Game Developers Conference

3.0 Shaders

Jason Mitchell

ATI Research

Outline

- Vertex Shaders
  - Vertex Textures
  - Flow control
- Pixel Shaders
  - Flow control
  - Optimization
    - Shadow Mapping
  - New functionality
    - `vPos` for interleaved sampling

3.0 Vertex Shaders

- Texture lookups
- Loop indexable inputs (v_n) and outputs (o_n)
  - Not just constants
- More temps (32)
- Longer programs
  - At least 512 instructions. See `MaxVertexShader30InstructionSlots` for exact number on a given chip
- Same flow control as devices which support the vs_2_a compile target

Vertex Texturing

- 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

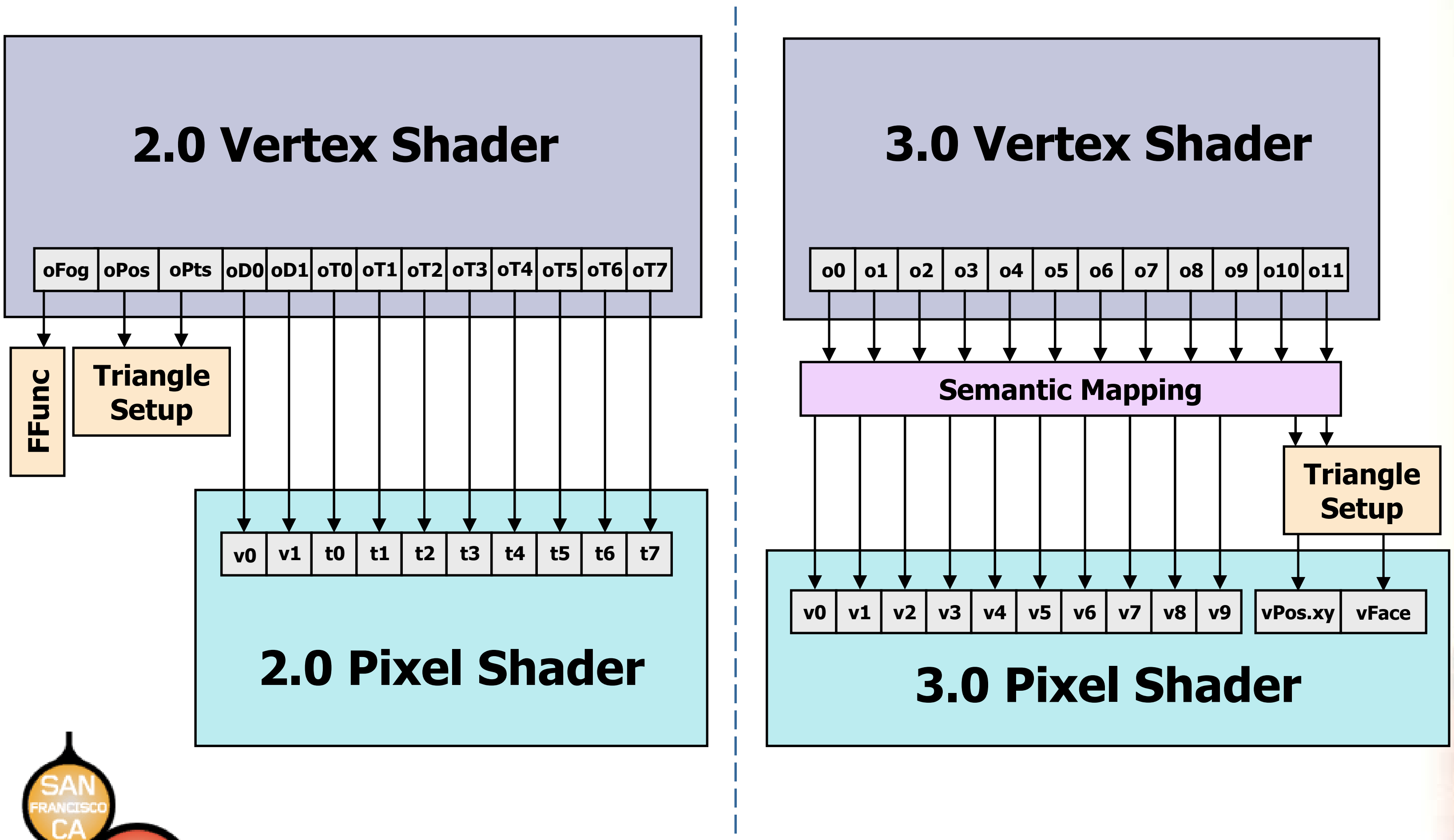
vs_3_0 Outputs

- 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 4-component output with the positiont semantic

Semantic Linkage

- Must use 3.0 vertex and pixel shaders together
- Input declarations take the usage names, and multiple usages are permitted for components of a given register

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
...
```

Dynamic Flow Control

- The 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
- Blindly changing compile targets can kill your performance, especially if you nest ifs

Hardware Parallelism

- There are many shader units executing in parallel
- Dynamic flow control can cause inefficiencies since different pixels/vertices can take different code paths
- Hardware will compute the right results, but you will not always see the intended performance gain
- 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 or pixels, the worst case performance can be worse than executing straight line code without any branching at all

Caveat emptor

Pixel Shaders

- Semantic linkage with vertex shader
  - Similar to vertex declarations
  - Generic $\mathtt{v}_n$ registers at asm level like vertex shader (all fp)
- Dynamic flow control
  - *caveat emptor*
- Longer programs
  - At least 512 (cap'd `MaxPixelShader30InstructionSlots`)
- More registers
  - Constants (224) and temps (32)
- Indexable input registers (but not constants)
- `tex*Dlod` (`texldl` at asm level)
  - Specify LOD (not bias) directly in texture load instruction
- New registers
  - `vFace` – Scalar face register
  - `vPos` - Screen (x, y) position register
  - aL – Loop counter

Input Registers

- Bank of 10 floating point registers
- Indexable

vFace

- Scalar register whose sign indicates the facing-ness of the triangle
  - Positive for front facing
  - Negative for back facing
- Can be interesting for things like two-sided lighting
- In future shader models, will contain primitive area

Pixel Shader Loop Register (`aL`)

- Incremented by `loop...endloop` block
- Can be used to index into interpolator registers only

Looping and HLSL

- Most of the time, this is a convenience to the developer and will actually be unrolled
- Dynamic number of iterations
  - Make it obvious to the compiler that there is an upper limit to the number of iterations that may dynamically occur
- HLSL constructs which cause unrolling of dynamic (not static) loops
  - Anything that needs a gradient (i.e. tex2D)
  - Indexing a local array, because these are not actually indexable in the virtual shader machine
  - Can index input iterators
- There is no `break` keyword in HLSL
  - Can be generated by the compiler in the asm based upon condition in while
  - Will show this in a later example

Known bounds on iteration

```
float4 ps_main( float4 inTexCoord : TEXCOORD0,
                float3 inOffset   : TEXCOORD1 ) : COLOR0
{
   float4 fH = 0;

   // Sample iteration map to determine how much to iterate
   int nNumSamples = (int)(tex2D( sAMap, inTexCoord ).r * 255.0) % 15;

   float2 dx = ddx( inTexCoord );
   float2 dy = ddy( inTexCoord );

   for ( int nIndex = 0; nIndex < nNumSamples; nIndex++ )
   {
      float2 texOffset = inTexCoord + inOffset * nIndex;
      fH += tex2Dgrad( sBMap, texOffset, dx, dy ).w;
   }
   return fH;
}
```

Speeds up compilation

Resulting Assembly

```
ps_3_0
    def c0, 255, 0, 1, 0
    def c1, 15, -15, 0, 0
    defi i0, 15, 0, 0, 0
    dcl_texcoord v0.xy
    dcl_texcoord1 v1.xy
    dcl_2d s0
    dcl_2d s1
…
    dsx r3.xy, v0
    dsy r4.xy, v0
    mov r1, c0.y
    mov r0.w, c0.y
    rep i0
      break_ge r0.w, r0.z
      mov r0.xy, v0
      mad r0.xy, v1, r0.w, r0
      texldd r2, r0, s0, r3, r4
      add r0.w, r0.w, c0.z
      add r1, r1, r2.w
    endrep
    mov oC0, r1
```

Returning

- If you want to return inside of an if…else it must be symmetric

Symmetric returns

```
edge = tex2D(EdgeSampler, oTex0).r;

if(edge > 0)
{
    return tex2Dlod(BaseSampler, oTex0);

}
else
{
    return 0;
}
```

```
texld r0, v0, s1
cmp r0.w, -r0.x, c0.x, c0.y
if_ne r0.w, -r0.w
  texldl oC0, v0, s0
else
  mov oC0, c0.x
endif
```

`vPos`

- `vPos.xy` contains screen-space position (`z` and `w` are undefined)
- Useful for screen-space operations such as interleaved sampling (see [Keller01])

Interleaved Sampling

- Do slightly different operations at neighboring pixels in screen space
- Two examples shown here:
  1. Volumetric Light shafts
     - Tweak position used in volume rendering
  2. Shadow filtering
     - Vary filter kernel layout as a function of screen position

Light Shafts with Interleaved Sampling

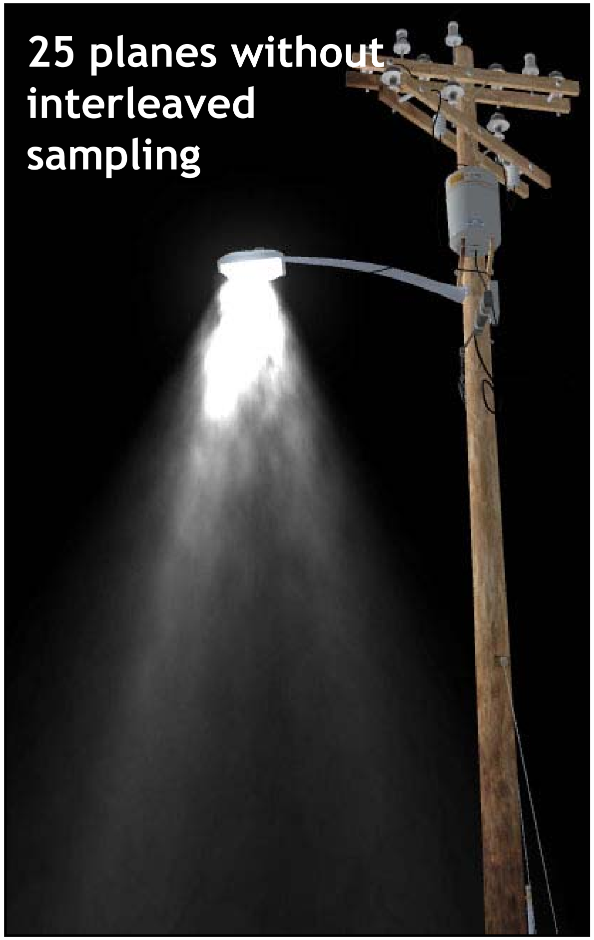
```
struct PsInput
{   float4 vWorldPos[4]      : TEXCOORD0;
    float4 vClipPos          : TEXCOORD4;
    float2 vScreenPos        : VPOS;
};

float4 main (PsInput i) : COLOR {
…

    // Based on the screen (x,y), determine whether the pixel is even or odd
    int2 vEvenOdd = (int) floor(fmod((i.vScreenPos.xy + 0.5), 2.0));

    int  iIndex = abs(3 * vEvenOdd.x - 2 * vEvenOdd.y);

    // Calculate the projective texture coordinate for the selected plane
    float4 vTexProj = mul(i.vWorldPos[iIndex], mLightViewProjBias);
```

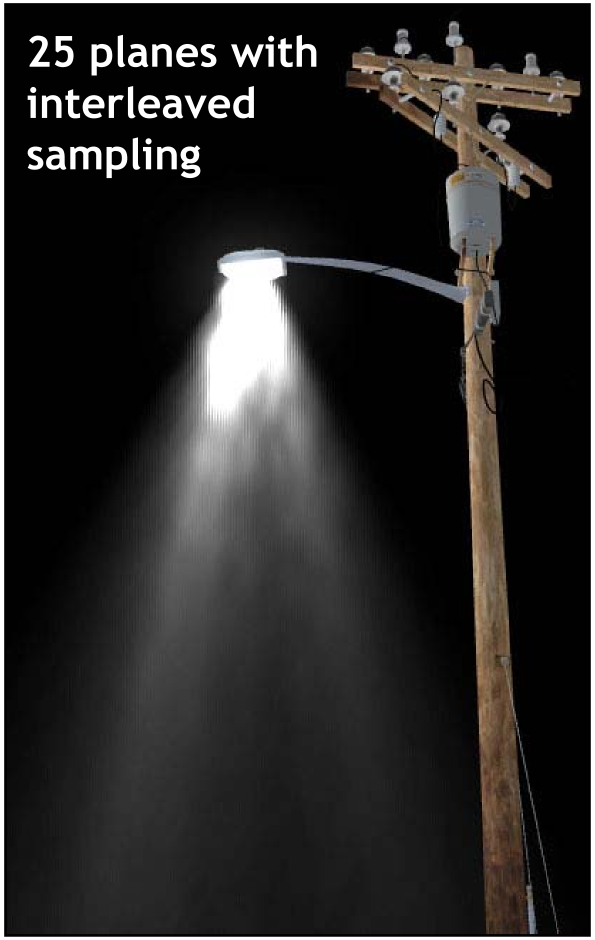
0	2	0	2
3	1	3	1
0	2	0	2
3	1	3	1

…Sample cookie, shadow and noise maps using tweaked coordinates
Compute attenuation based on tweaked position…

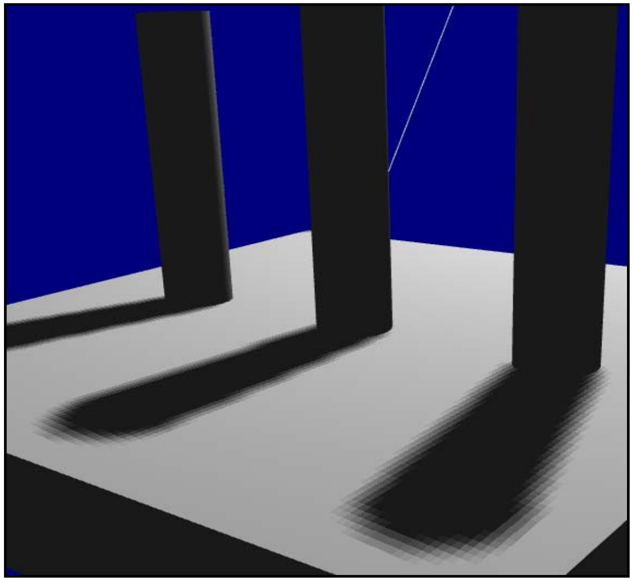
```
    // Final color output
    float fIntensity = fCompositeNoise * cCookie.rgb * fAtten * fScale;
    o.rgb = fIntensity;
    o.a = saturate(dot(o.rgb, float3(1.0f, 1.0f, 1.0f)));

    return o;
}
```

Light Shafts with Interleaved Sampling

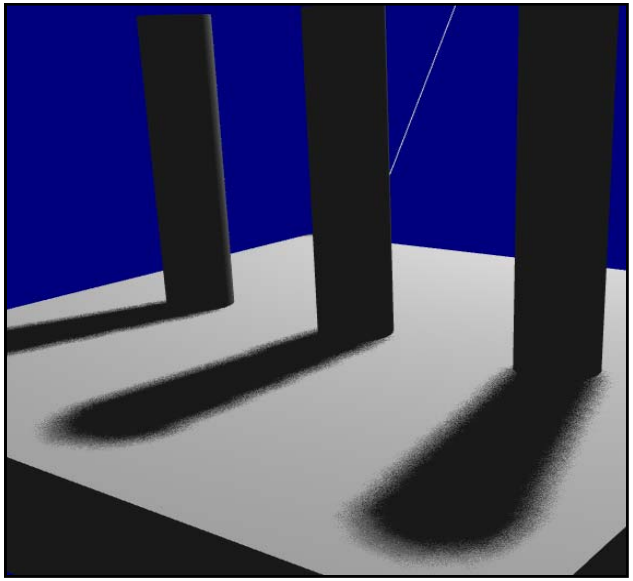




Spatially-varying PCF Offsets



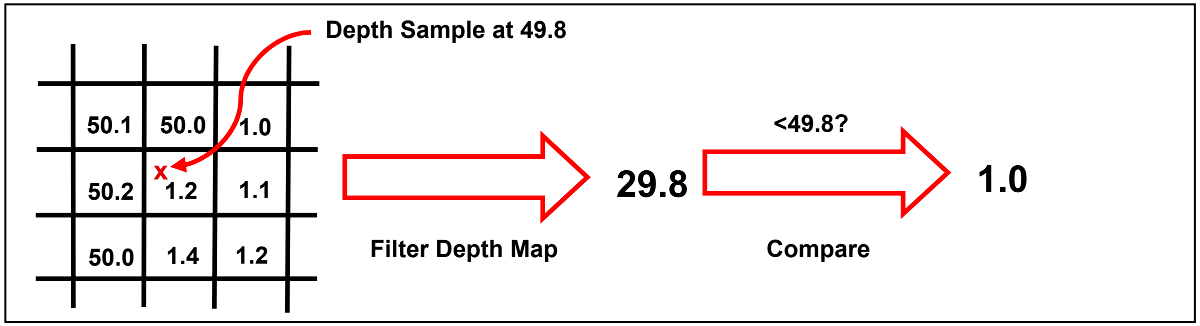
4×4 (16-tap) PCF



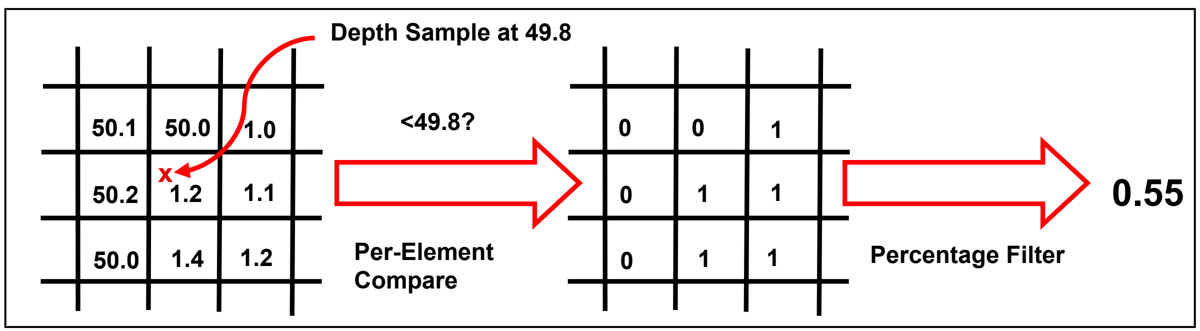
12-tap Spatially Varying PCF with Irregular sampling

- Grid-based PCF kernel needs to be fairly large to eliminate aliasing
  - Particularly in cases with small detail popping in and out of the underlying hard shadow.
- Irregular sampling allows us to get away with fewer samples
  - Error is still present, only the error is “unstructured” and thus less noticeable

Percentage Closer Filtering

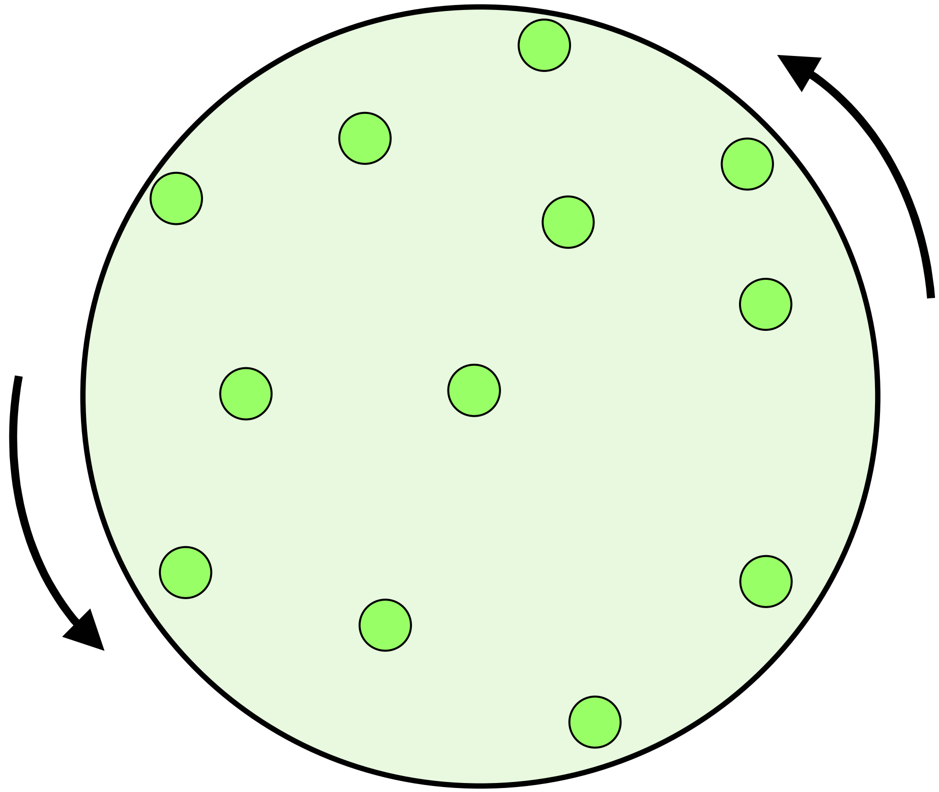


Standard filtering: Filter depth first, then use value for shadow map comparison.



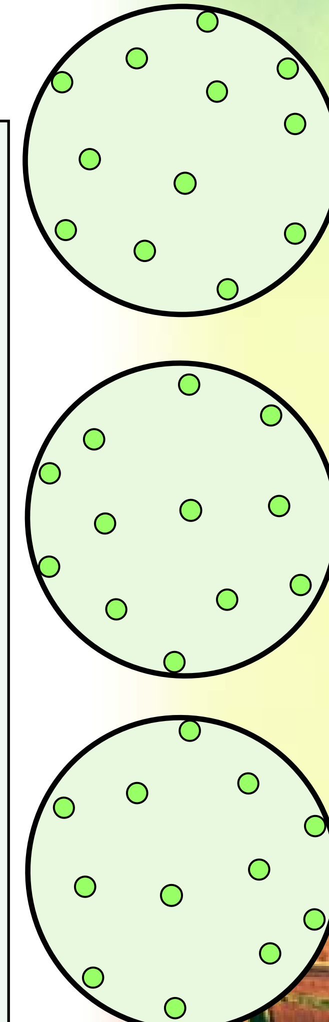
Percentage Closer Filtering: Perform shadow map comparison for each kernel elements first, then filter results!

Irregular Filter Kernel



Spatially-Varying Rotation

```
// Look up rotation for this pixel
float2 rot = BX2( tex2Dlod(RotSampler,
                  float4(vPos.xy * g_vTexelOffset.xy, 0, 0) ));

for(int i=0; i<12; i++) // Loop over taps
{
   // Rotate tap for this pixel location
   rotOff.x =  rot.r * quadOff[i].x + rot.g * quadOff[i].y;
   rotOff.y = -rot.g * quadOff[i].x + rot.r * quadOff[i].y;
   offsetInTexels = g_fSampRadius * rotOff;

   // Sample the shadow map
   float shadowMapVal = tex2Dlod(ShadowSampler,
   float4(projCoords.xy + (g_vTexOff.xy * offInTexels.xy), 0, 0));
   // Determine whether tap is in light
   inLight = ( dist < shadowMapVal );
   // Normalize
   percentInLight += inLight;
}
```



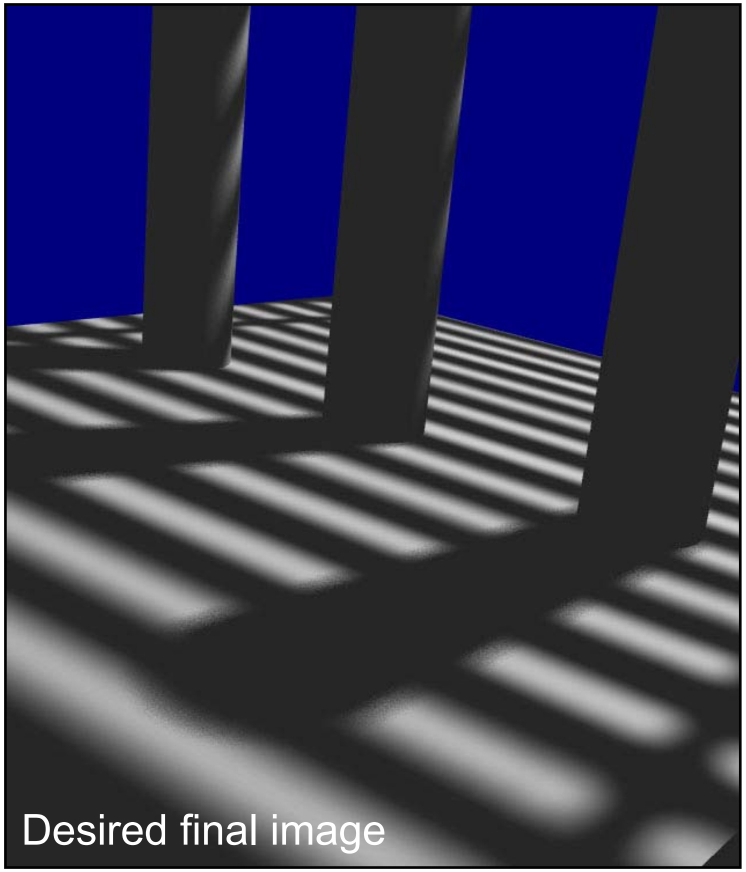
Obvious Early-Out Optimizations

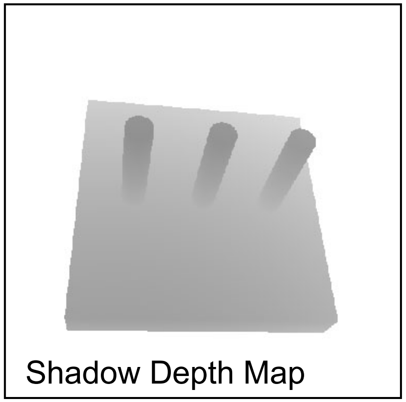
- Zero skin weight(s)
  - Skip bone(s)
- Light attenuation to zero
  - Skip light computation
- Non-positive Lambertian term
  - Skip light computation
- Fully fogged pixel
  - Skip the rest of the pixel shader
- Shadow Filtering
  - Only run costly filter in possible penumbra regions
- Many others like these…

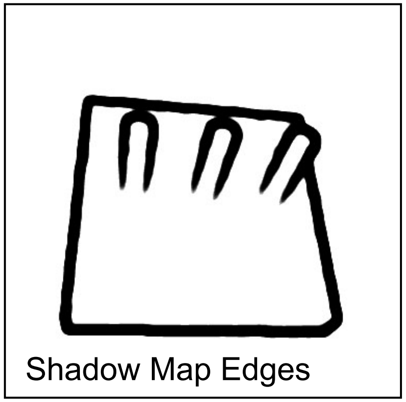
Shadow Filtering with ps_3_0

- Only do expensive filtering in areas likely to be penumbra regions
  - Dynamic flow control in pixel shader
- Can mask with a variety of values (no light or full light means no penumbra!)
  - $N \cdot L$
  - Projective Cookie texture (aka Gobo)
  - Edge-filtered shadow map

Simple example scene



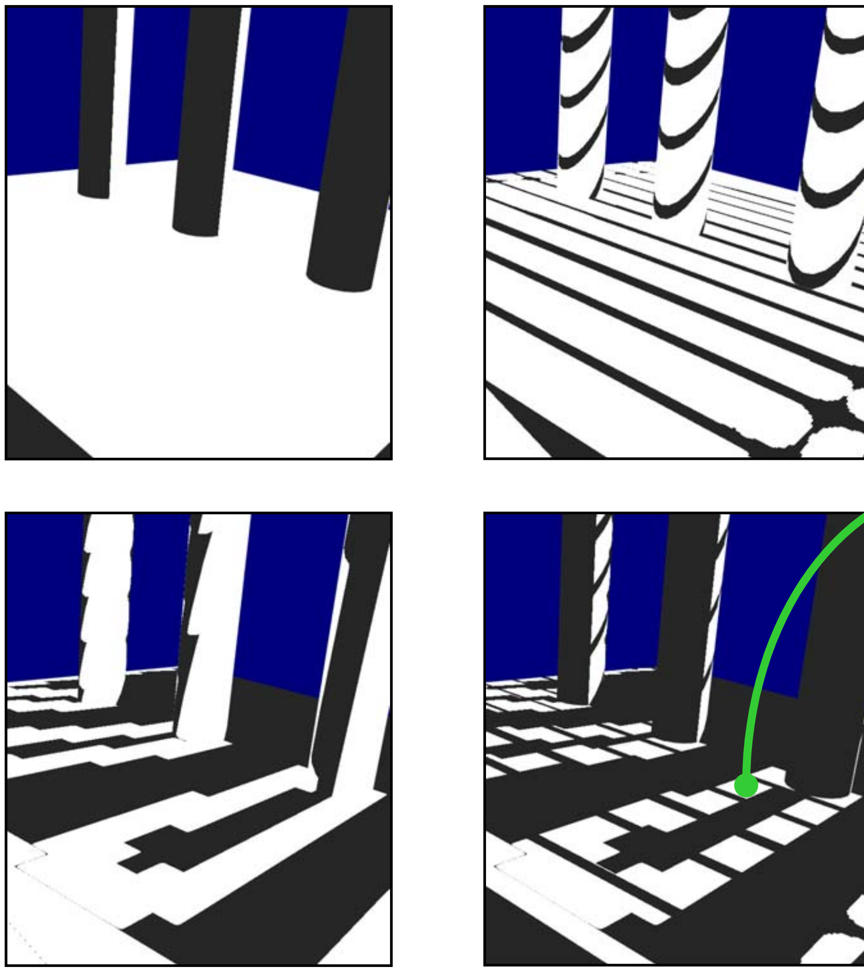




Mask off expensive filtering

N·L < 0

Gobo == 0



Only the white pixels execute the expensive path

Shadow Edge Filter

Union of all three masks

HLSL Shader With Early-Outs

...Compute projective coordinates and N.L...

```
if (dot(lightVal, float3(1,1,1)) == 0 ) {
   return 0;
}
else
{
```

...Sample edge map...

```
   if (edgeVal == 0) //compute hard shadows if we're not near an edge
   {
      shadowMapVal = tex2Dlod(ShadowSampler, projCoords );
      inLight = ( dist < shadowMapVal );
      percentInLight = dot(inLight, 0.25f );
      return (percentInLight * lightVal);
   }
   else
   {
      randRot = BX2( tex2Dlod(RandRotSampler, float4(vPos * g_vFullTexelOffset,0,0) ));
      for (int i=0; i<12; i++)
      {
```

...Do each expensive shadow sample...

```
      }
      return (percentInLight * lightVal);
   }
}
```

Resulting Assembly

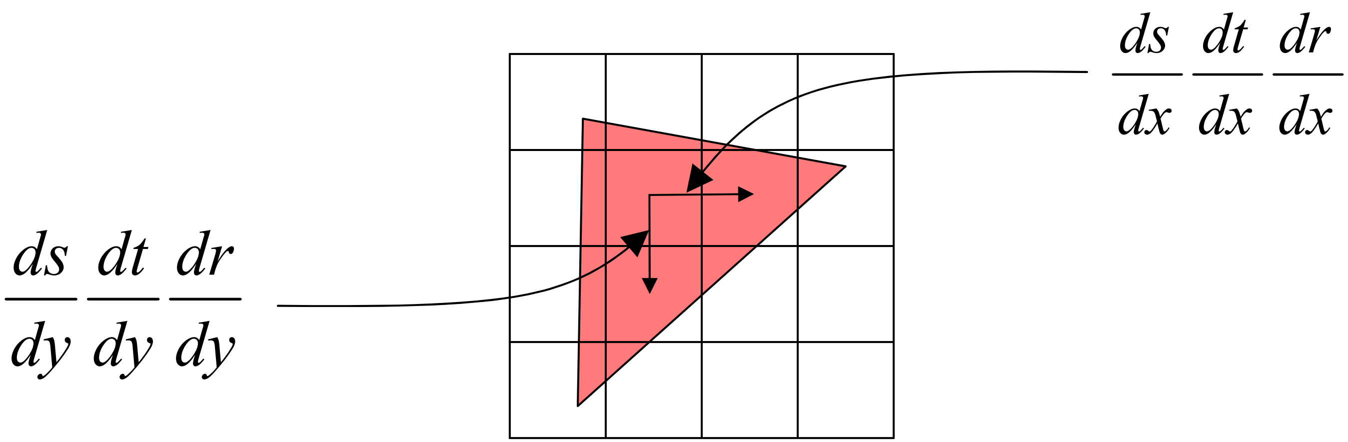
```
...
mul r0, r0, r1.z
dp3 r1.z, r0, c5.w
cmp r1.z, -r1_abs.z, c5.w, c5.z
if_ne r1.z, -r1.z
  mov oC0, c5.z
else
  rcp r5.z, r1.w
  rcp r1.w, v1.w
  mul r2.xy, r1.w, v1
  mov r2.z, c2.x
  texldl r1, r2.xyzz, s0
  cmp r1.w, -r1_abs.x, c5.w, c5.z
  if_ne r1.w, -r1.w
    mov r2.w, c5.z
    texldl r1, r2.xyww, s2
    mad r1, r5.z, c1.x, -r1
    cmp r1, r1, c5.z, c5.w
    dp4 r1.w, r1, c6.x
    mul oC0, r0, r1.w
  else
     mul r1.xy, vPos, c4
  ...130 instructions...
    mul oC0, r0, r1.w
  endif
endif
```

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 ps_2_a and ps_3_0, 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



- 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 neighboring pixels in a 2×2 quad may go down different 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 can compute the LOD or derivatives outside of the dynamic flow control
- RenderMan has similar restrictions

Derivatives and Dynamic Flow Control

```
float edge = tex2D(EdgeSampler, oTex0).r;
float2 duvdx = ddx(oTex0);
float2 duvdy = ddy(oTex0);

if(edge > 0)
{
   return tex2D(BaseSampler, oTex0, duvdx, duvdy);
}
else
{
   return 0;
}
```

```
texld r0, v0, s1
cmp r0.w, -r0.x, c0.x, c0.y
dsx r0.xy, v0
dsy r1.xy, v0
if_ne r0.w, -r0.w
  texldd oC0, v0, s0, r0, r1
else
  mov oC0, c0.x
endif
```

Summary

- Vertex Shaders
  - Vertex Textures
  - Flow control
- Pixel Shaders
  - Flow control
  - Optimization
    - Shadow Mapping
  - New functionality
    - `vPos` for interleaved sampling

Acknowledgements

- Big thanks to John Isidoro, Natalya Tatarchuk and Dan Ginsburg for many of the examples used in this presentation

References

- [Keller01] Alexander Keller and Wolfgang Heidrich, “Interleaved Sampling,” Eurographics Rendering Workshop 2001.
- [Reeves87] William T. Reeves, David H. Salesin, and Robert L. Cook, "Rendering Antialiased Shadows with Depth Maps", SIGGRAPH, 1987, pp. 283-291.