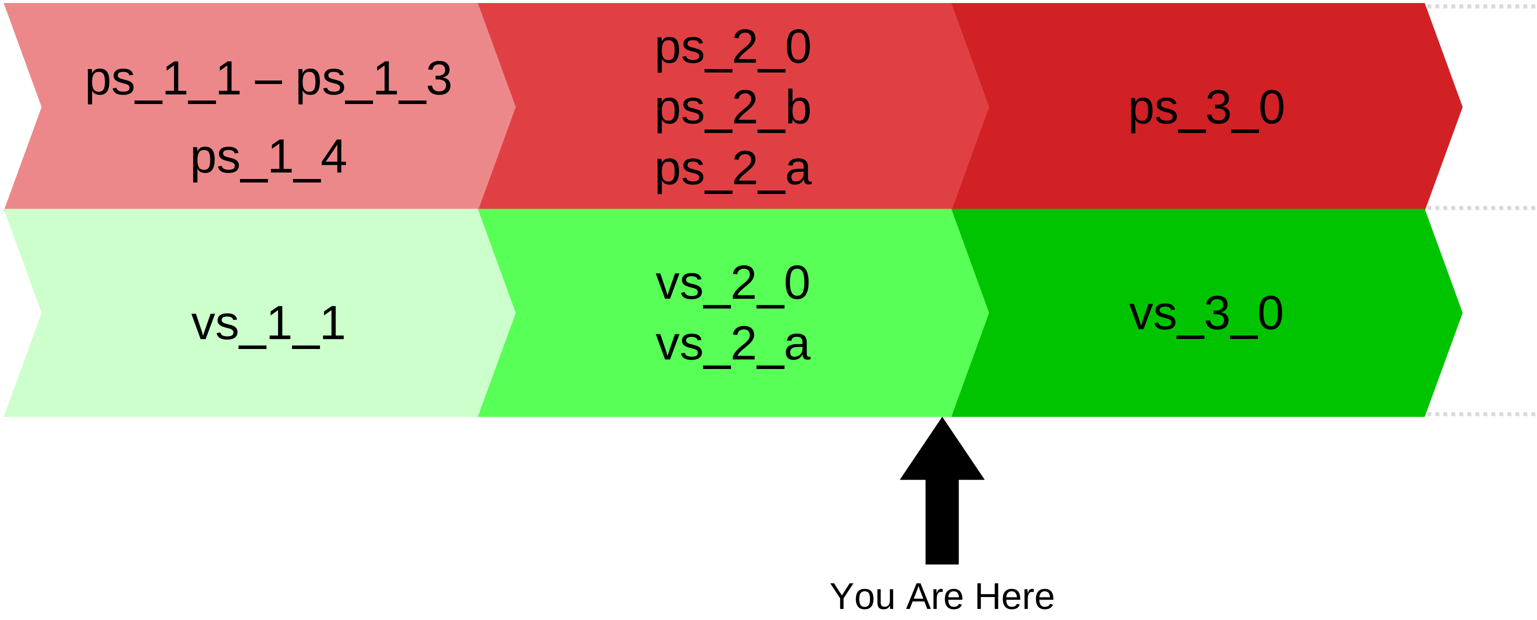
Direct3D Shader Models

Jason Mitchell
3D Application Research Group
ATI Research, Inc.

Outline

- Vertex Shaders
  - – Static and Dynamic Flow control
- Pixel Shaders
  - – ps_2_*x*
  - – ps_3_0

Shader Model Continuum



ps_1_1 – ps_1_3
ps_1_4

ps_2_0
ps_2_b
ps_2_a

ps_3_0

vs_1_1

vs_2_0
vs_2_a

vs_3_0

You Are Here

Tiered Experience

- PC developers have always had to scale the experience of their game across a range of platform capabilities
- Often, developers pick discrete tiers of experience
  - DirectX 7, DirectX 8, DirectX 9 is one example

- Shader-only games are in development
- Starting to see developers target the three levels of shader support as the distinguishing factor among the tiered experience for their users

Caps in addition to Shader Models

- In DirectX 9, devices can express their abilities via a base shader version plus some optional caps
- At this point, the only “base” shader versions beyond 1.*x* are the 2.0 and 3.0 shader versions
- Other differences are expressed via caps:
  - **`D3DCAPS9.PS20Caps`**
  - **`D3DCAPS9.VS20Caps`**
  - **`D3DCAPS9.MaxPixelShader30InstructionSlots`**
  - **`D3DCAPS9.MaxVertexShader30InstructionSlots`**
- This may seem messy, but it’s not that hard to manage given that you all are writing in HLSL and there are a finite number of device variations in the marketplace
- Can easily determine the level of support on the device by using the **`D3DXGet*ShaderProfile()`** routines

Compile Targets / Profiles

- Whenever a new family of devices ships, the HLSL compiler team may define a new target
- Each target is defined by a base shader version and a specific set of caps
- Existing compile targets are:
  - Vertex Shaders
    - vs_1_1
    - vs_2_0 and vs_2_a
    - vs_3_0
  - Pixel Shaders
    - ps_1_1, ps_1_2, ps_1_3 and ps_1_4
    - ps_2_0, ps_2_b and ps_2_a
    - ps_3_0

2.0 Vertex Shader HLSL Targets

- **vs_2_0**
  - 256 Instructions
  - 12 temporary registers
  - Static flow control (**StaticFlowControlDepth** = 1)
- **vs_2_a**
  - 256 Instructions
  - 13 temporary registers
  - Static flow control (**StaticFlowControlDepth** = 1)
  - Dynamic flow control (**DynamicFlowControlDepth** cap = 24)
  - Predication (**D3DVS20CAPS_PREDICATION**)

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 flow control instructions
  - **call, callnz, label, ret**
  - **If…else…endif**
  - **loop…endloop, endrep…rep**

vs_2_0 Registers

- Floating point registers
  - 16 Inputs ($\mathbf{v}_n$)
  - 12 Temps ($\mathbf{r}_n$)
  - At least 256 Constants ($\mathbf{c}_n$)
    - Cap'd: **`MaxVertexShaderConst`**
- Integer registers
  - 16 Loop counters ($\mathbf{i}_n$)
- Boolean scalar registers
  - 16 Control flow ($\mathbf{b}_n$)
- Address Registers
  - 4D vector: **`a0`**
  - Scalar loop counter (only valid in loop): **`aL`**

Vertex Shader Flow Control

- Goal is to reduce shader permutations, allowing apps to manage fewer shaders
  - The idea is to control the flow of execution through a relatively small number of key shaders
- Code size reduction is a goal as well, but code is also harder for compiler and driver to optimize
- Static Flow Control
  - Based solely on constants
  - Same code path for every vertex in a given draw call
- Dynamic Flow Control
  - Based on data read in from VB
  - Different vertices in a primitive can take different code paths

Static Flow Control Instructions

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

Conditionals

- Simple **`if…else…endif`** construction based on one of the 16 constant $\mathbf{b}_n$ registers
- May be nested
- Based on Boolean constants set through **`SetVertexShaderConstantB()`**

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

Static Conditional Example

```
COLOR_PAIR DoDirLight(float3 N, float3 V, int i)
{
    COLOR_PAIR Out;
    float3 L = mul((float3x3)matViewIT, -normalize(lights[i].vDir));
    float NdotL = dot(N, L);
    Out.Color = lights[i].vAmbient;
    Out.ColorSpec = 0;
    if(NdotL > 0.f)
    {
        //compute diffuse color
        Out.Color += NdotL * lights[i].vDiffuse;

        //add specular component
        if(bSpecular)
        {
            float3 H = normalize(L + V);   // half vector
            Out.ColorSpec = pow(max(0, dot(H, N)), fMaterialPower) * lights[i].vSpecular;
        }
    }
    return Out;
}
```

bSpecular is a boolean declared at global scope

The interesting part

Result

```
...
 if b0
   mul r0.xyz, v0.y, c11
   mad r0.xyz, c10, v0.x, r0
   mad r0.xyz, c12, v0.z, r0
   mad r0.xyz, c13, v0.w, r0
   dp3 r4.x, r0, r0
   rsq r0.w, r4.x
   mad r2.xyz, r0, -r0.w, r2
   nrm r0.xyz, r2
   dp3 r0.x, r0, r1
   max r1.w, r0.x, c23.x
   pow r0.w, r1.w, c21.x
   mul r1, r0.w, c5
  else
   mov r1, c23.x
  endif
...
```

Executes only if **`bSpecular`** is TRUE

Two kinds of loops

- Must be completely inside an **`if`** block, or completely outside of it
- **`loop aL,`** $\mathbf{i}_n$
  - $\mathbf{i}_n$**`.x`** - Iteration count (non-negative)
  - $\mathbf{i}_n$**`.y`** - Initial value of **`aL`** (non-negative)
  - $\mathbf{i}_n$**`.z`** - Increment for **`aL`** (can be negative)
  - **`aL`** can be used to index the constant store
  - No nesting in vs_2_0
- **`rep`** $\mathbf{i}_n$
  - $\mathbf{i}_n$ - Number of times to loop
  - No nesting

Loops from HLSL

- The D3DX HLSL compiler has some restrictions on the types of **`for`** loops which will result in asm flow-control instructions. Specifically, they must be of the following form in order to generate the desired asm instruction sequence:

```
for(i = 0; i < n; i++)
```

- This will result in an asm loop of the following form:

```
rep i0
  ...
endrep
```

- In the above asm, **`i0`** is an integer register specifying the number of times to execute the loop
- The loop counter, **`i0`**, is initialized before the **`rep`** instruction and incremented before the **`endrep`** instruction.

Static HLSL Loop

```
...
   Out.Color = vAmbientColor;             // Light computation

   for(int i = 0; i < iLightDirNum; i++)  // Directional Diffuse
   {
      float4 ColOut = DoDirLightDiffuseOnly(N, i+iLightDirIni);
      Out.Color += ColOut;
   }

   Out.Color *= vMaterialColor;           // Apply material color

   Out.Color = min(1, Out.Color);         // Saturate

...
```

Result

```
vs_2_0
def c58, 0, 9, 1, 0
dcl_position v0
dcl_normal v1
         ...
rep i0
  add r2.w, r0.w, c57.x
  mul r2.w, r2.w, c58.y
  mova a0.w, r2.w
  nrm r2.xyz, c2[a0.w]
  mul r3.xyz, -r2.y, c53
  mad r3.xyz, c52, -r2.x, r3
  mad r2.xyz, c54, -r2.z, r3
  dp3 r2.x, r0, r2
  slt r3.w, c58.x, r2.x
  mul r2, r2.x, c4[a0.w]
  mad r2, r3.w, r2, c3[a0.w]
  add r1, r1, r2
  add r0.w, r0.w, c58.z
endrep
mov r0, r1
mul r0, r0, c55
min oD0, r0, c58.z
```

Executes once for each directional diffuse light

Subroutines

- Can only call forward
- Can be called inside of a loop
  - **aL** is accessible inside that loop
- No nesting in vs_2_0 or vs_2_a
  - See **StaticFlowControlDepth** member of **D3DVSHADERCAPS2_0** for a given device
- Limited to 4 in vs_3_0

Subroutines

- Currently, the HLSL compiler inlines all function calls
- Does not generate **`call`** / **`ret`** instructions and likely won’t do so until a future release of DirectX
- Subroutines aren’t needed unless you find that you’re running out of shader instruction store

Dynamic Flow Control

- If **`D3DCAPS9.VS20Caps.DynamicFlowControlDepth > 0`**, dynamic flow control instructions are supported:
  - **`if_gt  if_lt  if_ge  if_le  if_eq  if_ne`**
  - **`break_gt break_lt break_ge break_le break_eq break_ne`**
  - **`break`**
- HLSL compiler has a set of heuristics about when it is better to emit an algebraic expansion, rather than use real dynamic flow control
  - Number of variables changed by the block
  - Number of instructions in the body of the block
  - Type of instructions inside the block
  - Whether the HLSL has texture or gradient instructions inside the block

Obvious Dynamic Early-Out Optimizations

- Zero skin weight(s)
  - Skip bone(s)
- Light attenuation to zero
  - Skip light computation
- Non-positive Lambertian term
  - Skip light computation
- Fully fogged pixel
  - Skip the rest of the pixel shader
- Many others like these…

Dynamic Conditional Example

```
COLOR_PAIR DoDirLight(float3 N, float3 V, int i)
{
   COLOR_PAIR Out;
   float3 L = mul((float3x3)matViewIT, -normalize(lights[i].vDir));
   float NdotL = dot(N, L);
   Out.Color = lights[i].vAmbient;
   Out.ColorSpec = 0;
   if(NdotL > 0.f)
   {
      //compute diffuse color
      Out.Color += NdotL * lights[i].vDiffuse;

      //add specular component
      if(bSpecular)
      {
         float3 H = normalize(L + V);   // half vector
         Out.ColorSpec = pow(max(0, dot(H,N)), fMaterialPower) * lights[i].vSpecular;
      }
   }
   return Out;
}
```

Dynamic condition which can be different at each vertex

The interesting part

Result

Executes only if N.L is positive

```
dp3 r2.w, r1, r2
  if_lt c23.x, r2.w
   if b0
    mul r0.xyz, v0.y, c11
    mad r0.xyz, c10, v0.x, r0
    mad r0.xyz, c12, v0.z, r0
    mad r0.xyz, c13, v0.w, r0
    dp3 r0.w, r0, r0
    rsq r0.w, r0.w
    mad r2.xyz, r0, -r0.w, r2
    nrm r0.xyz, r2
    dp3 r0.w, r0, r1
    max r1.w, r0.w, c23.x
    pow r0.w, r1.w, c21.x
    mul r1, r0.w, c5
   else
    mov r1, c23.x
   endif
   mov r0, c3
   mad r0, r2.w, c4, r0
  else
   mov r1, c23.x
   mov r0, c3
  endif
```

Hardware Parallelism

- This is **not** a CPU
- There are many shader units executing in parallel
  - These are generally in lock-step, executing the same instruction on different pixels/vertices at the same time
  - Dynamic flow control can cause inefficiencies in such an architecture since different pixels/vertices can take different code paths
- Dynamic branching is not always a performance win
- For an **`if…else`**, there will be cases where evaluating both the blocks is faster than using dynamic flow control, particularly if there is a small number of instructions in each block
- Depending on the mix of vertices, the worst case performance can be worse than executing the straight line code without any branching at all

Predication

- One way around the parallelism issue
- Effectively a method of conditionally executing code on a per-component basis, or you can think of it as a programmable write mask
- Optionally supported on {v|p}s_2_0 by setting **`D3D{V|P}S20CAPS_PREDICATION`** bit
- For short code sequences, it is faster than executing a branch, as mentioned earlier
- Can use fewer temporaries than **`if…else`**
- Keeps shader units in lock-step but gives behavior of data-dependent execution
  - All shader units execute the same instructions

`if…else…endif` vs. Predication

- You’ll find that the HLSL compiler does not generate predication instructions
- This is because it is easy for a hardware vendor to map **`if…else…endif`** code to hardware predication, but not the other way around

vs_3_0

- Basically vs_2_0 with all of the caps
- No fine-grained caps like in vs_2_0. Only one:
  - **`MaxVertexShader30InstructionSlots`** (512 to 32768)
- More temps (32)
- Indexable input and output registers
- Access to textures!
  - **`texldl`**
  - No dependent read limit

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 **`dcl_positiont`** semantic

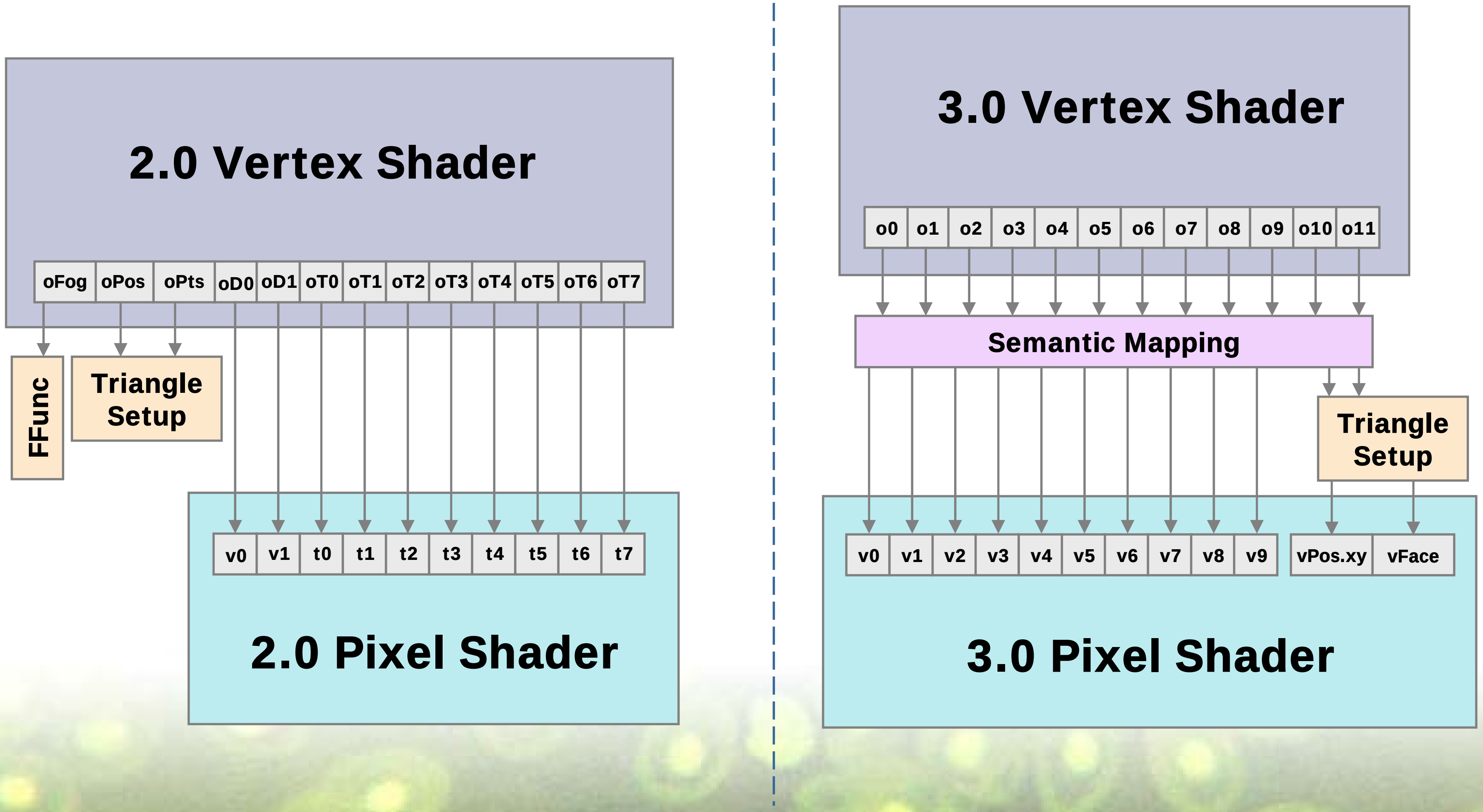
vs_3_0 Semantic Declaration

- Note that multiple semantics can go into a single output register

```
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
```

- HLSL currently doesn’t support this multi-packing

Connecting VS to PS



Vertex Texturing in vs_3_0

- With vs_3_0, vertex shaders can sample textures
- Many applications
  - Displacement mapping
  - Large off-chip matrix palette
  - Generally cycling processed data (pixels) back into the vertex engine

Vertex Texturing Details

- With the **`texldl`** instruction, a vs_3_0 shader can access memory
- The LOD must be computed by the shader
- Four texture sampler stages
  - **`D3DVERTEXTEXTURESAMPLER0..3`**
- Use **`CheckDeviceFormat()`** with **`D3DUSAGE_QUERY_VERTEXTEXTURE`** to determine format support
- Look at **`VertexTextureFilterCaps`** to determine filtering support (no Aniso)

2.0 Pixel Shader HLSL Targets

- **ps_2_0**
  - 64 ALU & 32 texture instructions
  - 12 temps
  - 4 levels of dependency
- **ps_2_b**
  - 512 instructions (any mix of ALU and texture, D3DPS20CAPS_NOTEXINSTRUCTIONLIMIT)
  - 32 temps
  - 4 levels of dependency
- **ps_2_a**
  - 512 instructions (any mix of ALU and texture, D3DPS20CAPS_NOTEXINSTRUCTIONLIMIT)
  - 22 temps
  - No limit on levels of dependency (D3DPS20CAPS_NODEPENDENTREADLIMIT)
  - Arbitrary swizzles (D3DPS20CAPS_ARBITRARYSWIZZLE)
  - Predication (D3DPS20CAPS_PREDICATION)
  - Most static flow control
    - **`if…else…endif, call/callnz…ret, rep…endrep`**
    - HLSL doesn’t generate static flow control for ps_2_a
  - Gradient instructions (D3DPS20CAPS_GRADIENTINSTRUCTIONS)

2.0 Pixel Shader HLSL Targets

	ps_2_0	ps_2_b	ps_2_a
Instructions	64 + 32	512	512
Temporary Registers	12	32	22
Levels of dependency	4	4	Unlimited
Arbitrary swizzles	✘	✘	✓
Predication	✘	✘	✓
Static flow control	✘	✘	✓
Gradient Instructions	✘	✘	✓

ps_3_0

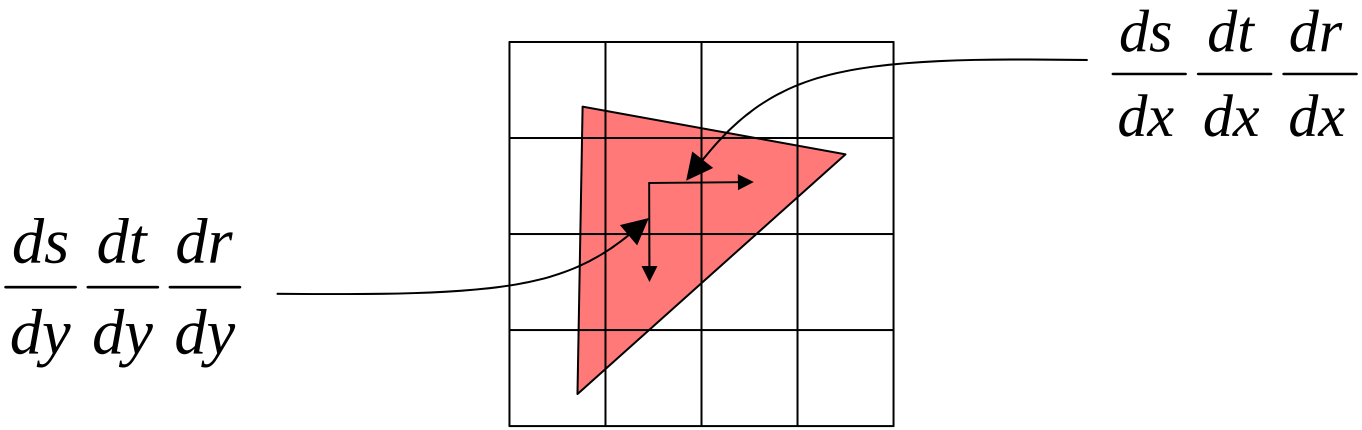
- Longer programs (512 minimum)
- Dynamic flow-control
- Access to **vFace** and **vPos.xy**
- Centroid interpolation

Aliasing due to Conditionals

- Conditionals in pixel shaders can cause aliasing!
- You want to avoid doing a hard conditional with a quantity that is key to determining your final color
  - Do a procedural smoothstep, use a pre-filtered texture for the function you're expressing or bandlimit the expression
  - This is a fine art. Huge amounts of effort go into this in the offline world where procedural RenderMan shaders are a staple
- On some compile targets, you can find out the screen space derivatives of quantities in the shader for this purpose…

Shader Antialiasing

- Computing derivatives (actually *differences*) of shader quantities with respect to screen *x*, *y* coordinates is fundamental to procedural shading
- LOD is calculated automatically based on a 2×2 pixel quad, so you don't generally have to think about it, even for dependent texture fetches
- The HLSL **`dsx()`**, **`dsy()`** derivative intrinsic functions, available when compiling for ps_2_a and ps_3_0, can compute these derivatives



$$\frac{ds}{dx} \quad \frac{dt}{dx} \quad \frac{dr}{dx}$$

$$\frac{ds}{dy} \quad \frac{dt}{dy} \quad \frac{dr}{dy}$$

- Use these derivatives to antialias your procedural shaders or
- Pass results of **`dsx()`** and **`dsy()`** to **`tex`*`n`*`D(s, t, ddx, ddy)`**

Derivatives and Dynamic Flow Control

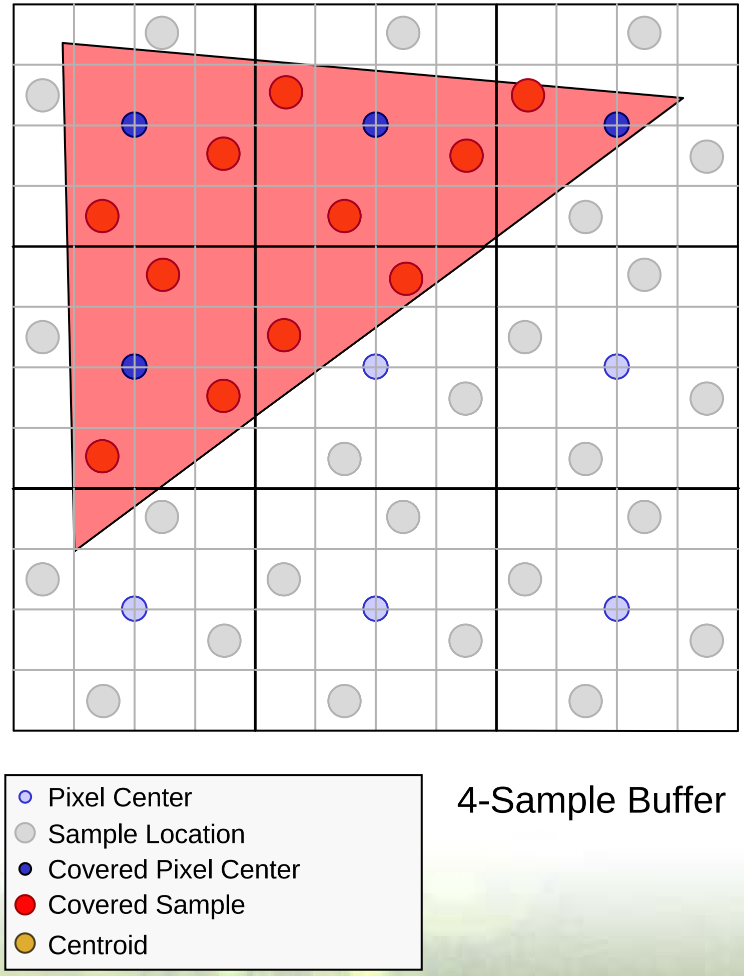
- The result of a gradient calculation on a computed value (i.e. not an input such as a texture coordinate) inside dynamic flow control is ambiguous when adjacent pixels may go down separate paths
- Hence, nothing that requires a derivative of a computed value may exist inside of dynamic flow control
  - This includes most texture fetches, **`dsx()`** and **`dsy()`**
  - **`texldl`** and **`texldd`** work since you have to compute the LOD or derivatives outside of the dynamic flow control
- RenderMan has similar restrictions

vFace & vPos

- **vFace** – Scalar facingness register
  - – Positive if front facing, negative if back facing
  - – Can do things like two-sided lighting
  - – Appears as either +1 or -1 in HLSL
- **vPos** – Screen space position
  - – *x*, *y* contain screen space position
  - – *z*, *w* are undefined

Centroid Interpolation

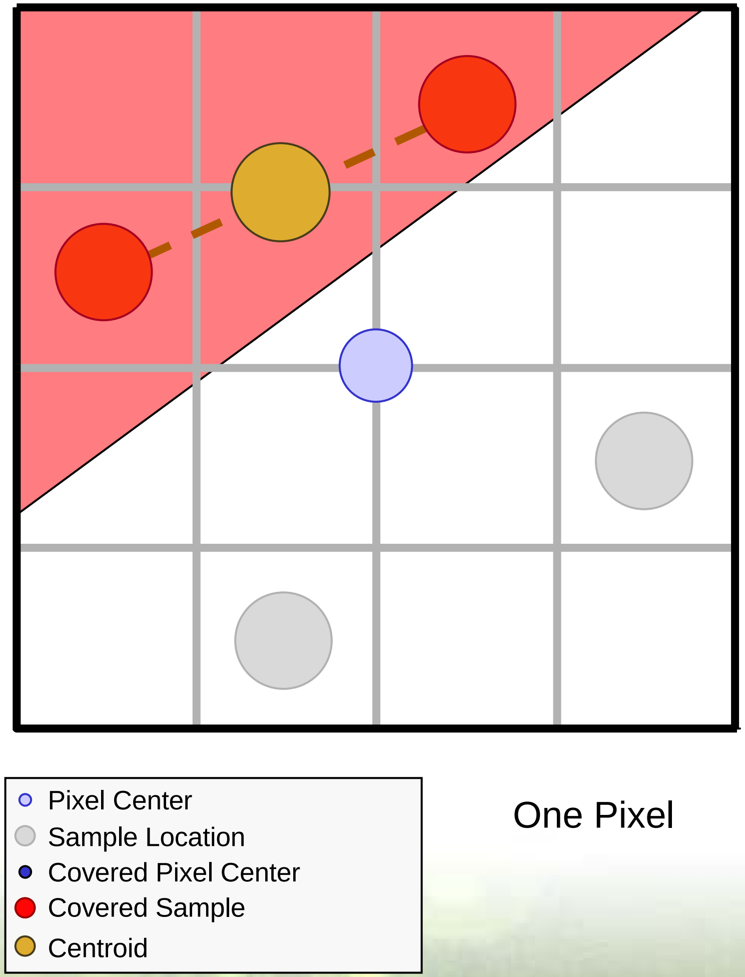
- When multisample antialiasing, some pixels are partially covered
- The pixel shader is run once per pixel
- Interpolated quantities are generally evaluated at the center of the pixel
- However, the center of the pixel may lie outside of the primitive
- Depending on the meaning of the interpolator, this may be bad, due to what is effectively extrapolation beyond the edge of the primitive
- Centroid interpolation evaluates the interpolated quantity at the centroid of the covered samples
- Available in ps_2_0 in DX9.0c



4-Sample Buffer

Centroid Interpolation

- When multisample antialiasing, some pixels are partially covered
- The pixel shader is run once per pixel
- Interpolated quantities are generally evaluated at the center of the pixel
- However, the center of the pixel may lie outside of the primitive
- Depending on the meaning of the interpolator, this may be bad, due to what is effectively extrapolation beyond the edge of the primitive
- Centroid interpolation evaluates the interpolated quantity at the centroid of the covered samples
- Available in ps_2_0 in DX9.0c



Centroid Usage

- When?
  - – Light map paging
  - – Interpolating light vectors
  - – Interpolating basis vectors
    - Normal, tangent, binormal
- How?
  - – Colors already use centroid interpolation automatically
  - – In asm, tag texture coordinate declarations with **_centroid**
  - – In HLSL, tag appropriate pixel shader input semantics:

```
float4 main(float4 vTangent : TEXCOORD0_centroid){}
```

Summary

- Vertex Shaders
  - – Static and Dynamic Flow control
- Pixel Shaders
  - – ps_2_*x*
  - – ps_3_0