Introduction to the DirectX 9 Shader Models

Sim Dietrich
SDietrich@nvidia.com

Jason L. Mitchell
JasonM@ati.com

Outline

- Vertex Shaders
  - vs_2_0 and extended bits
  - vs_3_0
- Pixel Shaders
  - ps_2_0 and extended bits
  - ps_3_0

Legacy 1.*x* Shaders

- No discussion of legacy shaders today
- There is plenty of material on these models from Microsoft and IHVs

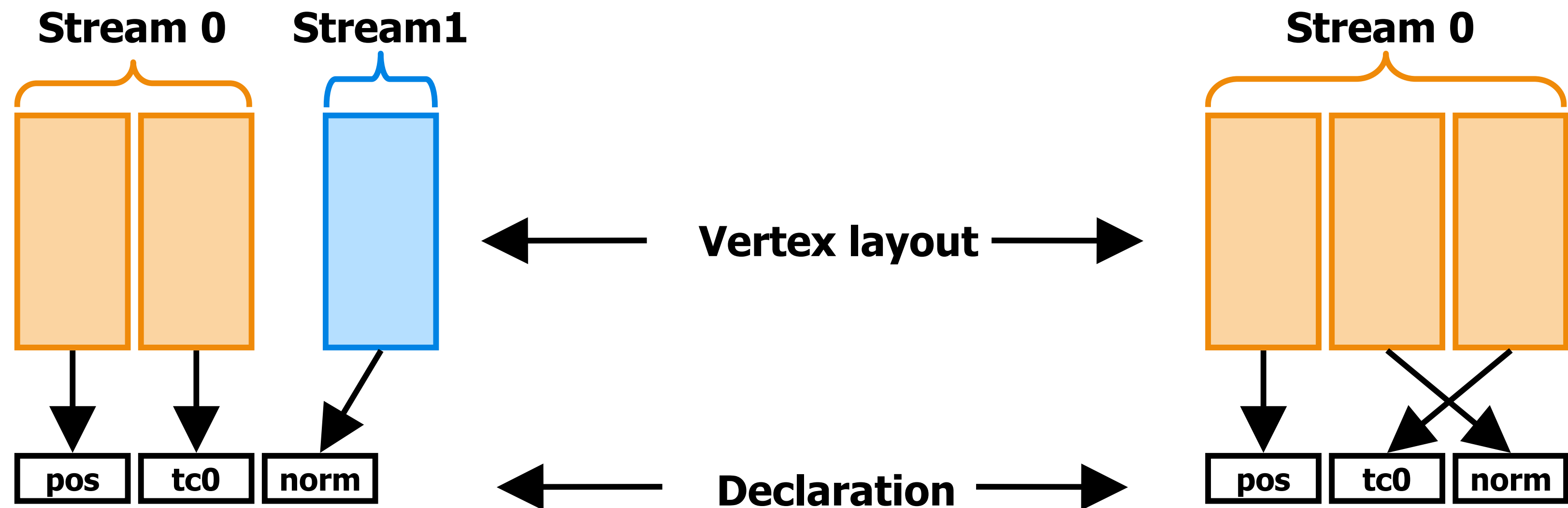
What's with all the asm?

- We'll show a lot of asm in this section to introduce the virtual shader machines
- Later talks today will focus on HLSL since that is what we expect most developers will prefer to use
- Familiarity with the asm is useful, however. You'll find that familiarity with the various virtual shader machines will help in development, even in HLSL.
- Also, the currently-available HLSL compiler does not support flow-control, though that is coming in a future revision. Microsoft will discuss more details of this in their event tomorrow.

vs_2_0

- Longer programs
- Integration with declarations
- Some new instructions
- Control flow
  - New instructions
  - New registers

Declarations



asm:

```
vs 1.1
dcl_position   v0
dcl_normal     v1
dcl_texcoord0  v2

mov r0, v0
…
```

HLSL:

```
VS_OUTPUT main (
float4 vPosition : POSITION,
float3 vNormal   : NORMAL,
float2 vTC0      : TEXCOORD0)
{
   …
}
```

vs_2_0

- Old reliable ALU instructions and macros
  - add, dp3, dp4, mad, max, min, mov, mul, rcp, rsq, sge, slt
  - exp, frc, log, logp, m3x2, m3x3, m3x4, m4x3 and m4x4
- New ALU instructions and macros
  - abs, crs, mova
  - expp, lrp, nrm, pow, sgn, sincos
- New control flow instructions
  - call, callnz, label, ret
  - if, else, endif
  - loop…endloop, endrep…rep

vs_2_0 registers

- Floating point registers
  - 16 Inputs (v_n)
  - 12 Temps (r_n)
  - At least 256 Constants (c_n)
    - Cap'd: `MaxVertexShaderConst`
- Integer registers
  - 16 Loop counters (i_n)
- Boolean scalar registers
  - 16 Control flow (b_n)
- Address Registers
  - 4D vector: a0
  - scalar loop counter (only valid in loop): aL

Setting Vertex Shader Registers

- Because of the new types of constant registers in DirectX 9, there are new APIs for setting them from the app:
  - SetVertexShaderConstantF()
  - SetVertexShaderConstantI()
  - SetVertexShaderConstantB()

vs_2_0 registers

- Output position (oPos)
  - Must be written (all 4 components)
- Eight Texture Coordinates (oT0..oT7)
- Two colors (oD0 and oD1)
- Scalar Fog (oFog)
- Scalar Point Size (oPts)

Control Flow

- Basic mechanism for reducing the shader permutation problem
- Enables you to write and manage fewer shaders
- You can control the flow of execution through a small number of key shaders
- Initial rev of HLSL compiler did not generate flow-control instructions in bytecode output. Microsoft is working on this now.

Control Flow Instructions

- **Loops**
  - `loop…endloop`
  - `rep…endrep`
- **Conditional**
  - `if…else…endif`
- **Subroutines**
  - `call`, `callnz`
  - `ret`

Two kinds of loops

- **Must be completely inside an if block, or completely outside of it**

- `loop aL, i`$_n$
  - `i`$_n$`.x` **- Iteration count (non-negative)**
  - `i`$_n$`.y` **- Initial value of `aL` (non-negative)**
  - `i`$_n$`.z` **- Increment for `aL` (can be negative)**
  - `aL` **can be used to index the constant store**
  - **No nesting in vs_2_0**
- `rep i`$_n$
  - `i`$_n$ **- Number of times to loop**
  - **No nesting**

Conditionals

- Simple if…else…endif construction
- May be nested
- Based on Boolean constants set through `SetVertexShaderConstantB()`

```
if b3
    // Instructions to run if b3 TRUE
else
    // Instructions to run otherwise
endif
```

HLSL and Conditionals

- Initial revision of HLSL compiler which shipped with the DirectX 9.0 SDK would handle conditionals via executing all codepaths and lerping to select outputs
- Future SDK releases will provide compiler which can compile to asm control flow instructions
- Microsoft will discuss this in more detail at their event tomorrow

HLSL Conditional Example

```
VS_OUTPUT main (float4 vPosition0 : POSITION0, float4 vPosition1 : POSITION1, float4 vPosition2 : POSITION2,
                float3 vNormal0   : NORMAL0,   float3 vNormal1   : NORMAL1,   float3 vNormal2   : NORMAL2,
                float2 fTC0 : TEXCOORD0)
{
   float3 lightingNormal, cameraSpacePosition, tweenedPosition;
   VS_OUTPUT Out = (VS_OUTPUT) 0;

   if (bTweening)
   {
      tweenedPosition = vPosition0.xyz * fWeight.x + vPosition1.xyz * fWeight.y + vPosition2.xyz * fWeight.z;
      Out.Pos = mul (worldview_proj_matrix, tweenedPosition);

      lightingNormal = vNormal0 * fWeight.x + vNormal1 * fWeight.y + vNormal2 * fWeight.z;
      cameraSpacePosition = mul (worldview_matrix, tweenedPosition);
   }
   else
   {  Out.Pos = mul (worldview_proj_matrix, vPosition0);
      lightingNormal = vNormal0;
      cameraSpacePosition = mul (worldview_matrix, vPosition0);
   }
   // Compute per-vertex color from lighting
   Out.Color = saturate(saturate(dot(modelSpaceLightDir, lightingNormal)) * matDiffuse + matAmbient);

   Out.TCoord0 = fTC0; // Just copy texture coordinates

   // f = (fog_end - dist)*(1/(fog_end-fog_start))
   Out.Fog = saturate((fFog.y - cameraSpacePosition) * fFog.z);

   return Out;
}
```

Original HLSL Compiler Results

```
vs_2_0
def c1, 0, 0, 0, 1
dcl_position v0
dcl_position1 v1
dcl_position2 v2
dcl_normal v3
dcl_normal1 v4
dcl_normal2 v5
dcl_texcoord v6
mul r0, v0.x, c4
mad r2, v0.y, c5, r0
mad r4, v0.z, c6, r2
mad r6, v0.w, c7, r4
mul r8.xyz, v0, c2.x
mad r10.xyz, v1, c2.y, r8
mad r0.xyz, v2, c2.z, r10
mul r7, r0.x, c4
mad r9, r0.y, c5, r7
mad r11, r0.z, c6, r9
add r1, -r6, r11
mad oPos, r1, c0.x, r6
mul r6.xyz, v3, c2.x
mad r10.xyz, v4, c2.y, r6
mad r7.xyz, v5, c2.z, r10
lrp r11.xyz, c0.x, r7, v3
dp3 r0.w, c19, r11
max r0.w, r0.w, c1.x
min r0.w, r0.w, c1.w
mul r1, r0.w, c21
add r8, r1, c22
max r6, r8, c1.x
min oD0, r6, c1.w
mul r0.w, r0.x, c8.x
mad r5.w, r0.y, c9.x, r0.w
mad r7.w, r0.z, c10.x, r5.w
mul r2.w, v0.x, c8.x
mad r4.w, v0.y, c9.x, r2.w
mad r1.w, v0.z, c10.x, r4.w
mad r6.w, v0.w, c11.x, r1.w
lrp r5.w, c0.x, r7.w, r6.w
add r2.w, -r5.w, c23.y
mul r9.w, r2.w, c23.z
max r4.w, r9.w, c1.x
min oFog, r4.w, c1.w
mov oT0.xy, v6
```

38 ALU ops every time

Beta1 Compiler Results

```
vs_2_0
def c0, 0, 0, 0, 1
dcl_position v0
dcl_position1 v1
dcl_position2 v2
dcl_normal v3
dcl_normal1 v4
dcl_normal2 v5
dcl_texcoord v6
if b0
  mul r0.xyz, v0, c2.x
  mad r2.xyz, v1, c2.y, r0
  mad r4.xyz, v2, c2.z, r2
  mul r11, r4.x, c4
  mad r1, r4.y, c5, r11
  mad r3, r4.z, c6, r1
  mul r10.xyz, v3, c2.x
  mad r0.xyz, v4, c2.y, r10
  mad r2.xyz, v5, c2.z, r0
  mul r2.w, r4.x, c8.x
  mad r2.w, r4.y, c9.x, r2.w
  mad r2.w, r4.z, c10.x, r2.w
  mov oPos, r3
else
  mul r11, v0.x, c4
  mad r1, v0.y, c5, r11
  mad r10, v0.z, c6, r1
  mad r0, v0.w, c7, r10
  mul r7.w, v0.x, c8.x
  mad r9.w, v0.y, c9.x, r7.w
  mad r6.w, v0.z, c10.x, r9.w
  mad r8.w, v0.w, c11.x, r6.w
  mov r8.xyz, v3
  mov r2.xyz, r8
  mov r2.w, r8.w
  mov oPos, r0
endif
dp3 r3.w, c19, r2
max r10.w, r3.w, c0.x
min r5.w, r10.w, c0.w
mul r0, r5.w, c21
add r7, r0, c22
max r4, r7, c0.x
min oD0, r4, c0.w
add r11.w, -r2.w, c23.y
mul r6.w, r11.w, c23.z
max r1.w, r6.w, c0.x
min oFog, r1.w, c0.w
mov oT0.xy, v6
```

24 or 25 ALU ops, depending on bTweening

Currently requires the -Gp compiler flag

Subroutines

- Can only call forward
- Can be called inside of a loop
  - aL is accessible inside that loop
- No nesting in vs_2_0
- Limited nesting depth in vs_2_x
  - See `StaticFlowControlDepth` member of `D3DVSHADERCAPS2_0`
- Limited to 4 in vs_3_0

Returning from Subroutines

- Limited stack, hence the ability to nest subroutines is limited
- Cannot have more than one return in main or in any subroutine. The first return is treated as the end of main or subroutine

Gotchas

- **Loading `a0` now rounds-to-nearest rather than truncating**
  - **Can only be loaded with the `mova` instruction**
  - **Not hard to cope with but could surprise you…**

- **No jumping into or out of loops or subroutines**

Extended shaders

- **What are the extended shader models?**
- **A shader is considered 2_x level if it requires at least one capability beyond stock ps.2.0 or vs.2.0**
- **Use the 'vs_2_x' or 'ps_2_x' label in your assembly or as your compilation target**

Caps for vs_2_x

- **New D3DVSHADERCAPS2_0 structure**

D3DCAPS9.VS20Caps	CineFX support
Caps	D3DVS20CAPS_PREDICATION
NumTemps	16
StaticFlowControlDepth	4
DynamicFlowControlDepth	24

Extended Vertex Shader Caps

- D3DVS20CAPS_PREDICATION
- Predication is a method of conditionally executing code on a per-component basis
- For instance, you could test if a value was > 0.0f before performing a RSQ
- Faster than executing a branch for short code sequences

Vertex Shader Predication – HLSL

This example adds in specular when the self-shadowing term is greater than zero.

```
if ( dot(light_vector, vertex_normal) > 0.0f )
{
   total_light += light_color;
}
```

If targeting a vs_2_x profile, the compiler should use predication for a short conditional block

Vertex Shader Predication - ASM

With Predication :

```
def c0, 0.0f, 0.0f, 0.0f, 0.0f
. . .
setp_gt p0, r1.w, c0.w
p0 add r0, r0, r2
```

Without Predication :

```
def c0, 0.0f, 0.0f, 0.0f, 0.0f
. . .
sgt r3.w, r1.w, c0.w
mad r0, r3.w, r2, r0
```

Vertex Shader Predication Details

- Predication requires fewer temporaries
- Can use predication on any math instruction
- Predication
  - 4D Predication register : p0
    - SETP, then call instructions with _PRED
    - BREAK_PRED, CALLNZ_PRED, IF_PRED
  - Can invert predication using "not" (!)
  - Predication very useful for short IF / ELSEIF blocks

Nested Static Flow Control

```
for ( int i = 0; i < light_count; ++i )
{
    total_light += calculate_lighting();
}
```

- You can nest function calls inside loops
  - Also nested loops
- Both count toward limit reported in :

`D3DCAPS9.VS20Caps.StaticFlowControlDepth`

Nested Static Flow Control

- **Static flow control is a standard part of vs_2_0**
  - **CALL / CALLNZ-RET, LOOP-ENDLOOP**
- **Most useful for looping over light counts**
  - **As long as they aren't shadowed**

Dynamic Flow Control

- **Branching information is derived *per vertex***
- **Can be based on arbitrary calculation, not just constants**
- **Best used to skip a large number of instructions**
  - **Some architectures may pay a perf penalty when vertices take disparate branches**

Dynamic Flow Control - HLSL

```
for ( int i = 0; i < light_count; ++i )
{
   // calculate dist_to_light

   if ( dist_to_light < light_range )
   {
         // perform lighting calculation
   }
}
```

Dynamic Flow Control

- Very useful to improve batching for matrix palette skinning
- Do a dynamic loop over the # of non-zero bones in the vertex
  - Sort per-vertex indices & weights by priority
  - If the a weight is zero, break out of skinning loop
- Can do automatic shader LOD
  - Distance to light or viewer large enough
  - Or when fully fogged, etc.

vs_3_0

- Longer programs (512 minimum)
- Dynamic flow-control
  - Also in vs_2_x if `VS20Caps.DynamicFlowControlDepth` > 0
  - `break`
  - Comparisons
    - `break_`*`comp`*, `if_`*`comp`*
    - Where *comp* is one of `_gt`, `_lt`, `_ge`, `_le`, `_eq` or `_ne`
    - Scalar operations, so replicate swizzle is required on arguments
  - Predication
    - Also in vs_2_*x* with the `D3DVS20CAPS_PREDICATION` bit set
    - Kind of a programmable write-mask
    - `setp` – Sets the predicate register `p0`
    - `break_pred`, `callnz_pred`, `if_pred`
    - Can precede ALU ops as well
- Access to textures!
  - `texldl`
  - No dependent read limit

Vertex Texturing

- With the `texldl` instruction, a vs_3_0 shader can access memory
- The LOD is computed by the shader, hence the use of the `texldl` instruction
- This represents the next step in the march toward unifying the vertex and pixel shader programming models

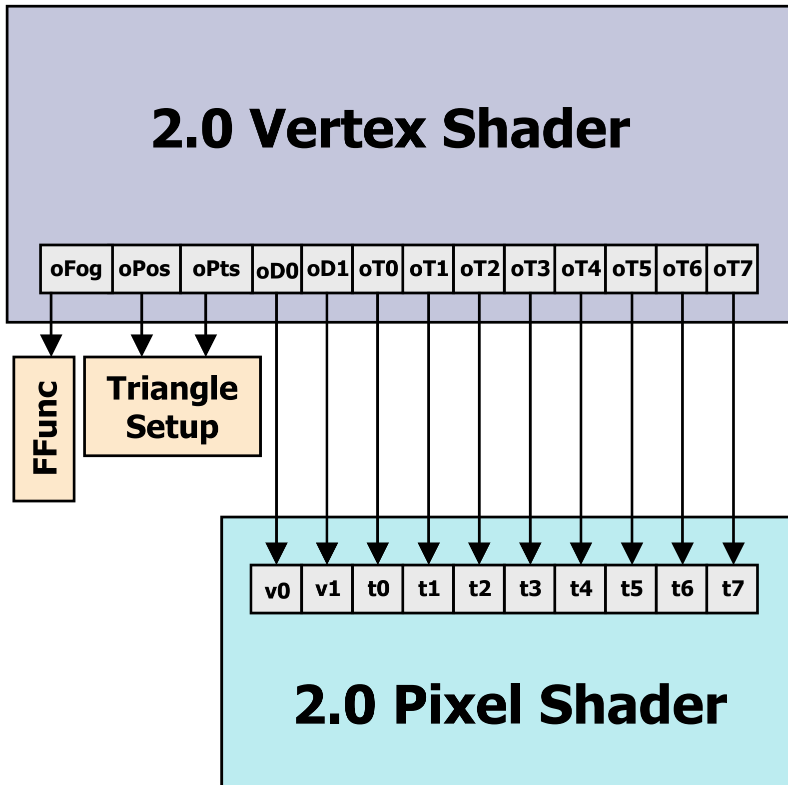
Applications of Vertex Texturing

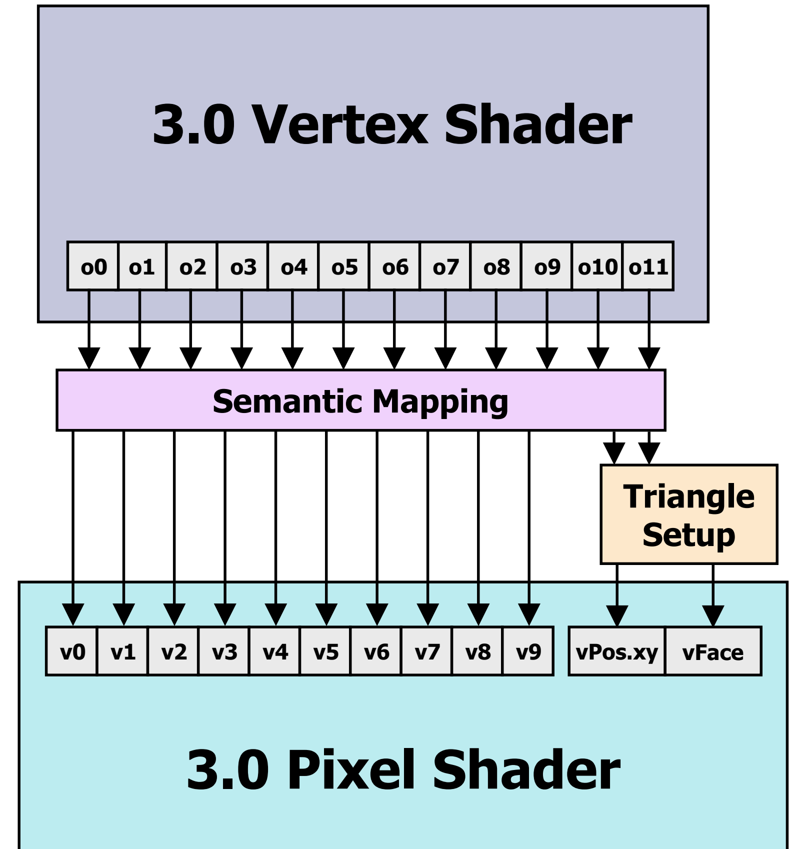
- Displacement mapping
- Large off-chip matrix palette
- Tons of other uses!

vs_3_0 Outputs

- **12 generic output (o_n) registers**
- **Must declare their semantics up-front like the input registers**
- **Can be used for any interpolated quantity (plus point size)**
- **There must be one output with the positiont semantic**

Connecting VS to PS





vs_3_0 Semantic Declaration

```
vs_3_0
dcl_color4 o3.x          // color4 is a semantic name
dcl_texcoord3 o3.yz      // Different semantics can be packed into one register
dcl_fog o3.w
dcl_tangent o4.xyz
dcl_positiont o7.xyzw    // positiont must be declared to some unique register
                         // in a vertex shader, with all 4 components
dcl_psize o6             // Pointsize cannot have a mask
...
```

ps_2_0

- More sophisticated programs than 1.*x*
  - 64 ALU operations
  - 32 texture operations
  - 4 levels of dependent read
- Floating point
- No flow control

2.0 Pixel Shader Instruction Set

- **ALU Instructions**
  - `add`, `mov`, `mul`, `mad`, `dp2add`, `dp3`, `dp4`, `frc`, `rcp`, `rsq`, `exp`, `log` **and** `cmp`
- **ALU Macros**
  - `MIN`, `MAX`, `LRP`, `POW`, `CRS`, `NRM`, `ABS`, `SINCOS`, `M4X4`, `M4X3`, `M3X3` **and** `M3X2`
- **Texture Instructions**
  - `texld`, `texldp`, `texldb` **and** `texkill`

2.0 Pixel Shader Resources

- 12 Temp registers
- 8 4D texture coordinate iterators
- 2 color iterators
- 32 4D constants
- 16 Samplers
- Explicit output registers
  - `oC0`, `oC1`, `oC2`, `oC3` and `oDepth`
  - Must be written with a `mov`
- Some things removed
  - No source modifiers except negate
  - No instruction modifiers except saturate
  - No Co-issue

Argument Swizzles

- `.r`, `.rrrr`, `.xxxx` or `.x`
- `.g`, `.gggg`, `.yyyy` or `.y`
- `.b`, `.bbbb`, `.zzzz` or `.z`
- `.a`, `.aaaa`, `.wwww` or `.w`
- `.xyzw` or `.rgba` (No swizzle) or nothing
- `.yzxw` or `.gbra` (can be used to perform a cross product operation in 2 clocks)
- `.zxyw` or `.brga` (can be used to perform a cross product operation in 2 clocks)
- `.wzyx` or `.abgr` (can be used to reverse the order of any number of components)

Samplers

- Separated from texture stages
- Sampler State
  - **U, V and W address modes**
  - **Minification, Magnification and Mip filter modes**
  - **LOD Bias**
  - **Max MIP level**
  - **Max Anisotropy**
  - **Border color**
  - **sRGB conversion (new in DirectX® 9)**
- Texture Stage State
  - **Color, Alpha and result Arguments for legacy multitexture**
  - **Color and Alpha Operations for legacy multitexture**
  - **EMBM matrix, scale and offset**
  - **Texture coordinate index for legacy multitexture**
  - **Texture Transform Flags**
  - **Per-stage constant (new in DirectX® 9)**

ps.2.0 Review – Comparison with ps.1.4

```
ps.1.4

texld r0, t0 ; base map
texld r1, t1 ; bump map

; light vector from normalizer cube map
texld r2, t2

; half angle vector from normalizer cube map
texld r3, t3

; N.L
dp3_sat r2, r1_bx2, r2_bx2

; N.L * diffuse_light_color
mul r2, r2, c2

; (N.H)
dp3_sat r1, r1_bx2, r3_bx2

; approximate (N.H)^16
; [(N.H)^2 - 0.75] * 4 == (N.H)^16
mad_x4_sat r1, r1, r1, c1

; (N.H)^32
mul_sat r1, r1, r1

; (N.H)^32 * specular color
mul_sat r1, r1, c3

; [(N.L) * base] + (N.H)^32
mad_sat r0, r2, r0, r1
```

```
ps.2.0

dcl t0
dcl t1
dcl t2
dcl t3

dcl_2d   s0
dcl_2d   s1

texld r0, t0, s0   ; base map
texld r1, t1, s1   ; bump map

dp3 r2.x, t2, t2   ; normalize L
rsq r2.x, r2.x
mul r2, t2, r2.x

dp3 r3.x, t3, t3   ; normalize H
rsq r3.x, r3.x
mul r3, t3, r3.x

mad r1, r1, c4.y, c4.x ; scale and bias N

dp3_sat r2, r1, r2 ; N.L
mul r2, r2, c2     ; N.L * diffuse_light_color
dp3_sat r1, r1, r3 ; (N.H)
pow r1, r1.x, c1.x ; (N.H)^k
mul r1, r1, c3     ; (N.H)^k *specular_light_color

mad_sat r0, r0, r0, r1 ; [(N.L) * base] + (N.H)^k

mov oC0, r0
```

Caps for Pixel Shader 2.x

D3DCAPS9	CineFX support
MaxPShaderInstructionsExecuted	1024
PS20Caps.NumInstructionSlots	512
PS20Caps.NumTemps	28
PS20Caps.StaticFlowControlDepth	0?
PS20Caps.DynamicFlowControlDepth	0
PS20Caps.Caps	ARBITRARYSWIZZLE
PS20Caps.Caps	GRADIENTINSTRUCTIONS
PS20Caps.Caps	PREDICATION
PS20Caps.Caps	NODEPENDENTREADLIMIT
PS20Caps.Caps	NOTEXINSTRUCTIONLIMIT

Pixel Shader 2.x Caps

- **Static flow control**
  - IF-ELSE-ENDIF, CALL / CALLNZ-RET, REP-ENDREP
  - As in vertex shader (less LOOP-ENDLOOP)
- **Gradient instructions and texture fetch**
  - DSX, DSY, TEXLDD
- **Predication**
  - As in vertex shader 2.x
- **Arbitrary swizzling**
  - Use them to pack registers

Pixel Shader 2.x

- 512 instruction-long shaders
  - Shaders can easily be > 96 instructions
  - Long shaders run slowly!
  - Great for prototyping
- No dependent read limit
  - Increases "ease of use"
- No texture instruction limit
  - Can fetch greater number of samples

Static flow control

- One way of controlling number of lights
- Usually 4 active lights is enough on any one triangle
- Benefits of single-pass lighting
  - Greater speed than multi-pass if vertex bound
    - Allows more complex vertex shaders
  - Better precision
    - fp32-bit shader precision vs. 8-bit FB blender
  - More flexible combination of lights
    - Combine lights in ways FB blender doesn't allow
- Shadows are tricky
  - Can use pre-baked occlusion information, either per-vertex or with textures (similar to lightmaps)

Single Pass Lighting?

- Sometimes it does make sense to collapse multiple vertex lights in one pass
  - Hence the fixed-function pipeline
- This works because the fixed function pipeline doesn't handle shadows
- With vertex shaders, one can do per-vertex shadowing
    - Paletted Lighting & Shadowing

Single Pass Lighting?

- Doing Shadowing typically brings us
  - from a Per-Object render loop
  - to a Per-Light render loop
- As each light's shadowing & calculation take more resources
  - Becomes harder to cram into one pass

Single-Pass Lighting ?

- **Detailed per-pixel lighting can typically be performed on DirectX8 cards in 1-3 passes per light**
  - **One pass for shadow creation**
  - **One pass for attenuation & shadow testing**
  - **One pass for lighting**
- **DirectX9 cards can do all the math for a light in one pass**
  - **Longer shaders mean attenuation, shadow testing & lighting can be performed in one pass**
  - **More lighting per pass means fewer, larger batches**
- **Does it make sense to all lighting in one pass?**

Single Pass Lighting?

- Shadow Volume Approaches require each shadowed light to be handled separately
  - Because they either use the stencil or dest alpha to store occlusion info
    - Which are single resources
- Shadow Maps allow multiple lights to be performed in one pass
  - Once the shadow maps are created, each light can perform shadow test & lighting calcs in one pass

Single Pass Lighting?

- Putting multiple lights in one pass leads to some issues
  - There is still a limit to the #of lights per pass that can be handled
    - Due to 16 samplers or 8 interpolators
    - Or overly long shaders
  - What to do when light count exceeded?
    - Fall back to multi-pass
    - Drop or merge 'unimportant' lights
      - Careful of popping and aliasing

Single Pass Lighting?

- **It makes sense to collapse a single light into a single render pass if possible**
  - Less vertex work
  - Less draw calls
  - Less bandwidth
    - Not really an issue if shader bound
  - Perf scales linearly with # of lights

Single Pass Lighting?

- It doesn't necessarily make sense to try to fit multiple shadowed lights in one pass
  - Shadow volumes
    - You can't really do this anyway
  - Shadow maps
    - Still a hard limit on the # that can be handled per pass
      - Multiple code paths
    - Non-linear perf falloff with more lights
    - As # of light 0/1/2/3 / material combinations go up, batch size goes down anyway
    - Probably not worth the hassle factor

Lighting Render Loop

- Per-Object Pass 0
  - Write Z
  - Ambient lighting
    - Light maps
  - Global Reflections
    - Cube-map reflections
  - Emissive
    - LED Panels, Light Sources
  - If outdoor, sunlight
    - With shadow if texture-based

Lighting Render Loop

- Per Light Pass L + 1
  - Create Shadow
    - Render Shadow Volume
    - Render to Shadow Map
- Per Light Pass L + 2
  - Test Shadow
    - Test vs Shadow Volume
    - Test vs Shadow Map
  - Perform Mask
    - Batman logo, Window Frame, etc.
  - Perform Attenuation
  - Store in Dest Alpha

Lighting Render Loop

- Per-Light Pass L + 3
  - Perform lighting
    - Diffuse Bump
    - Blinn or Phong
    - Colored Projection
      - Spotlight
      - Stained Glass Window
  - Blend
    - SrcColor * DestAlpha + DestColor * One

Lighting Render Loop

- Per-Object Pass 1
  - Optional Fog Pass
  - Fog Texture
    - Radial, Volume, etc.
  - Blend
    - SrcColor * One + InvSrcColor * DestColor

Lighting Render Loop

- On DirectX8 HW
  - Passes are as listed
- On DirectX9 HW
  - Passes L + 1 and L + 2 can be combined
  - Fog Pass can be collapsed into Pass 0
    - Due to not needing dest alpha for attenuation & mask
    - Perform Lighting passes with black fog
  - Pass L + 2 might have better lighting
    - Due to per-pixel calculation of H or R and real exponents for specular
    - Various more complex lighting models possible

Lighting Render Loop Summary

- Good solution for D3D8 / D3D9 scalability
- For shadowed scenes, use single light at a time
  - Easier than packing multiple lights into one pass
  - Scales linearly
- DirectX9 lets you do all lighting in one pass
  - But for shadows and similarity to DirectX8 path, use one pass per light

Greater number of instructions

- **Longer shader programs make multi-sample AA really "free"**
  - **Sweet spot for low-end DirectX9 cards is w/ 4x AA**
  - **640x480x4x is way faster than 1280x960x1x on a pixel shader-bound app**

- **Long shaders more practical with HLSL**

Predication

- Essentially a destination write mask
  - Analogous to use of CMP / CND
  - Computed per component
  - Can be swizzled and negated
- The perf win is mainly fewer temporaries compared to MAD-style
  - The p0 predication register is ‘free’
  - Doesn’t count towards temp register count
- Provides “lock-step” data-dependent execution
  - All pixels execute the same instructions

Pixel Shader Predication – 2x2 Shadow Testing

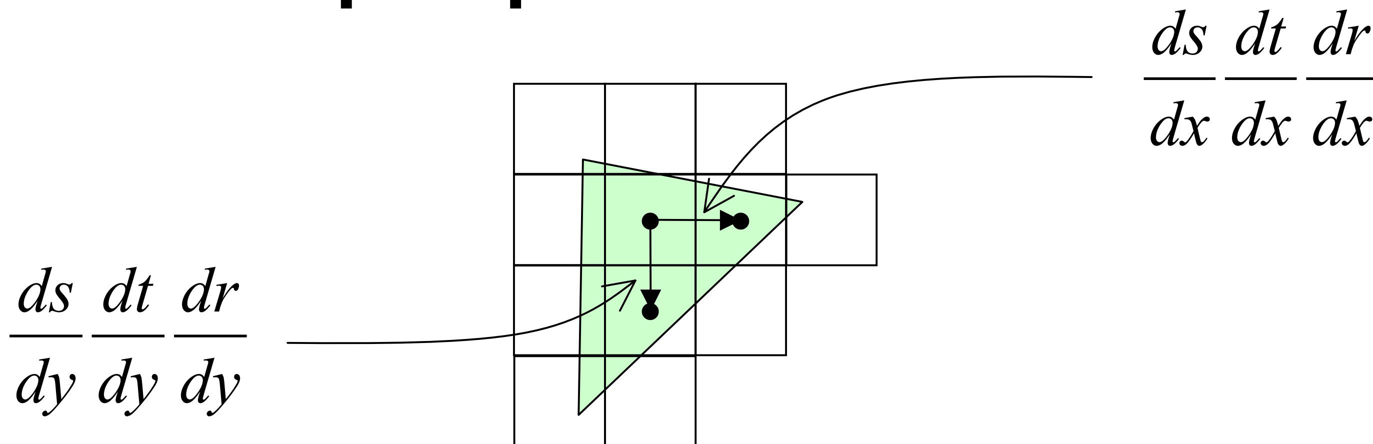
```
   def c0, 0.0f, 0.0f, 0.0f, 0.0f
   def c1, 1.0f, 1.0f, 1.0f, 1.0f
   . . .
   sub_sat  r0, r0, r1  // compare 4 values
   dp4      r0, r0, c1  // sum the result

   mov r2, c2           // ambient light
   setp_gt  p0, r0, c0  // set predication

p0 add r2, r2, c3       // conditionally add lighting
```

Caveats for Conditional Instructions

- Conditional instructions can easily cause visual discontinuities
- Useful for shaders that sometimes need a sharp edge
  - Or effects where the edge is faded out anyway
  - Like light attenuation
- A filtered texture fetch into a sharp gradient texture can give smoother results

Gradient Instructions

- **Useful for shader anti-aliasing**
- **DSX, DSY compute derivative of a vector wrt screen space position**



- **Use derivatives with TEXLDD…**

Texture fetch with gradients

- Gradients used to calculate texture LOD
  - Custom Mipmapping
  - Anisotropic filtering
- LOD clamping and constant biasing still apply
- Example
  - Calculate per-pixel vector for environment map
  - Find derivative of vector wrt X,Y
  - Fetch envmap using TEXLDD to bias mip level

Shader Texture LOD

- If you do a texture fetch in a pixel shader, LOD is calculated automatically
  - Based on a 2x2 pixel quad
- So, you only need the gradient instructions if you want a non-default LOD
- Or if you want to band-limit your shading function to avoid aliasing
  - Switch shading models or drop high-frequency terms

Arbitrary swizzling

- **Extremely useful**
  - For extracting components (r0.x = tcoord0.w, r0.y = tcoord1.w)
  - For replicating components (r0 = light.zzzz)

- **Pack more data per register**
  - Especially temporaries (r0.xy, r0.zw)
  - On some HW, fewer temporaries yields better performance

Pixel Shader Precision

- **2.0 Pixel Shaders support variable fp precision**
- **High Precision Values (at least fp24)**
  - **Texture coordinates**
  - **fp32 texture fetches**
  - **Temps derived from at least one high precision source**

Pixel Shader Precision

- **Low Precision Values (at least fp16)**
  - **Constants (careful with this, can only store ±2048.0 exactly)**
  - **Texture fetches from < fp32 textures**
  - **Iterated diffuse & specular**
  - **Temps derived from above**
  - **Any value declared 'half' or _pp**

Why use fp16 precision?

- SPEED
  - On some HW, using fewer registers leads to faster performance
  - On some HW, fp16 takes half the register space of fp32, so can be 2x faster
- That said, the first rule of optimization is : DON'T
- If your shaders are fast enough at full precision, great
  - But make sure you test on low-end DirectX9 cards, too
- Otherwise, here is how to optimize your shaders for precision

How to use fp16

- **In HLSL, use the `half` keyword instead of `float`**
- **Because the spec requires high-precision when mixing high & low precision, you may have to use extra casts or temporaries at times**
- **Assuming s0 maps to a 32bit fp texture :**
  - **Original :**

```
float3 a = tex2d( s0, tex0 )
```

  - **Optimized :**

```
half3 a = tex2d( s0, (half3)tex0 )
```

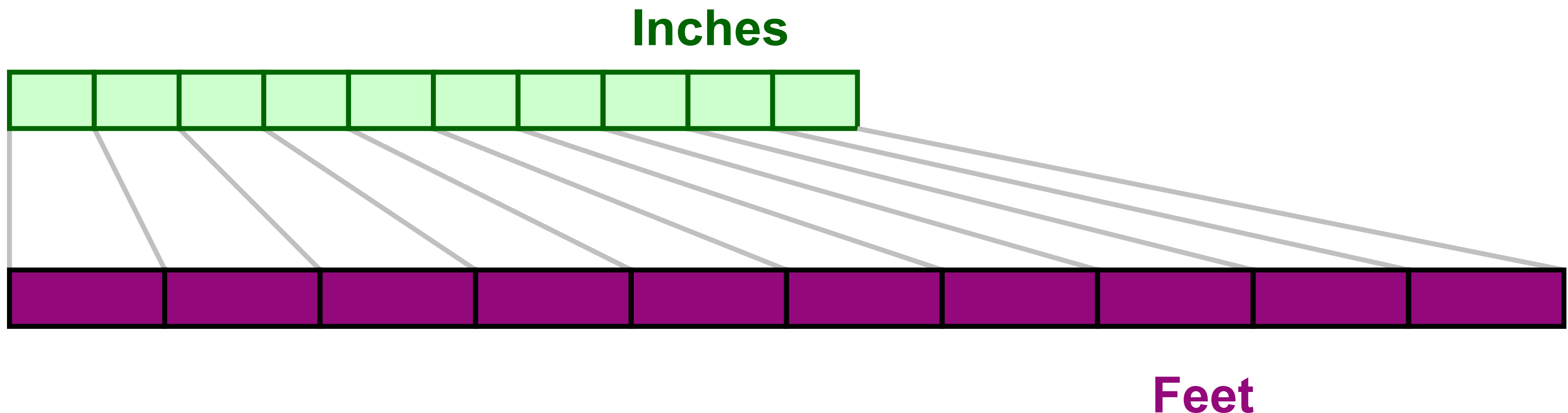
How to use fp16

- When using pixel shader 2.0+ assembly, you can ask for 16-bit precision via `_pp` modifier
  - Modifier applies to instructions
  - Modifier applies to inputs

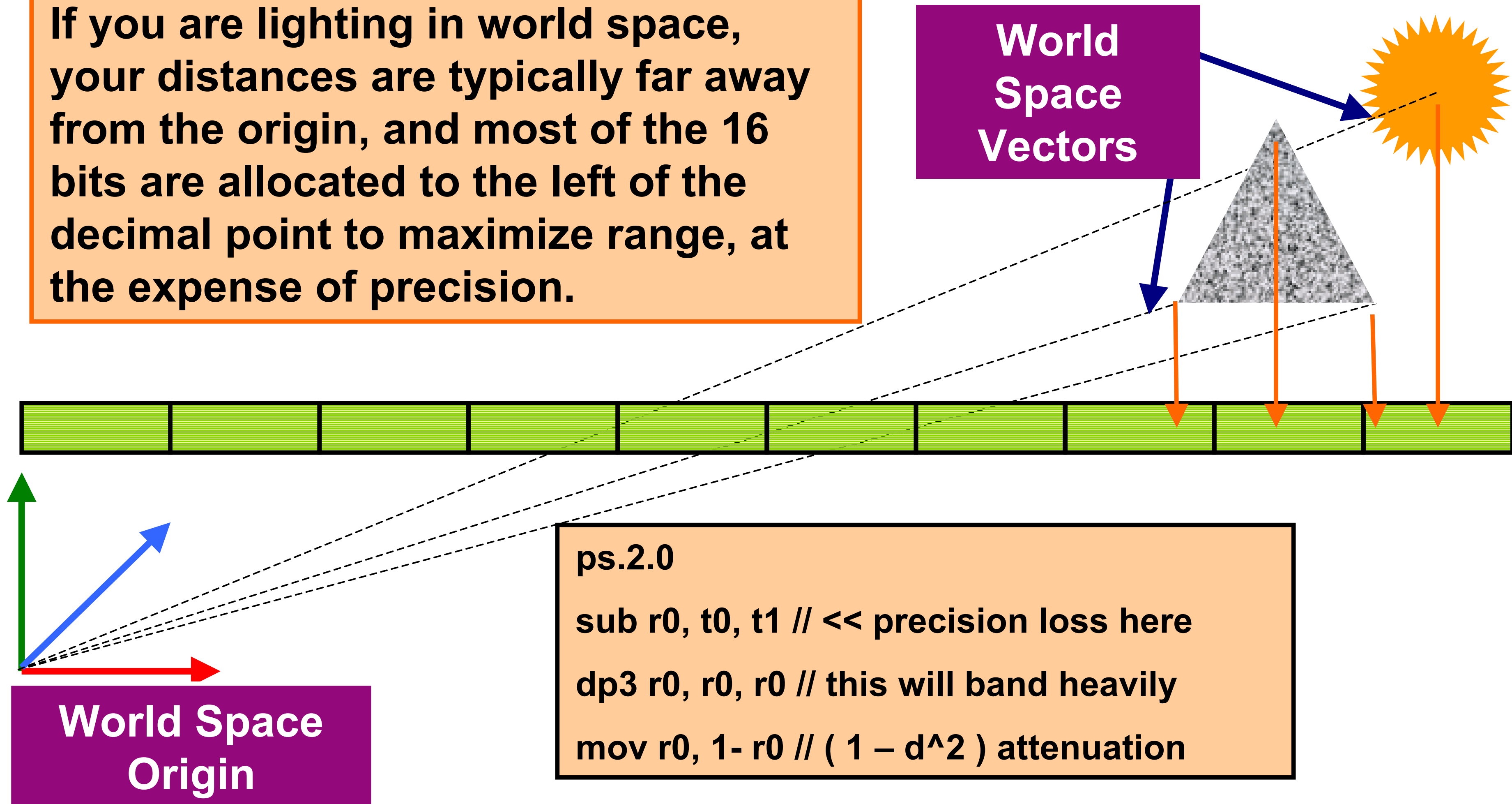
```
dcl_[pp] dest[.mask]

dcl_pp  t0.xyz // declare t0 as partial precision

sub_pp r0, r1, t0 // perform math at fp16
```

Precision Pitfalls

- An IEEE-like fp16 is 1 sign bit, 5 bits of exponent and 10 bits of mantissa
- Think of mantissa as tick marks on a ruler
- The Exponent is the length of ruler
  - +/-1024 ticks, no matter what, so .1% precision across whatever range you have



Precision Pitfalls



Precision Pitfalls

- Most precision problems we've seen at fp16 are due to developers doing world-space computations per-pixel
  - Classic example is per-pixel attenuation
  - Easy solution, change to light space in the vertex shader instead and pass down the result

Precision Pitfalls

- The bad news is that it's a bit inconvenient not to be able to write the entire shader at the fragment level
  - And get great speed
- The good news is that you really didn't want that anyway
  - Except for prototyping

Fully Fragment Shading?

- Doing the entire shading equation at the maximum computation frequency doesn't make sense
- Some of it is constant
  - Light colors, Material colors
- Much of it is linear
  - Positions
- The HW can interpolate linear components for free
- Why recompute something linear per-pixel?
  - Only if you run out of interpolators

Precision Pitfalls

- Viewspace positions
  - Subtract from the view point in the vertex shader – the vertex shader has 32-bit precision
- Tangent Space positions
  - For bump maps
  - Translate & Rotate in the vertex shader
- Light Space positions
  - For attenuation
  - Translate & Rotate in the vertex shader
  - Then scale by 1 / light range

Precision Pitfalls

- Avoid precision issues with Normalization Cubemaps
- Use a normalization cubemap to normalize vectors derived from position
  - Texture coordinates are fp24+
    - Unless marked `_pp`
- The resulting value from the cubemap is low-precision but derived at high precision

Texcoords and Precision

- **10 bits of mantissa is enough to exactly address a 1024-wide texture**
  - **With no filtering**
- **10 bits can do a 512-wide texture with 2 filtering levels**

- **So, large textures may require high precision to sample with good filtering**

Precision Summary

- If high-precision is fast enough, great
  - But remember the low-end DirectX9 cards!
- If you need more speed
  - Move constant things to the CPU
    - Except when that hurts batching too much
  - Move linear things to the vertex shader
  - Use texture lookups to replace math
  - Use half instead of float
  - Reduce shader version
    - Some HW runs 1.x faster than 2.0+

ps_3_0

- **Longer programs (512 minimum)**
- **Flow-control**
- **Access to `vFace` and `vPos.xy`**
- **Center vs Centroid sampling**
- **Arbitrary swizzling**

Summary

- **Vertex Shaders**
  - **vs_2_0 and extended bits**
  - **vs_3_0**
- **Pixel Shaders**
  - **ps_2_0 and extended bits**
  - **ps_3_0**