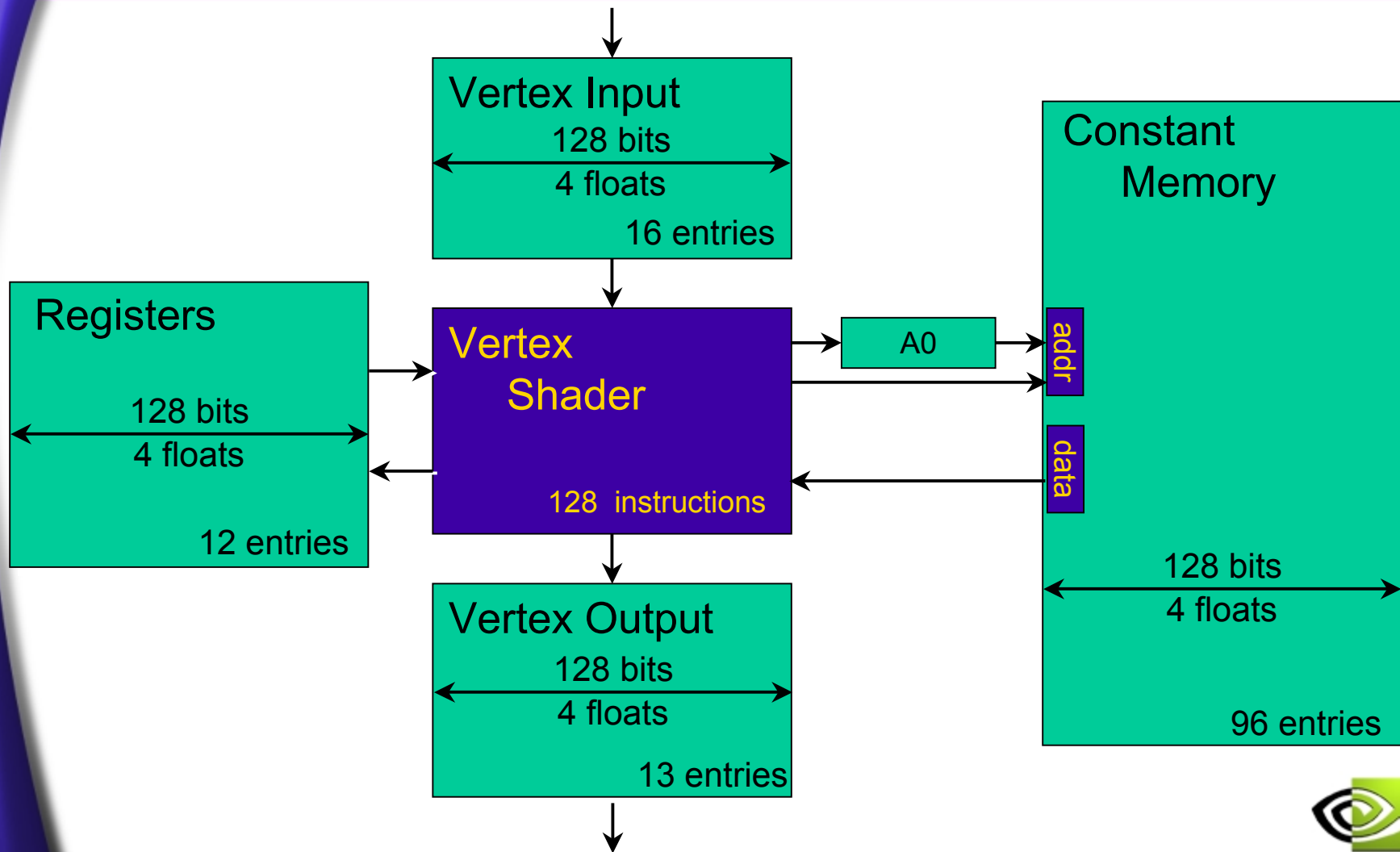


Assembly language

- **Fixed, complete, very powerful SIMD instruction set**
- **Four operations simultaneously (argb, xyzw)**
- **Dynamically loaded between primitive calls**
- **Extensive support for vector and matrix operations (lighting, rotations, etc.)**
- **Capable of efficiently implementing the entire functionality of DX7**



Custom Substitute for Standard T&L



What does it do?

- **Per *vertex* calculation**
- **Processing of:**
 - **Colors – true color, pseudo color**
 - **3D coordinates - procedural geometry, blending, morphing, deformations**
 - **Texture coordinates – texgens, set up for pixel shaders, tangent space bumpmap setup**
 - **Fog – elevation based, volume based**
 - **Point size**
- **Shader program accepts one input vertex, generates one output vertex**



What doesn't it do?

- **Does not perform polygon based operations**
 - **Back face culling**
 - **Two sided lighting (more on this later)**
 - **Occlusion culling**
- **Can't write to other vertices**
- **Does not create vertices**



What is calculated?

- **Create a completely specified vertex.**
 - **Vertex position in HCLIP space**
- **And, optionally:**
 - **Texgen/texture matrix/texture coord output**
 - **Lighting/color output**
 - **Fog**
 - **Point size**

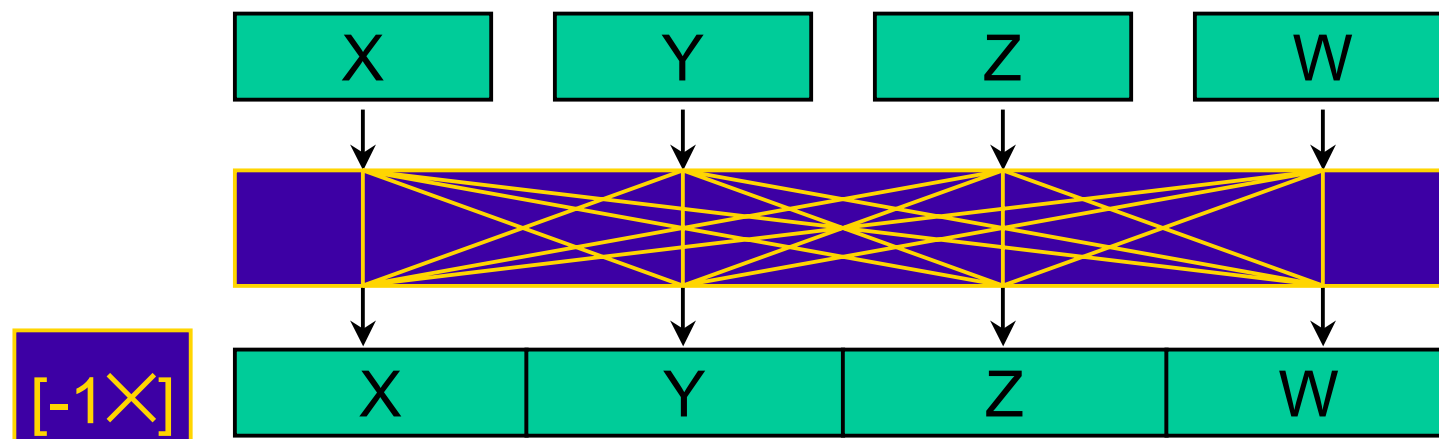


Then what happens?

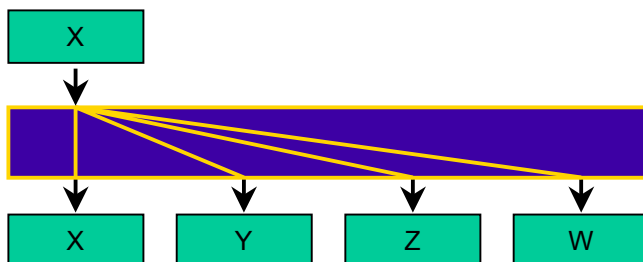
- **Frustum clip**
- **Homogenous divide**
- **Viewport Mapping**
- **Back Face Cull**
- **Rasterization**



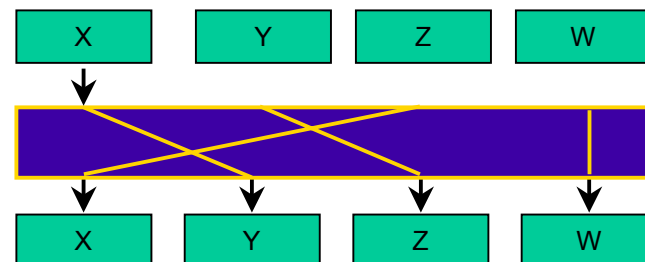
Flexible Input Sign and Muxing



V_{xxxx}



V_{zxyw}



Swizzles

Source registers can be swizzled:

```
MOV    R1, R2.yzwx;
```

before

R1	
0.0	x
0.0	y
0.0	z
0.0	w

R2	
7.0	x
3.0	y
6.0	z
2.0	w

after

R1	
3.0	x
6.0	y
2.0	z
7.0	w

R2	
7.0	x
3.0	y
6.0	z
2.0	w



NVIDIA™

Negations

Source registers can be negated (and swizzled):

```
MOV    R1, -R2.yzzx;
```

before

R1	
0.0	x
0.0	y
0.0	z
0.0	w

R2	
7.0	x
3.0	y
6.0	z
2.0	w

after

R1	
-3.0	x
-6.0	y
-6.0	z
-7.0	w

R2	
7.0	x
3.0	y
6.0	z
2.0	w



NVIDIA™

Masks

Destination register can mask which components are written to...

R1 ⇒ write all components

R1.x ⇒ write only x component

R1.xw ⇒ write only x, w components



The “Constant Area”

- **96 entries, each is a Vec4. Typical uses:**
 - **Matrix data - 4 of Vec4's are typically the matrix for the transform**
 - **Light characteristics, (position, attenuation etc)**
 - **Current time**
 - **Vertex interpolation data**
 - **Procedural data**
 - **Only one constant per instruction (but you can use it several times if you want)**
E.g. `mad r1, v0, C[95], C[95]`

Note that this is how you access *all* constant data



NVIDIA

Input Vertex Data

- **16 Vec4's from your own VertexBuffer(s)**
- **Input vertex is completely flexible**
 - “Weakly typed” – meaning it's up to you to interpret it consistently
- **Position, normal, texture coordinates etc.**



Vertex output data

- **A well defined vertex**
 - **HCLIP(x,y,z,w)**
 - **Diffuse color (r,g,b,a) -> 0.0 to +1.0**
 - **Specular color (r,g,b,a) -> 0.0 to +1.0**
 - **Up to 4 Texture coordinates (each as s,t,r,q)**
 - **One for each physical hardware texture unit**
 - **Fog (f,*,*,*) -> value used in fog equation**
 - **Point size (p,*,*,*)**
- **Outputs of shader clamped as required**



Instruction format

Generally of the form:

OpName dest, [-]s1 [, [-]s2 [, [-]s3]] ;comment

e.g.

mov r1, r2

mad r1, r2, r3, r4

Destination 'r' can have a write-mask

Source 'r' can be swizzled

e.g. mov r1.x, r2.y mov r1, r2.zxyw

'[' and ']' indicate optional modifiers



What are the instructions?

- `nop`
- `mov`
- `mul`
- `mad`
- `add`
- `rsq`
- `dp3`
- `dp4`
- `dst`
- `lit`
- `min`
- `max`
- `slt`
- `sge`
- `expp`
- `log`
- `rcp`



nop, mov, mul

- **nop**
 - Do nothing
- **mov dest, src**
 - Move (with conditional sign change, mask and swizzle)
- **mul dest, src1, src2**
 - Set dest to the product of src1 and src2



add, mad, rsq

- **add dest, src1, src2**
 - Add src1 to src2. [And the optional negation creates subtraction]
- **mad dest, src1, src2, src3**
 - Multiply src1 by src2 and add src3 - into dst
- **rsq dest, src**
 - Source must have one subscript...
 - $\text{dest.x} = \text{dest.y} = \text{dest.z} = \text{dest.w} = 1/\text{sqrt}(\text{src})$
 - Reciprocal square root of src (much more useful than straight 'square root').



dp3, dp4

- 3 and 4 Component dot products
- **dp3** **dest, src1, src2**
 - **dest.x = dest.y = dest.z = dest.w =**
 - **(src1.x * src2.x) +**
 - **(src1.y * src2.y) +**
 - **(src1.z * src2.z)**
- **And dp4 does the same but includes 'w' in the computation**



min, max

- **min dest, src1, src2**
 - **Component-wise min operation**
- **max dest, src1, src2**
 - **Component-wise max operation**



slt, sge

- **slt** **dest, src1, src2**
 - $\text{dest} = (\text{src1} < \text{src2}) ? 1 : 0$
 - For each component...
- **sge** **dest, src1, src2**
 - $\text{dst} = (\text{src1} \geq \text{src2}) ? 1 : 0$
 - Which is equivalent to...
 - $\text{dst} = (\text{src1} < \text{src2}) ? 0 : 1$
 - i.e. the exact opposite of slt
 - For each component...



dst

- **dst dest, src1, src2**
 - Calculate distance vector. src1 vector is $(NA, d*d, d*d, NA)$ and src2 is $(NA, 1/d, NA, 1/d)$.
 - dest is set to $(1, d, d*d, 1/d)$
- Which is what you want for standard attenuation...



lit

- **lit dest, src**
 - **Calculates lighting coefficients from two dot products and a power. src is:**
 - **$\text{src.x} = n \cdot l$ (unit normal and light vectors)**
 - **$\text{src.y} = n \cdot h$ (unit normal and halfangle vectors)**
 - **src.z is unused**
 - **src.w = power (in range +128 to -128)**
 - **dest set to (1.0, src.x, L, 1.0)**
 - **If $\text{src.x} > 0.0$**
 - **$L = (\text{MAX}(\text{src.y}, 0))^{\text{src.w}}$**
 - **else $L = 0$**



expp, log

- **expp dest, src.w**
 - **dest.x = 2 ** (int)src.w**
 - **dest.y = fractional part (src.w)**
 - **dest.z = 2 ** src.w**
 - **dest.w = 1.0**
- **log dest, src.w**
 - **dest.x = exponent((int)src.w)**
 - **dest.y = mantissa(src.w)**
 - **dest.z = log2(src.w)**
 - **dest.w = 1.0**



rcp

- **rcp dest, src.w**
 - Source must have just one subscript (x, y, z or w)
 - $\text{dest.x} = \text{dest.y} = \text{dest.z} = \text{dest.w} = 1 / \text{src.w}$
 - So... this is the other half of the puzzle for division
 - ... you divide by doing a 'rcp' and then a 'mul'



Minimal code snippet...

```
#define CV_WORLDVIEWPROJ0 0
#define CV_WORLDVIEWPROJ1 1
#define CV_WORLDVIEWPROJ2 2
#define CV_WORLDVIEWPROJ3 3
#define CV_FIXED_COLOR 4

;transform to clip space
dp4 oPos.x, v0, c[WORLDVIEWPROJ0]
dp4 oPos.y, v0, c[WORLDVIEWPROJ1]
dp4 oPos.z, v0, c[WORLDVIEWPROJ2]
dp4 oPos.w, v0, c[WORLDVIEWPROJ3]
;write out color
mov oD0, c[CV_FIXED_COLOR]
```



These instructions give you complete control over the T & L hardware

- Use them to get more complex operations:
 - `ABS` = `max s1, -s1`
 - `-ABS` = `min s1, -s1`
 - `XFORM4` = `4 dp4's`
 - `DIV` = `rcp, mul`
 - `SUB` = `add s1, -s2`
 - `SEQ` = `sle d1, s1, s2`
 - `sge d2, s1, s2`
 - `mul d1, d2`
 - `SNE` = `SEQ` but with `add` in place of `mul`
 - `CLAMP0` = `max c[ZERO], s1`
 - `CLAMP1` = `min c[ONE], s1`



How do I branch?

- No branching, no early out
- Why?
 - Performance and predictability
 - No execution dependencies
- You *can* multiply by zero, and accumulate:
 - See 'Two Sided Lighting' slide for a classic example...



Two sided lighting...

```
; Dot face normal with eye vector (from eye to vertex)
dp3 r1, R_FACE_NORMAL, R_EYE_VECTOR

; (front_facing ? 1 : 0) into R6.x
sge r6.x, r1.x, c[C_ZERO].x

; (front_facing ? 0 : 1) into R6.y
slt r6.y, r1.x, c[C_ZERO].x

; Get the intensity of the lighting for the front face
; Dot normal with light direction in eye space
dp3 r5.x, R_NORMAL, c[C_LIGHT1_DIRECTION]
dp3 r5.y, -R_NORMAL, c[C_LIGHT1_DIRECTION]

; Put front and back colors into R7 and R8
mul r7, r5.x, c[C_FRONTCOLOR]
mul r8, r5.y, c[C_BACKCOLOR]

; Pick the right color
; and add the front and back lighting calculation together
mul r7, r7, r6.x
mad oD0, r8, r6.y, r7
```



Efficient Instructions

- **Full 4x4 XFORM (4 ops)**
- **Simple lighting (2 instructions per directional)**
 - **dp3 r1, r2, C[Light_Vector]**
 - **mad r3, C[Light_Colour], r1, r4**
- **lit, dst instructions for lighting calculations**
- **Clamping some of the results ('lit' and some outputs have auto clamping to 0 or 1)**
 - **Color value output:-**
 - **mad oD0, r4, r1, r5**



Plus: Novel Effects...

- **Irregular transform**
 - **Like Fish-Eye lens**
- **Novel texture coordinate calculations**
 - **Projected textures**
- **Paletted skinning with 20 or more bones!**
 - **Now you can be much more efficient than with DX7**
- **Geometry morphing**
 - **Blending multiple meshes**
- **Procedural Geometry Deformations**



A quick look at errors

- **Because I can't avoid API specifics here...**
 - I'll use a prefix to identify those parts
 - "D3D:" or "GL:"



Parse Problems I

- **Make sure your program...**
 - **Writes out the hclip space position of the vertex**
 - **D3D: oPos**
 - **GL: o[HPOS]**
 - **Has at most 128 native instructions**
 - **D3D: Watch out for macro expansion...**
 - **Has no syntax errors**
 - **D3D: Has a valid version header “vs.1.0”**
 - **GL: Has a valid version header “!!VP1.0”**
 - **Address Register loads only to the x component of A0:**
 - **D3D: MOV A0.x, R0.y**
 - **GL: ARL A0.x, R0.y;**



Parse Problems II

- **Make sure your program...**
 - **GL: Uses program parameter indices in range [0 - 95]**
 - **D3D: Uses constant indices in range [0 - 95]**
 - **GL: Uses vertex attribute indices in range [0 – 15]**
 - **(D3D names are v0-v15)**
- **No instruction has:**
 - **More than one unique vertex attribute register as source:**
`ADD R0, v[NRML], v[POS] ;<- fails to compile`
 - **More than one unique program parameter (D3D: 'constant') register as source:**
`ADD R0, c[1], c[0] ;<- fails to compile`



GL: Vertex State Program Parse Problems

- **Make sure your program...**
 - **Writes out at least one program parameter register**
 - **Has at most 128 instructions**
 - **Has the required header “!!VSP1.0”**
 - **No instruction reads from more than one program parameter register**
 - **ADD c[0], c[1], c[1] ;OK**
 - **ADD c[0], c[1], c[2] ;FAILS**
 - **Uses only vertex attribute v[0] (if it uses any...)**



GL: Error management I

- **If `glGetError()` returns `INVALID_OPERATION`**
 - Anything we've met so far has occurred, or,
 - The program Id is already being used by a program that has a different target, i.e. Vertex Program vs Vertex State Program
- **Use `glGetIntegerv` with `GL_PROGRAM_ERROR_POSITION_NV` to get the unsigned byte offset at which the last error occurred in the program string**
- **If `glGetError()` returns `OUT_OF_MEMORY`**
 - Attempting to replace an existing program and getting this error means the original program is unaffected



D3D: Error Management

- **D3DXAssembleShader function call returns a pointer to a D3DXBUFFER, which contains error messages**
- **You can view debug output, when you create the vertex/pixel shader**
- **Running nvasm.exe will validate the shader for you and report validation problems.**

