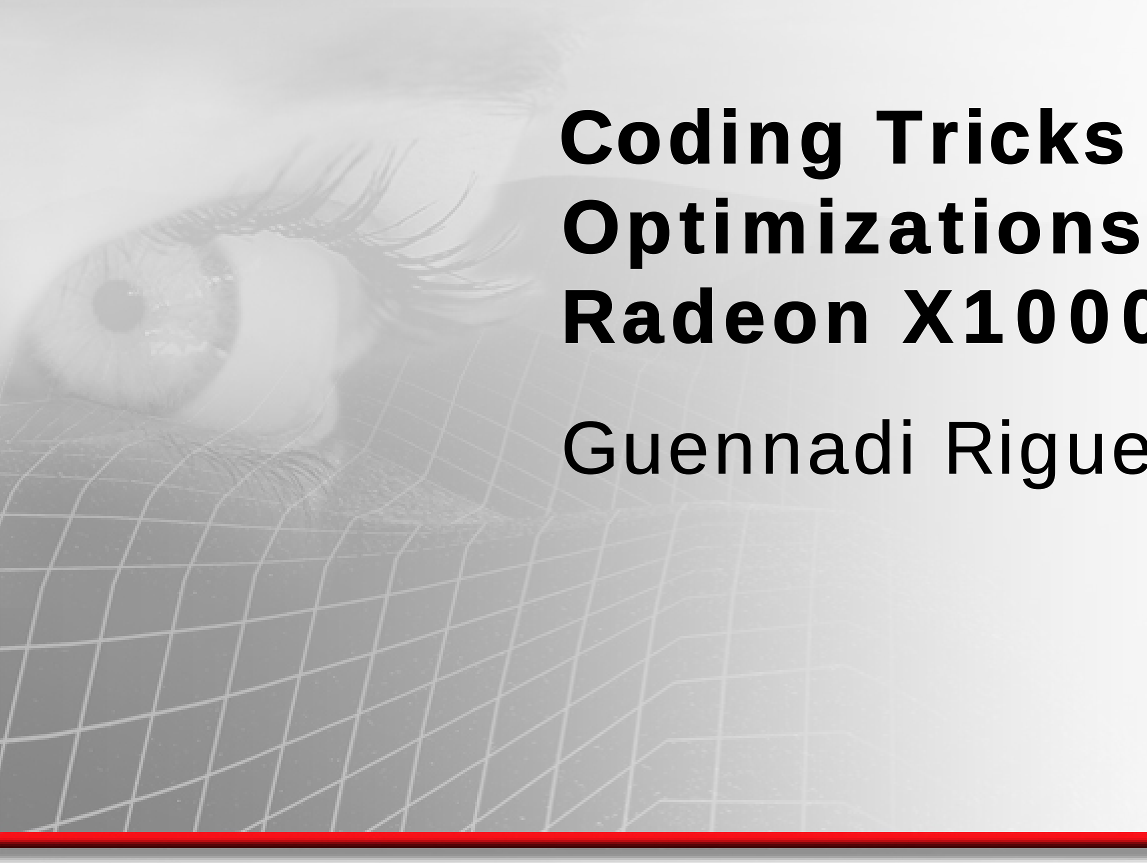


# Coding Tricks and Optimizations for Radeon X1000 Series

Guennadi Riguer

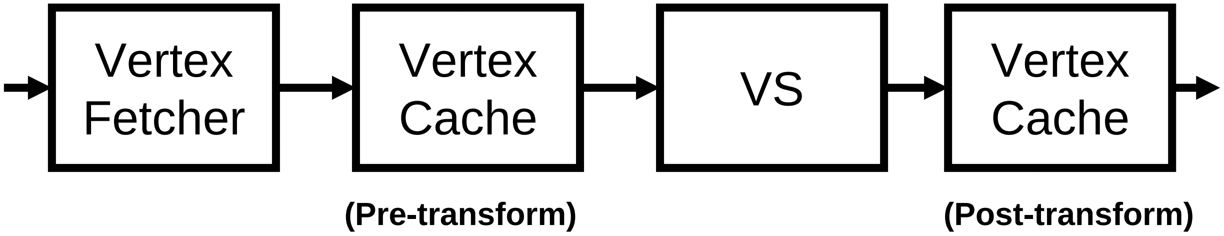
# Outline

- Radeon X1000 tricks and optimizations
- HDR on Radeon X1000
- Shadow mapping on Radeon X1000

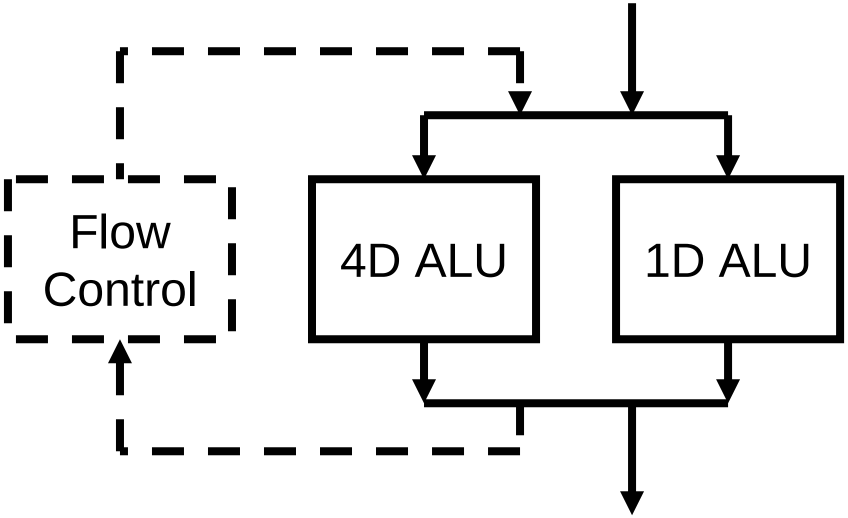
# Optimizations

# Vertex Processing

- Pipeline architecture


Vertex Fetcher
Vertex Cache
VS
Vertex Cache
(Pre-transform)
(Post-transform)

- Vertex processor architecture


Flow Control
4D ALU
1D ALU

# Vertex Fetching

- Significant improvements in VS processing power
  - More vertex processors
  - Higher core clocks
- Memory improvements are smaller
  - Bandwidth only somewhat improved
  - Latency stays the same
- Starts emerging as a similar problem to random texture fetches

# Vertex Caching

- Pre-transform cache on access to vertices in memory
  - Reduces vertex fetch bandwidth
  - Hides memory access latency
- More burden on vertex fetching and pre-transform caching than ever before
- Reduce random memory access as much as possible

# Geometry Optimization

- Always optimize your vertex data for cache friendliness
  - Use D3DXMesh->Optimize()
  - Use custom stripifier
  - Reorder vertices for locality of access
- Try to align vertex data to 32 or 64 bytes boundaries whenever it makes sense
- Use as few streams as possible
- This is important as never before!

# Vertex Shaders

- Full VS 3.0 support in Radeon X1x00

| | 9500-X850 | X1x00 |
|---|---|---|
| VS version | 2.0 | 3.0 |
| Static flow control | Yes | Yes |
| Dynamic flow control | No | Yes |
| Instructions | 256 | 1024 |
| Constants | 256 | 256 |

# Static Flow Control in Vertex Shaders

- Static flow control
  - Branching is known before shader execution
- Useful for shader management
- Driver can recompile shaders based on the Boolean constants
  - Compiled shaders are cached
- Pre-cache all shaders based on the Booleans on the first frame

# Dynamic Flow Control in Vertex Shaders

- Dynamic flow control (DFC)
  - Branching is driven by the computations in the shader
- Two types
  - Actual flow control instructions
  - Predication
- Functionally not as important as in PS
  - More architectural improvements went into PS
- Minimize use of DFC in VS
- Use a few short “if” statements or predications

# Instancing

- First class citizen on SM 3.0 part, including all Radeon X1x00
  - Some improvements for low polygon meshes
  - All ATI’s DirectX® 9 parts also support it
- Render similar objects using common data
- Reduces number of batches
  - Improvements to CPU bound cases

# Why Are Small Polys Bad?

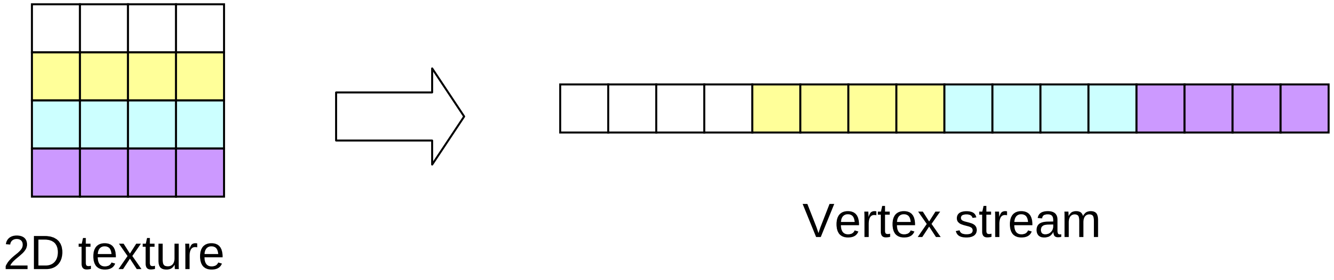
- The geometry throughput is huge:
  - >600 Mtri/s
- Should we highly tessellate everything?
  - Answer is NO!
  - Tiny triangles are BAD for performance and visual quality!
- The smallest pixel processing chunk is 2x2 quad
  - Small triangles = small quad population
  - Wastes a lot of pixel pipe power
- Ideally 20-50 pixel per triangle, at least 10-15

# Vertex Textures

- Optional SM 3.0 feature
- Radeon X1000 family doesn't support vertex textures
- Use CheckDeviceFormat() with D3DUSAGE_QUERY_VERTEXTEXTURE to determine support
- Capped OFF for all texture formats

# Render to VB

- Solves similar to Vertex Textures problems
- Very general approach
  - Allows “aliasing” textures to VB and fetching 2D texture linearly
  - Can even alias data types



# Render to VB

- Allows rendering into a texture for subsequent use in VS as vertex buffer
- Implemented as an API extension
  - Check support with FOURCC code

```
#define R2VB_FOURCC_R2VB MAKEFOURCC('R','2','V','B')
```

  - Create RT texture with D3DUSAGE_DMAP flag
  - Set stream source texture through DMAP sampler
  - Stride and offset set as normal with dummy VB
  - Enable R2VB settings through overloaded D3DRS_POINTSIZE render state

# New in Texturing

- Bigger textures (4K x 4K)
- New formats
  - ATI1N
  - New Depth Texture format
- New fetch type
  - Fetch-4
- Rotation-invariant anisotropic filtering
  - Old, cheaper method is still supported
  - No API control, driven by Control Panel settings

# Texture Cache

- Texture cache is fully associative on X1x00
  - Higher efficiency than before
- Always use mip-mapping!
  - Especially true for volume textures
- Watch out for random access
  - Will trash the texture cache
  - Occurs e.g. when sampling an environment map from a bump map
- Use compressed formats
  - DXT1-5, ATI1N, ATI2N (3Dc)

# ATI1N

- ATI1N present on all X1x00 variants
  - Single-channel compressed format
  - Used for e.g. height maps, light maps, etc.
  - 2:1 compression ratio
- ATI1N has same encoding as DXT5 alpha block

| A0 | | A1 | |
|---|---|---|---|
| xxx | xxx | xxx | xxx |
| xxx | xxx | xxx | xxx |
| xxx | xxx | xxx | xxx |
| xxx | xxx | xxx | xxx |

*ATI1N block*

| xxx | Color (A0<=A1) | Color (A0>A1) |
|---|---|---|
| 000 | A0 | |
| 001 | A1 | |
| 010 | (4/5)*A0 + (1/5)*A1 | (6/7)*A0 + (1/7)*A1 |
| 011 | (3/5)*A0 + (2/5)*A1 | (5/7)*A0 + (2/7)*A1 |
| 100 | (2/5)*A0 + (3/5)*A1 | (4/7)*A0 + (3/7)*A1 |
| 101 | (1/5)*A0 + (4/5)*A1 | (3/7)*A0 + (4/7)*A1 |
| 110 | 0 | (2/7)*A0 + (5/7)*A1 |
| 111 | 255 | (1/7)*A0 + (6/7)*A1 |

# ATI1N

- Extracted color in ***red*** channel
  - Not just a simple replacement for A8
- Accessible through FOURCC

```
#define FMT_ATI1N  MAKEFOURCC('A', 'T', 'I', '1')
```

# Depth Textures (DST)

- Allows fetching depth buffer as texture
    - Bind as depth buffer for rendering
    - Bind as texture for fetching
- Useful for shadow mapping
    - No need to output color = lower memory bandwidth
- Returns actual depth value from depth buffer
    - Doesn’t have automatic PCF-like solutions
    - Could be used for more than just shadow maps

# Depth Textures (DST)

- 16-bit format (DF16)
  - Supported on all ATI DirectX® 9 cards
  - Radeon X1800
- 24-bit format (DF24)
  - Radeon X1300 and X1600
- Uses Four-CC codes

```
#define FMT_DF16  MAKEFOURCC(‘D', ‘F', ‘1', ‘6')
#define FMT_DF24  MAKEFOURCC(‘D', ‘F', ‘2', ‘4')
```

- Create with CreateTexture() API
  - Not with *CreateDepthStencilSurface()*!

# Cool Stuff You Can Do With DST

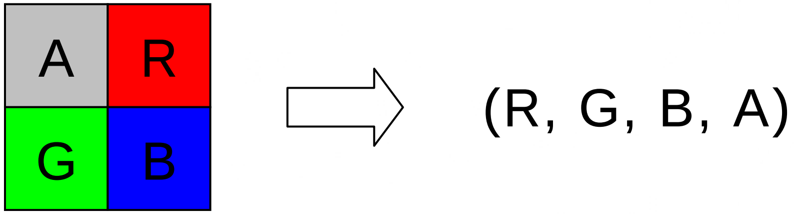
- Shadow mapping
- Depth of field
- Smooth compositing of billboarded semi-transparent objects
    - Fade out based on distance to scene
- Lens flares
- Volumetric fog
- Many others

# Fetch-4

- New feature of X1600 and X1300
  - Not available in X1800
- Returns four neighboring texels in one fetch
  - Closest 2x2 texel block
  - Same texels as in bilinear fetch
- Single-channel textures only
- Point-sampled taps
- Perfect for PCF implementations

# Fetch-4

- Four adjacent texels swizzled into RGBA channels



A R

G B

(R, G, B, A)

- Check for DF24 support for Fetch-4 availability

- Enable/disable Fetch-4 by passing the following values to D3DTSS_MIPMAPLODBIAS sampler state

```
#define FOURCC_GET4  MAKEFOURCC('G','E','T','4')
#define FOURCC_GET1  MAKEFOURCC('G','E','T','1')
```

# Cool Things You Can Do With Fetch-4

- Higher order filtering than bilinear
- Very large custom kernels
- Perlin noise evaulation
- Morphology / Edge filtering
  - Fetching the 4-connected neighborhood takes 2 fetches (vs. 5 nearest fetches)
  - Fetching the 8-connected neighborhood takes 4 fetches (vs. 9 nearest fetches)

# Floating Point Surfaces

- D3DFMT_A16B16G16R16F render target blending
  - Good render target for HDR implementations
- Multisampling supported as well!
  - Ideal for HDR rendering without sacrificing quality
- Consider killing fragments when blending to FP16 with MSAA
  - Only for short pixel shaders
- Implement fog in shaders

# FP16 Filtering

- FP16 texture filtering not supported
- Use I16 formats for filtering
  - Render to FP16
  - Convert FP16->I16
- In rare cases when absolute precision is required could simulate filtering in shader
  - This should be the last resort!
  - Don't do trilinear and anisotropic filtering in the shader

# FP16 Filtering Emulation Code

- Example of the optimal filtering code

```
float2 texWidthHeight = {TEX_WIDTH, TEX_HEIGHT};
float4 texOffsets = {
   -0.5/TEX_WIDTH+fudge, -0.5/TEX_HEIGHT+fudge,
   0.5/TEX_WIDTH-fudge, 0.5/TEX_HEIGHT-fudge};

float4 tex2D_bilerp(sampler s, float2 texCoord)
{
   float4 offsetCoord = texCoord.xyxy + texOffsets;
   float2 fracCoord = frac(offsetCoord.xy * texWidthHeight);

   float4 s00 = tex2D(s, offsetCoord.xy);
   float4 s10 = tex2D(s, offsetCoord.zy);
   float4 s01 = tex2D(s, offsetCoord.xw);
   float4 s11 = tex2D(s, offsetCoord.zw);
   s00 = lerp(s00, s10, fracCoord.x);
   s01 = lerp(s01, s11, fracCoord.x);
   s00 = lerp(s00, s01, fracCoord.y);
   return s00;
}
```

# 10-bit Surfaces

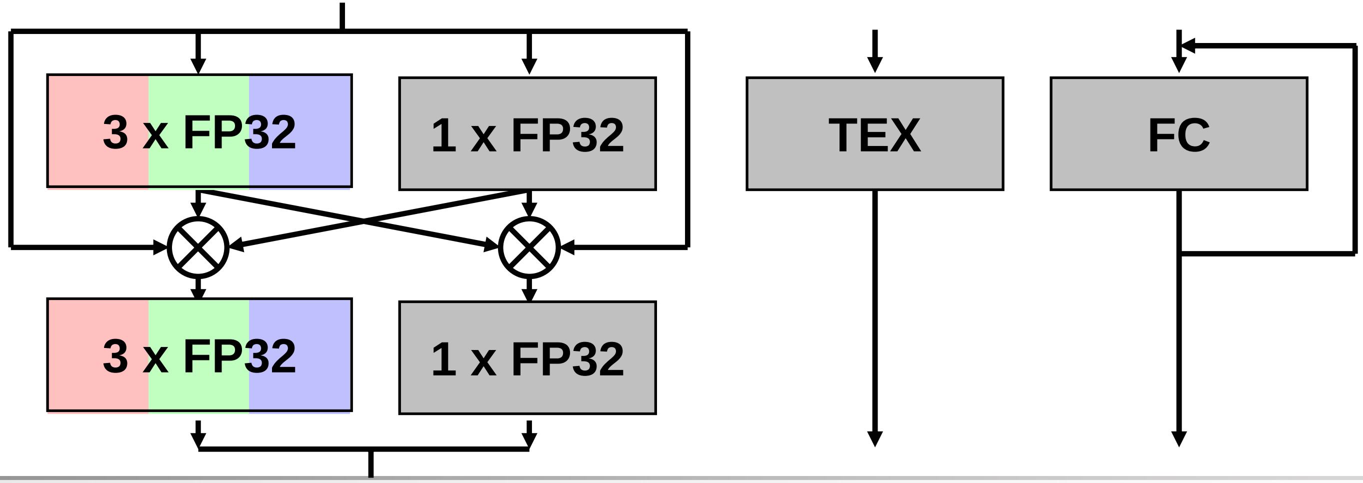
- First-class citizen on all Radeon X1x00
  - Supports blending, filtering and MSAA
- It is also displayable!
  - Full screen only
  - High fidelity visual outputs
  - Benefits LCDs as well – high-quality dithering in display engine
- Perfect for not-so-high HDR (MDR)
  - Fixed point 2.8 format
  - Could use gamma correction to boost the range at small quality degradation

# Multiple Render Targets

- X1x00 now supports separate color masks for Multiple Render Targets
- Indicated by D3DPMISCCAPS_ INDEPENDENTWRITEMASKS cap bit
- Set with SetRenderState()
    - D3DRS_COLORWRITEENABLE0
    - D3DRS_COLORWRITEENABLE1
    - D3DRS_COLORWRITEENABLE2
    - D3DRS_COLORWRITEENABLE3

# Pixel Shader

- New pixel shader architecture
  - Advanced thread scheduler
  - Full PS 3.0 support
- ALU based on proven R300 architecture with improvements
  - Simultaneous ALU, TEX and FC execution



# PS 3.0 Capabilities

- Full PS 3.0 support in Radeon X1x00

| | X800-X850 | X1x00 |
|---|---|---|
| PS version | 2.x | 3.0 |
| Static flow control | No | Yes |
| Dynamic flow control | No | Yes |
| Instructions | 512 | 512 |
| Dependent reads | 4 | No limit |
| Face and position | No | Yes |
| Arbitrary swizzles | No | Yes |
| Temps | 32 | 32 |
| Constants | 32 | 224 |

# Shader Optimization Tips

- Compiler does a pretty good job optimizing shaders
  - Don’t get hung up too much on hand-tuning
- Use swizzles and write masks to enable automatic co-issue
- Explicitly vectorize calculations
  - Especially important for 2D+2D case in post-processing shaders
- Use literal constants in shaders

# Instruction Balancing

- New and exciting trend – bigger ALU:TEX ratio
  - Easier to increase ALU power than memory bandwidth
  - History trends support this
- X1800, X1300 aim at least at 4:1 ratio
  - Simultaneous 1 macro-ALU + 1 TEX execution
- X1600 aim at least at 8:1 or even 12:1 ratio
  - Simultaneous 3 macro-ALU + 1 TEX execution
- Expensive filtering skews ratio towards ALU

# Static Flow Control in Pixel Shaders

- Same as in VS
  - Useful for shader management
  - Need pre-caching for the best runtime performance
- Why all this pre-caching anyway?
  - The flow control instructions are relatively inexpensive, but...
  - They limit compiler's ability to re-schedule instructions
  - In extreme cases up to 50% performance loss

# Dynamic Flow Control (DFC) in Pixel Shaders

- One the most important features of PS 3.0
  - Major architectural improvements in X1x00
- First class citizen
  - Almost “free” flow control instructions
  - Smallest execution thread granularity
    - X1800: 16 pixels
    - X1900: 48 pixels

# Why is efficient DFC hard?

- GPU are massively parallel
  - Smallest processing element is 2x2
  - Actual threads have multiple quads
  - All pixels in the thread follow all executed paths
- Does it mean DFC is bad?
  - No, just have to be careful

# Using DFC in Pixel Shaders

- Ensure reasonable coherency of execution
- Use DFC for optimizations – skip unnecessary computations and fetches
    - if (dot(N, L)>0) ...
    - if (!bInShadow) ...
    - if (DistanceFromLight<Falloff) ...
    - Early out with alpha testing
- Avoid many small conditionals
- No loops and less than 6 nested levels of branching goes into fast path mode

# Predication in Pixel Shaders

- Predication – flow control technique in vector processors
- Based on conditionals control vector instruction execution per-channel
- No need to explicitly use it
  - DirectX® spec is flexible, compiler can convert predication <-> flow control statements

# Screen Gradients

- Gradient computations use 2x2 quad
  - With DFC pixels in the quad could go down different paths resulting in ambiguous results
- Nothing that computes gradients can reside inside flow control
  - Fetches based on computed coordinates
  - Gradients of computed values
- Interpolated texture coordinates are OK
  - Otherwise use texldl, texldd and etc.
- Failure to comply will remove DFC from shader code in HLSL or will fail in ASM

# Optimal HLSL Use

- For VS use:
  - D3DXSHADER_AVOID_FLOW_CONTROL compiler flag
  - /Gfa command line option
- For PS use:
  - D3DXSHADER_PREFER_FLOW_CONTROL compiler flag
  - /Gfp command line option

# Hyper-Z Technology

- Fast Z clear
- Z compression
- Hierarchical Z
- Early Z testing

# Z Compression and Fast Clears

- Fast Z Clears
  - Z and stencil buffer are contained in the same surface
  - Clear Z and stencil together
- Compressed Z buffer
  - Z buffer values are block-compressed to save Z bandwidth
  - Lossless compression
  - Main Z buffer automatically compressed for performance
  - Depth textures are not compressed (DF16, DF24)

# Hierarchical Z

- Keeps the max or min Z value per block in on-chip memory
- Depth compare is performed per-block (tile)
    - If incoming Z values are greater/smaller than block Z then the triangle portion is hidden
    - Else triangle is split into smaller blocks
- Allows fast Z culling of whole (or portions of) triangles
    - Doesn’t work with PS depth output
    - Can break if Z compare modes are reversed
- Render alpha-tested/texkill primitives after opaque ones
    - This increases the chance of those being rejected by HiZ culling
- Make Z near/far range as close to geometry as possible

# Early Z Testing

- Ability to reject pixels before shading them
- Independent of Hierarchical-Z
- Early Z doesn’t work in two cases
    - When shader outputs depth
    - When alpha-test or texkill is used
- Sort front-to-back or consider depth-only pass

# HDR on Radeon X1000

# What is HDR?

- High Dynamic Range
- Dynamic range is ratio of brightest to darkest values
  - HDR has dynamic range greater than 255:1 used in “normal” rendering
- Usually HDR requires greater than [0..1] range
- Allows seeing details in both shadows and bright areas

# Radeon X1x00 Capabilities

- I10 formats
  - Filtering
  - Blending
  - MSAA
- I16 formats
  - Filtering
- FP16 (4-channel)
  - Blending
  - MSAA
- I16 complements FP16 in terms of functionality

# Good Low-End Implementation on Radeon X1300- X1600

- Perfect for MDR
- Use I10 for rendering with MSAA
  - E.g. use 2.8 fixed point format
  - Gamma correction can give bigger range
- Resolve to I10 buffer to be used for further post-processing
- For intermediate post-processing steps could use I10 or I16 (both are filterable)
- Use I10 or I8 for final display

# Good High-End Implementation on Radeon X1600- X1900

- Use FP16 for rendering with MSAA
- Use alternative HDR format representations for textures
- Resolve and copy to I16 buffer to be used for further post-processing
- For intermediate post-processing steps could use I10 or I16 (both are filterable)
- Use I10 for final display

# Is I16 Good Enough For Post-Processing and Texturing?

- In majority of cases – YES!
- Represents 65535:1 dynamic range
- For limited ranges better precision than FP16 (16 vs. 10 bits)
- Many more bits than will be actually displayed
  - Many bits to spare for post-processing
- There are some really cool solutions based on integer formats
  - E.g. expanded range support using integer formats

# How Do You Do It?

- Many different methods Fixed scaling
  - RGBS
  - RGBE
  - Compressed RGBE
  - PPP
  - RGBS+PPP
  - EEE
- Examples use extreme range of values – up to [0.004, 76800.0]
  - That’s 19,200,000:1 dynamic range!
  - Extreme exposure (night sky looks like daylight)

# Fixed Scaling

- Use I16 with a simple scale
    - E.g. fixed point 8.8 format
- Pros
    - Range up to 0..255 with good precision (8.8)
    - Works well for over-brightening
    - In practice can tolerate a bit of range clamping
    - Matches bilinear filtering of FP16
    - The simplest and cheapest method for both encoding and decoding
- Cons
    - Fails for large ranges

# Fixed Scaling Implementation

- Encoding

```
// Convert to [0..1] range
color.rgb *= invMaxValue;
```

- Decoding

```
// Decode to full range
float3 tex = tex2D(Texture, texCoord).rgb;
tex.rgb *= maxValue;
```

# Fixed Scaling Example

# RGBS Format

- Store common variable scale factor in the alpha channel
  - Floating point with linear range distribution
- Pros
  - Cheap decoding (2 instr.)
  - Better range than with a fixed scale
  - Overall pretty good quality for reasonably large ranges
- Cons
  - Fails for extremely large ranges
  - Fairly expensive encoding (up to 9 instr.)

# RGBS Implementation

- Encoding

```
// Might need to clamp
color.rgb = min(color.rgb, maxValue);
// Find max value
float maxChannel = max(max(color.r,
color.g), color.b);
// Move scale to alpha
color.rgb /= maxChannel;
color.a = maxChannel * invMaxValue;
```

- Decoding

```
// Decode to full range
float4 tex = tex2D(Texture, texCoord);
tex.rgb *= tex.a * maxValue;
```

# Improved RGBS

- Precision problem in darker parts
  - Too few bits for scale of dark values
  - This is because color uses max bits available
- Redistributing bits between color and scale
  - Try to use similar number of bits for both
  - Use adjustment factor
    - Divide color by the adjustment
    - Multiply scale by the adjustment

$$\frac{\max(R,G,B)}{f} = f \cdot A$$

Adjustment: $f = \sqrt{\frac{1}{A}}$

# Improved RGBS Implementation

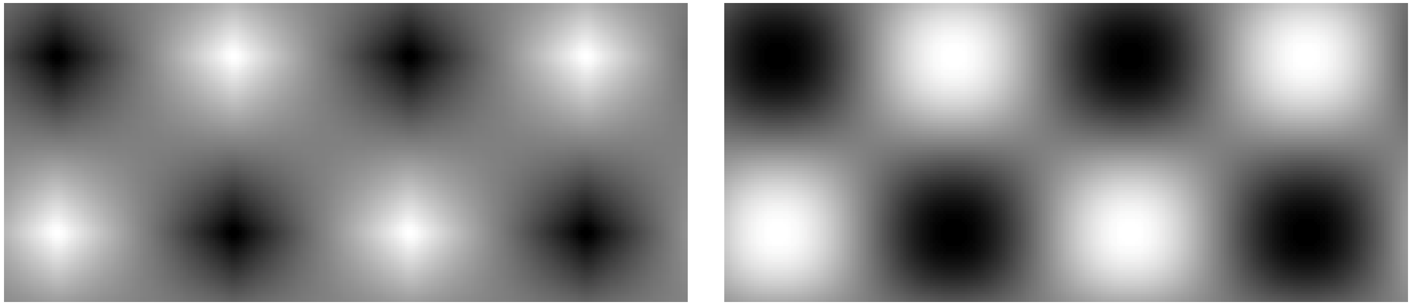
- Encoding

```
// Might need to clamp
color.rgb = min(color.rgb, maxValue);
// Find max value
float maxChannel = max(max(color.r,
color.g), color.b);
// Move scale to alpha
color.rgb /= maxChannel;
color.a = maxChannel * invMaxValue;
// Redistribute bits
color.a *= rsqrt(color.a);
color.rgb *= color.a;
```

# RGBS Format Example

# Wouldn't It Break Filtering?

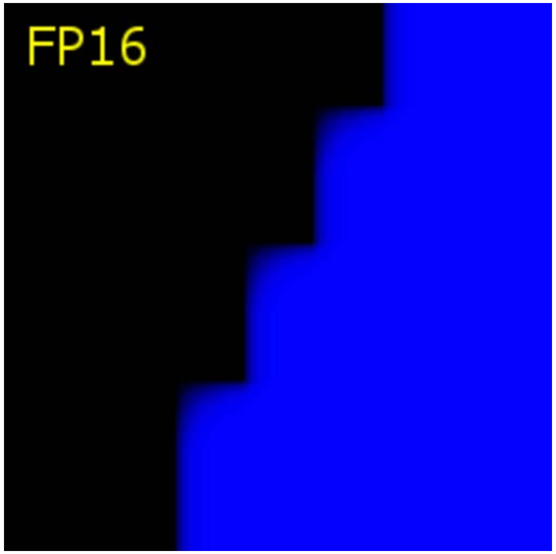
- Yes, it breaks bilinear filtering, but that's not really a problem
- Bilinear filter is far from perfect for reconstructing signals
  - Bilinear is used because it's cheap and "good enough" (makes images look smooth)
- There are better filters
  - Just compare bilinear and bicubic…

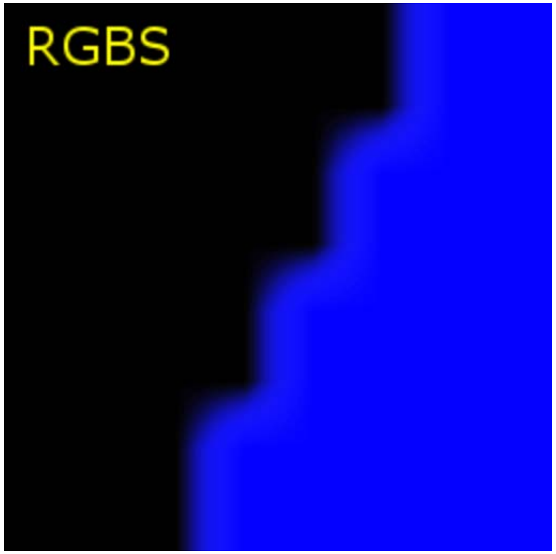


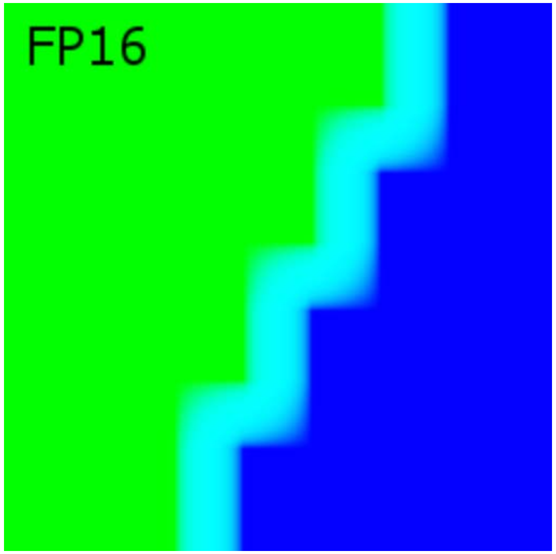
# Wouldn't It Break Filtering?

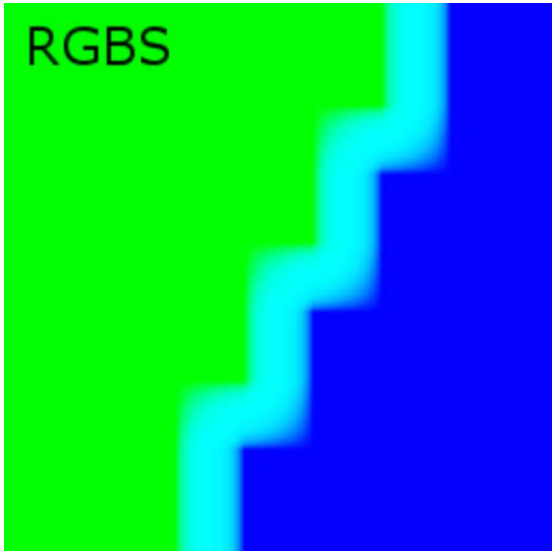
- These methods don't match bilinear filter
  - ... but they achieve the same goal
- Pros
  - Make images look smooth
  - When dealing with HDR non-linearity not always a bad thing
- Cons
  - Could produce slight haloing in cases of very rapid value changes (rarely a huge problem)
  - Might need some tweaking to make it look the best

# Filtering Comparison (RGBS)









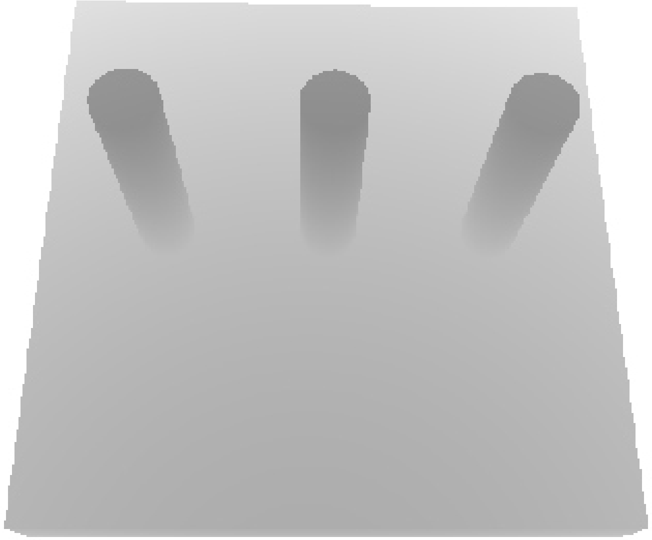
# Summary of HDR Formats

- Check out **HDR Texturing** whitepaper
- Pick method that works the best
  - Quality vs. cost tradeoff

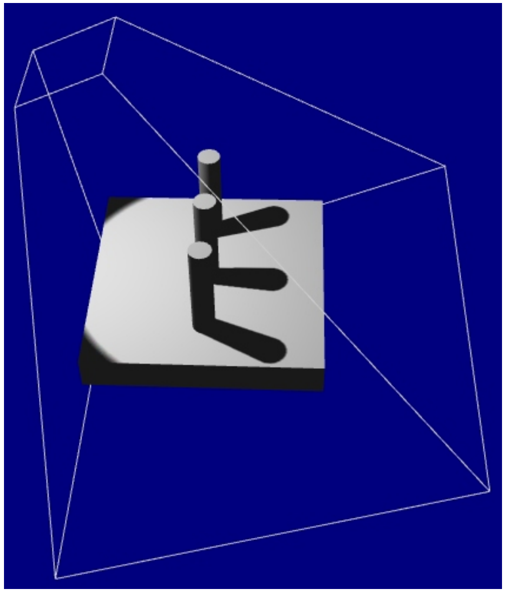
| **Method** | **Encod. instr.** | **Decod. Instr.** | **Free Alpha** | **Range** | **Filter quality** | **Overall quality** |
|---|---|---|---|---|---|---|
| Fixed scale | 1 | 1 | Yes | Low | +++ | + |
| RGBS | 5-9 | 2 | No | Med | ++ | + |
| RGBE | 5-6 | 3 | No | High | + | +++ |
| PPP | 5/8 | 2/8 | Yes | High | +++ | ++ |
| RGBS+PPP | 10-14 | 3 | No | High | ++ | +++ |
| EEE | 4 | 4 | Yes | High | + | ++ |

# Shadow mapping on Radeon X1000

# Shadow Mapping



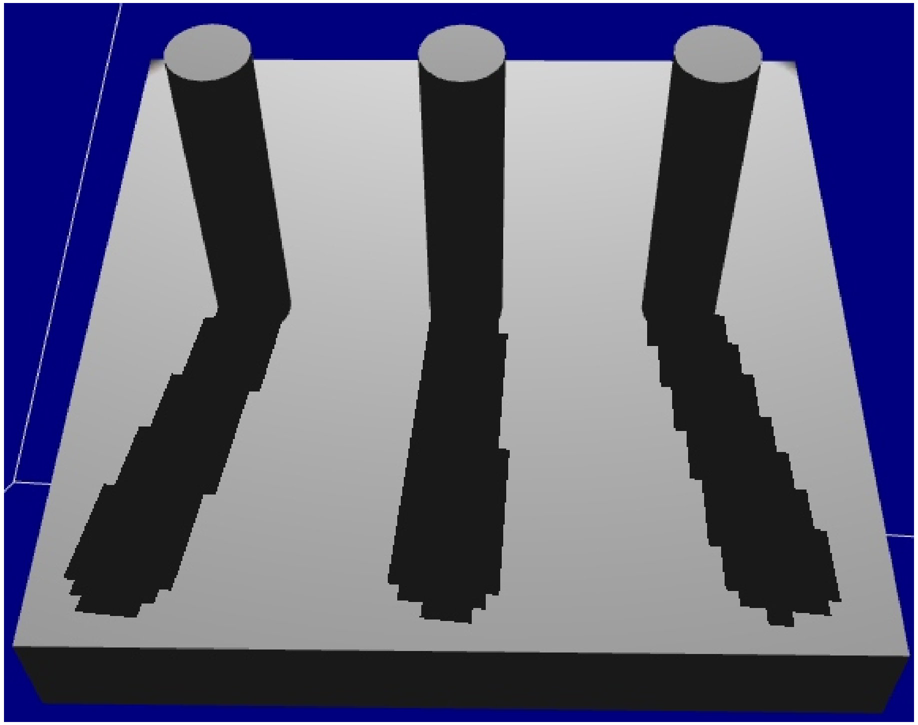
Shadow Map



Scene With Shadow Map

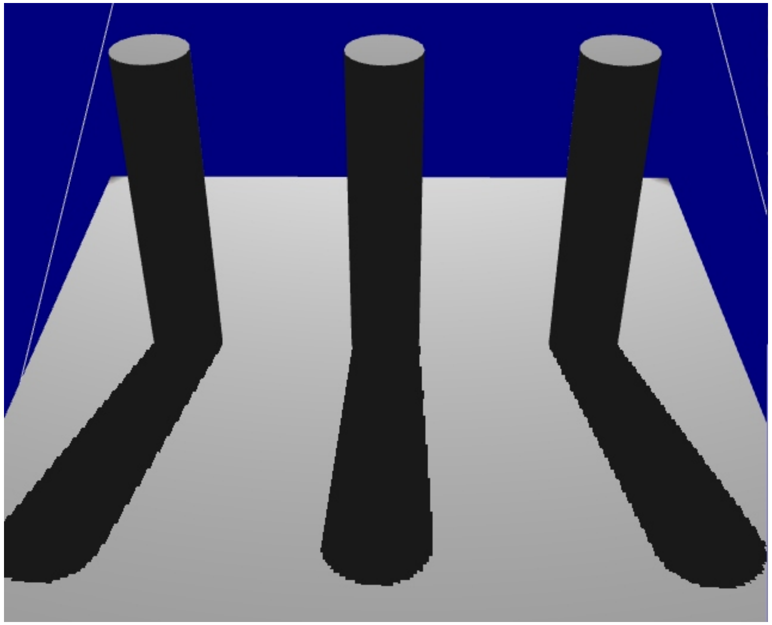
- Shadow map: render depth from the light's point of view
- Render the scene from the eye's point of view
  - Project the shadow map onto the scene using the light space transform.
  - Transform the current position into light space, and compare its depth values with the depth values stored in the shadow map

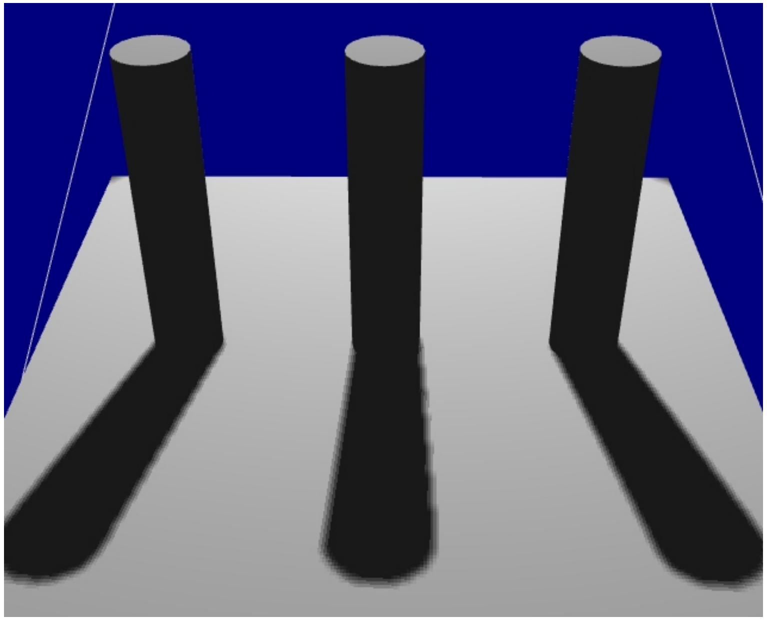
# Aliasing



- A standard issue with shadow mapping is aliasing
  - Raising shadow map resolution is expensive

# Percentage Closer Filtering (PCF)


1-Tap Hard Shadowmapping


4x4 (16-tap) PCF

- Helps with aliasing problem
- Use multiple samples from the shadow map
- First compare then perform filtering

# PCF Optimization: Step 1

- Processing multiple taps in parallel

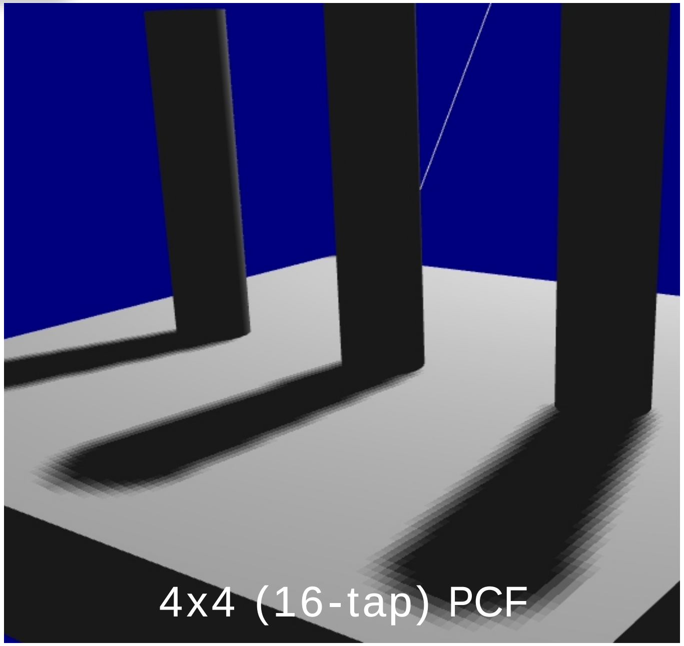
```
//Projected coords
projCoords = oTex1.xy / oTex1.w;

//Sample nearest 2x2 quad
shadowMapVals.r = tex2D(ShadowSampler, projCoords);
shadowMapVals.g = tex2D(ShadowSampler, projCoords +
       texelOffsets[1].xy * g_vFullTexelOffset.xy);
shadowMapVals.b = tex2D(ShadowSampler, projCoords +
       texelOffsets[2].xy * g_vFullTexelOffset.xy);
shadowMapVals.a = tex2D(ShadowSampler, projCoords +
       texelOffsets[3].xy * g_vFullTexelOffset.xy);

//Evaluate shadowmap test on quad of shadow map texels
inLight = (dist < shadowMapVals);

//Percent in light
percentInLight = dot(inLight, float4(0.25, 0.25, 0.25, 0.25));
```

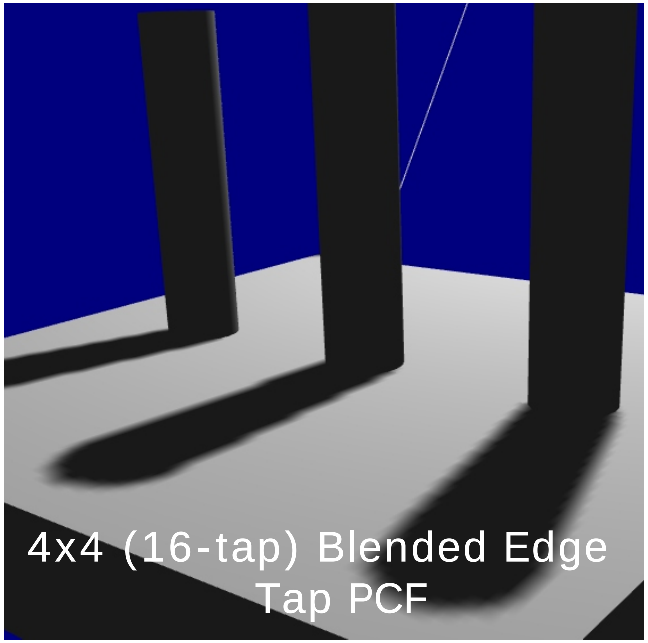
# PCF Optimization: Step 2

- Take advantage of “Fetch-4”

```
// Sample nearest 2x2 quad
// (using 2x2 neighborhood fetch into .rgba )
shadowMapVals.rgba = tex2Dproj(ShadowSampler, projCoords);

//Evaluate shadowmap test on quad of shadow map texels
inLight = (dist < shadowMapVals);

//Percent in light
percentInLight = dot(inLight, float4(0.25, 0.25, 0.25, 0.25));
```

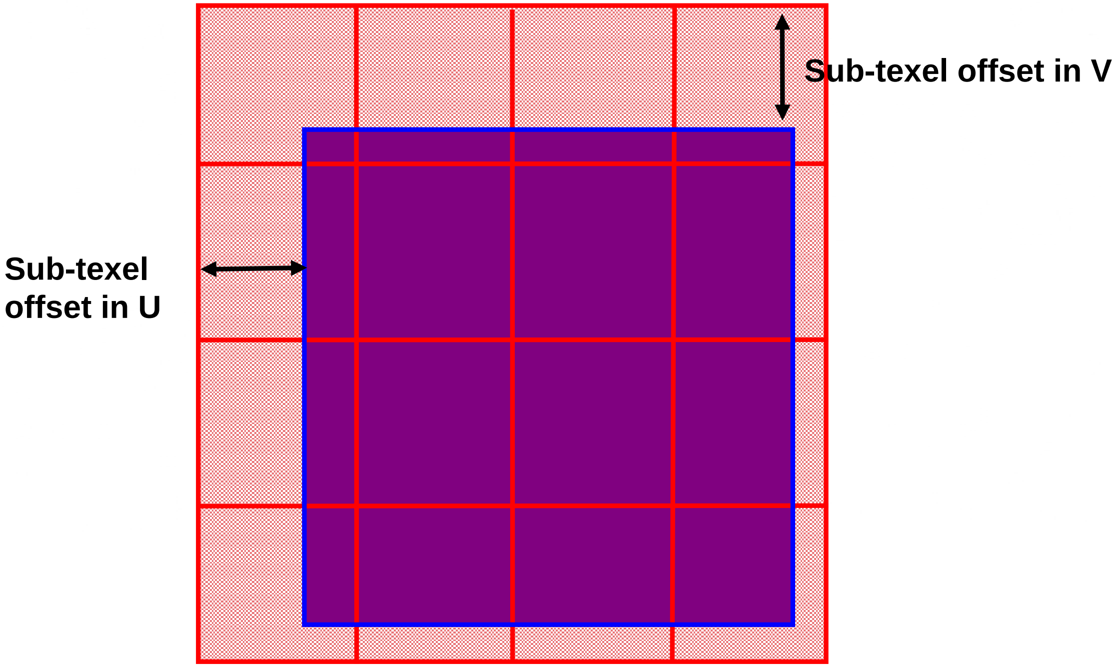
# Edge Tap Smoothing


4x4 (16-tap) PCF


4x4 (16-tap) Blended Edge Tap PCF

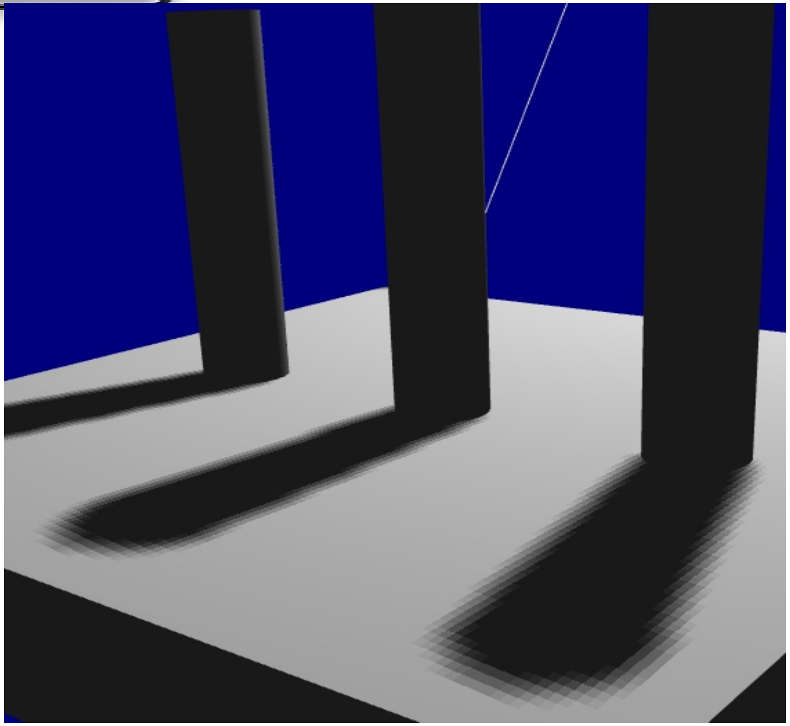
- In basic PCF has a limited number of intensity levels:
  - 2x2 PCF = 4 intensity levels
  - 4x4 PCF = 16 intensity levels
  - 6x6 PCF = 36 intensity levels
  - 8x8 PCF = 64 intensity levels
- Cheap alternative: area filter

# Edge Tap Smoothing

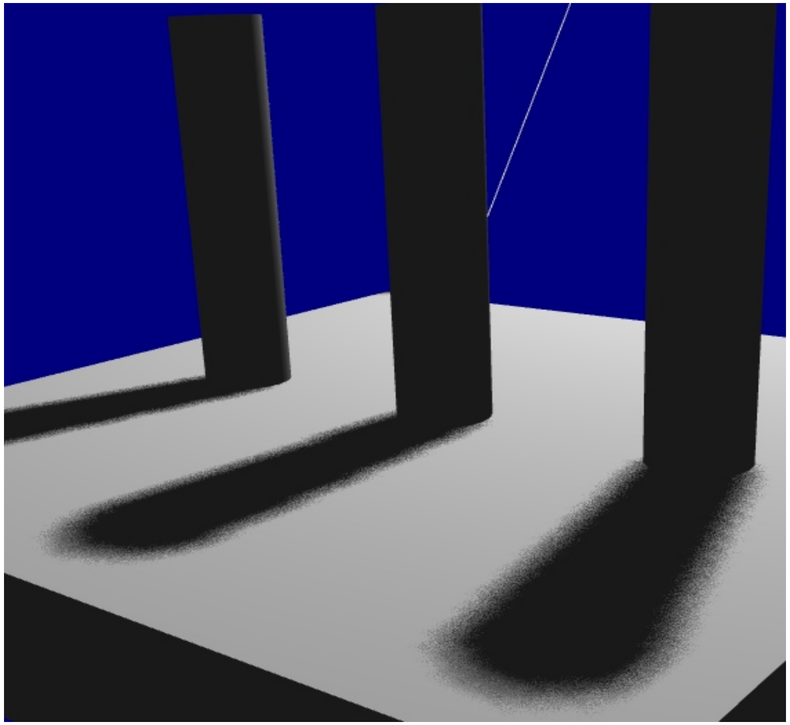


- 3x3 area filter with 16 taps
- Can be optimized using fetch-4 (4 fetches)
- Fast alternative to bicubic, Gaussian, or other higher order kernels

# Non-grid Based PCF Offsets



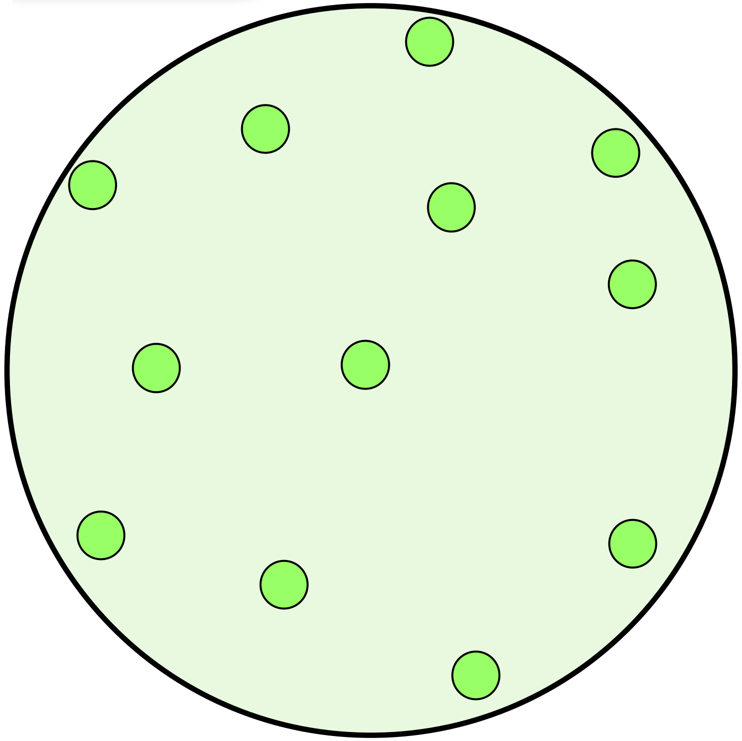
4x4 (16-tap) PCF



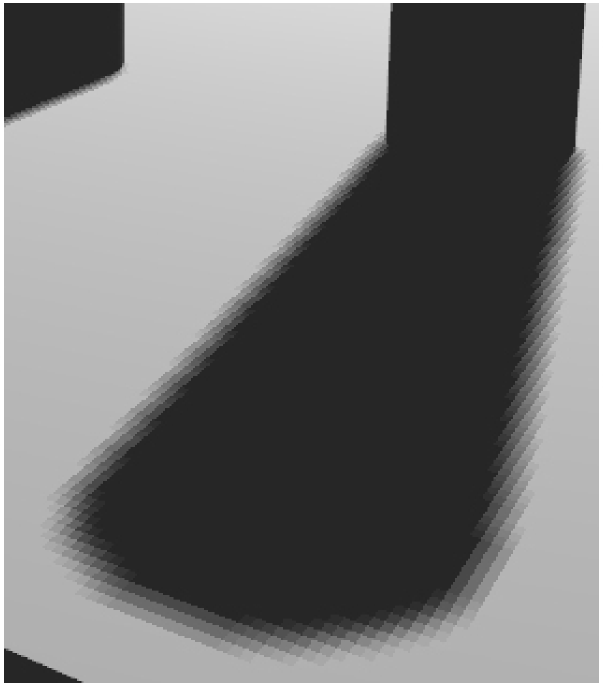
(12-tap) Randomized Offset PCF

- Grid based PCF kernel needs to be fairly large to eliminate aliasing artifacts
- Need fewer samples with non-uniform sampling

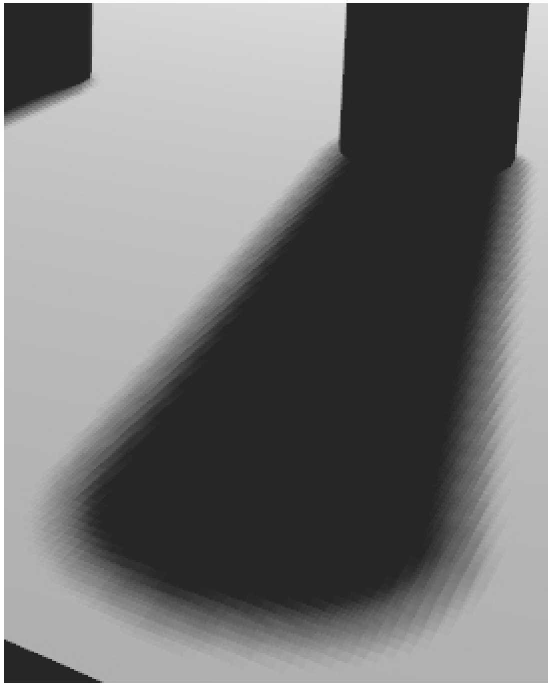
# Non-Uniform Disc Sampling



12-tap disk PCF



4x4 (16 tap) PCF



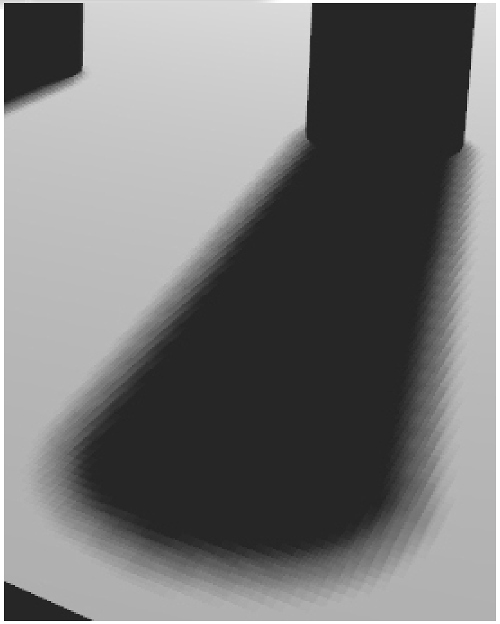
12-tap fixed disk PCF

- Store tap offsets from center of the kernel as constants

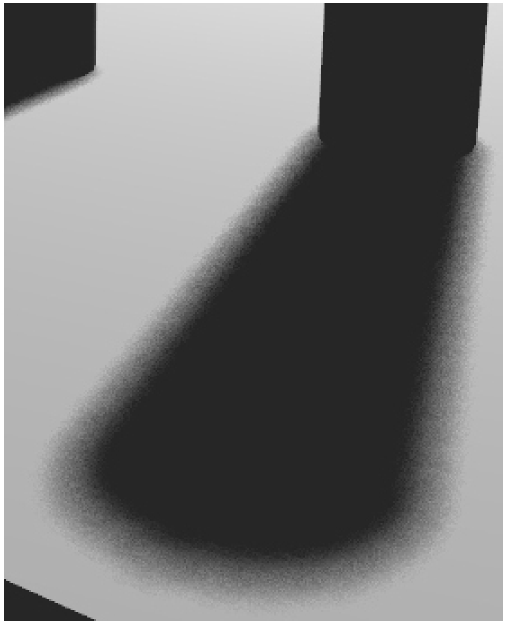
# Randomized PCF Offsets

- Changing random offsets per frame has undesirable “TV noise” effect
- Precompute random values in screen aligned texture:
  - When scene is static, randomness in penumbra is static
- Unique per pixel rotation of the disc kernel works well
  - Preserves distances in between taps in the kernel
  - Make sure no tap is directly in the center

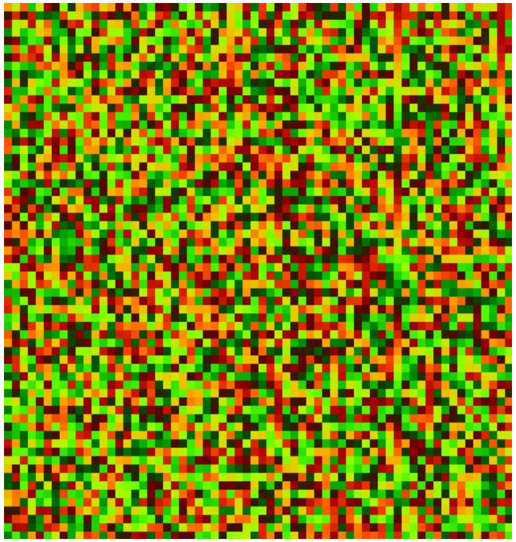
# Rotated Disk Kernel



12-tap fixed disk PCF

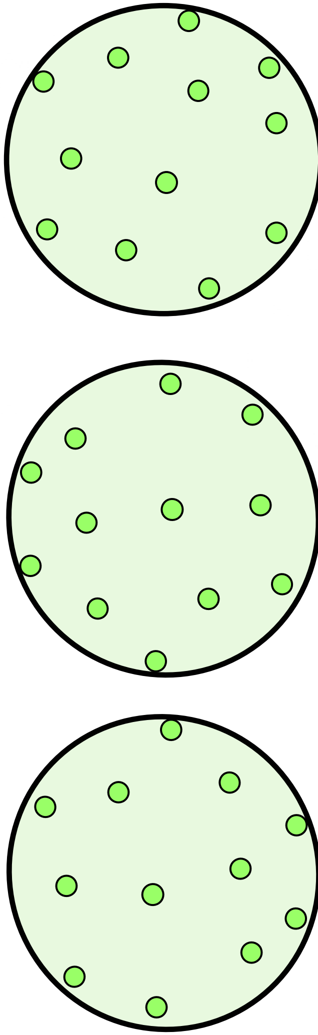


12-tap per-pixel uniquely rotated disk PCF



Example 64x64 unique rotation texture

red=cos(x)
green=sin(x)

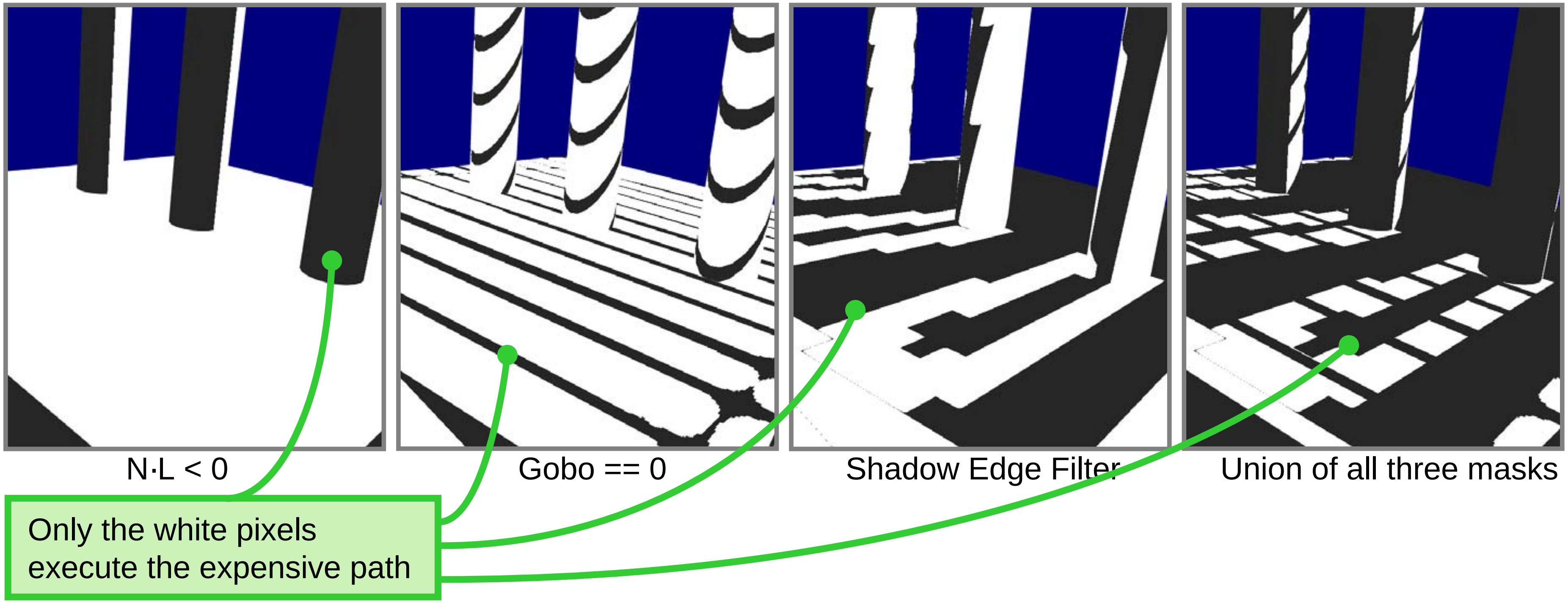


- Use screen space location as random seed
- Look up in “random” rotation texture

# Shadow Map Filtering Mask

- Need to use expensive filter only on the shadow edges
- Use flow control in PS to skip expensive computations
  - Trivially compute full shadow and lighting
- Use shadow mask

# Shadow Mask Construction



N·L < 0

Gobo == 0

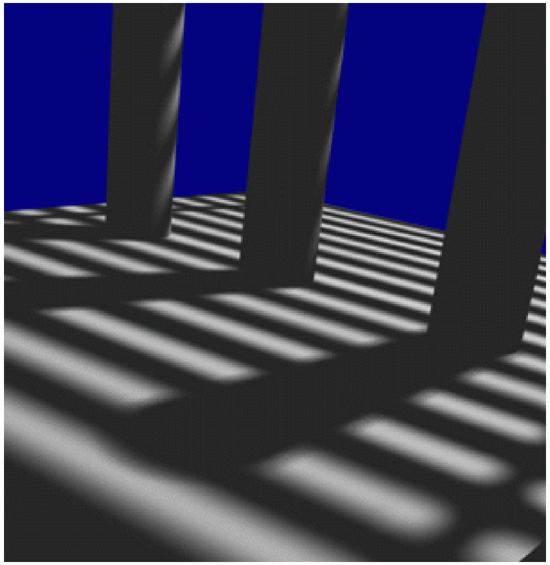
Shadow Edge Filter

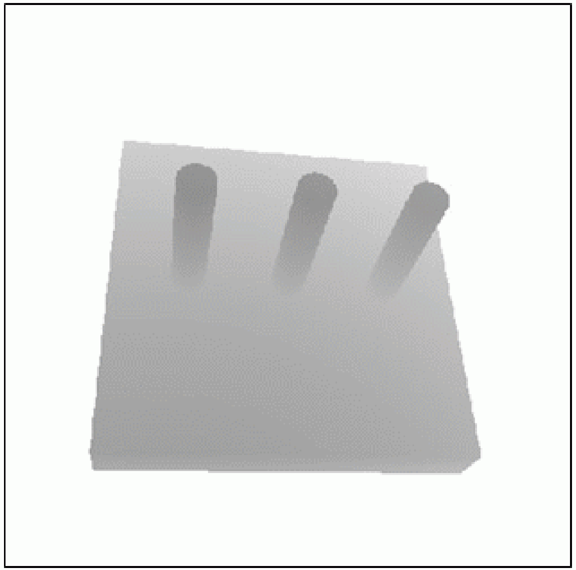
Union of all three masks

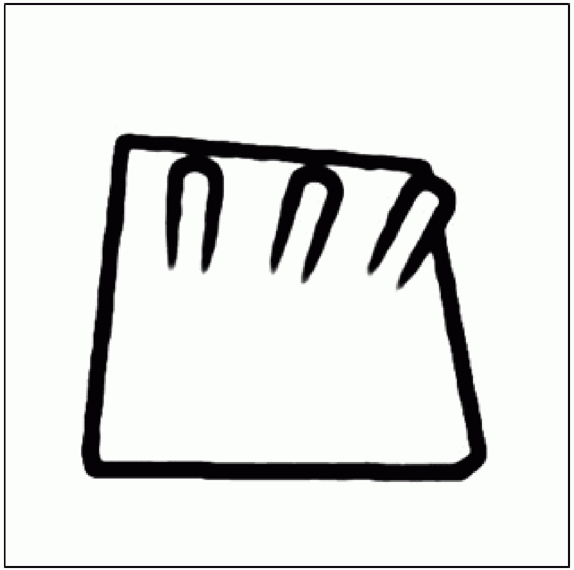
Only the white pixels execute the expensive path

- Combine several trivial rejections together

# Edge Mask

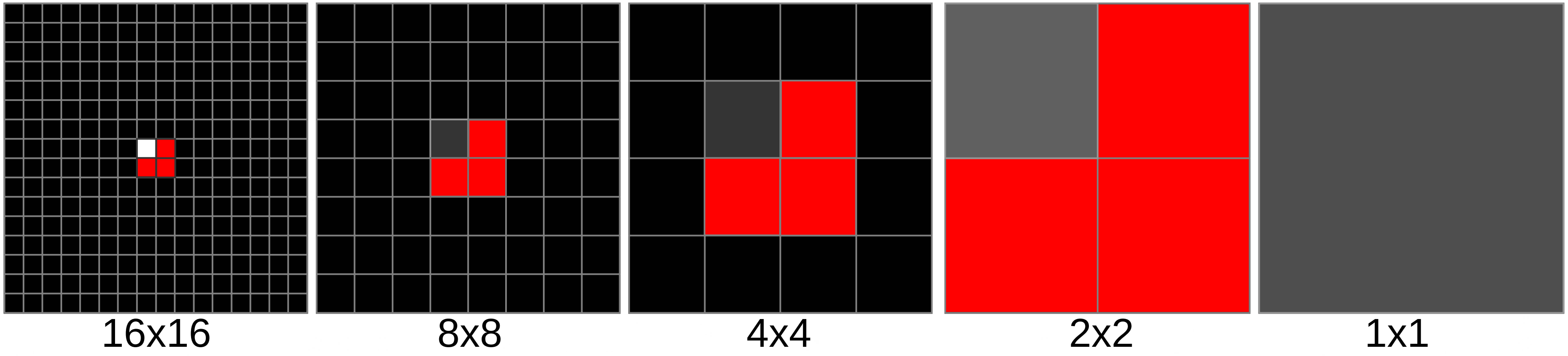

Desired final image


Shadow Map


Edge Map

- Penumbra regions only near depth discontinuities (edges) on the shadow map
- Find edges based on depth
- Dilate edge map to at least the width of the filtering kernel

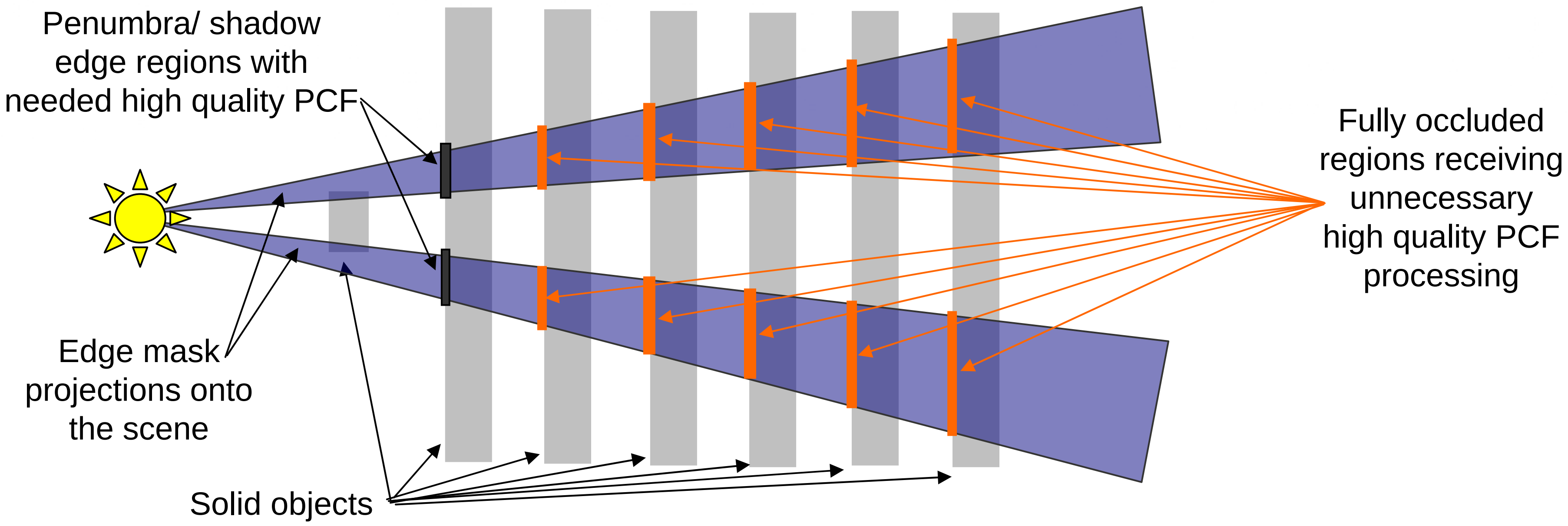
# Edge Mask Dilation



- Use bilinear filter for mask expansion
- Use lower mip level for mask testing
  - PCF kernel size determines mip-level
  - Test mask for non-zero values for detecting penumbra regions

# Scene Depth Complexity

- Edge mask works well only for low depth complexity



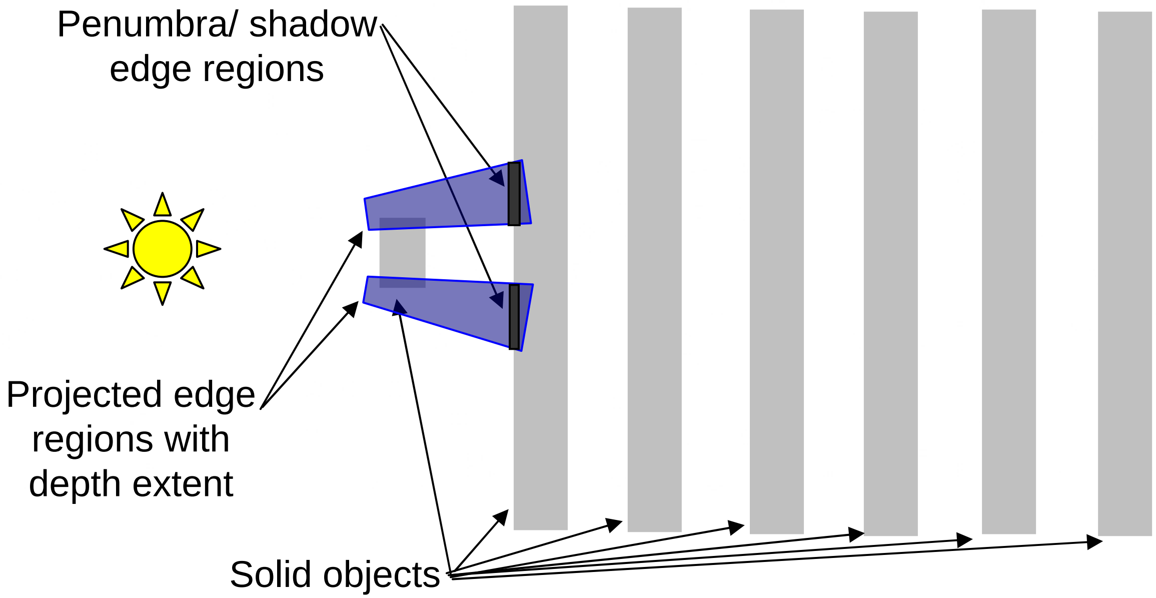
Penumbra/ shadow edge regions with needed high quality PCF

Fully occluded regions receiving unnecessary high quality PCF processing

Edge mask projections onto the scene

Solid objects

# Depth Extent Masking

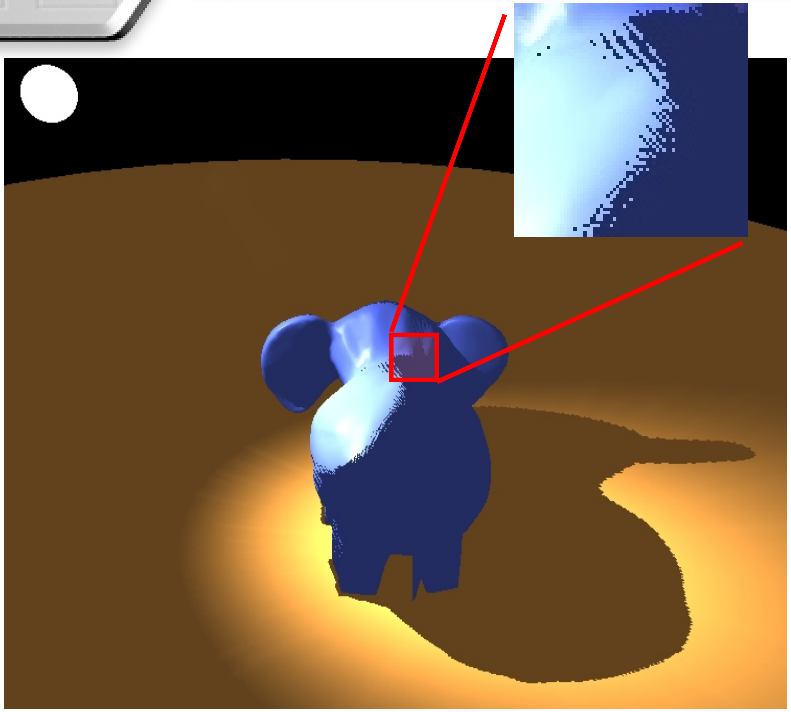


- Compute min/max depths for the region
- Propagate min/max values during dilation
- Similar to hierarchical Z

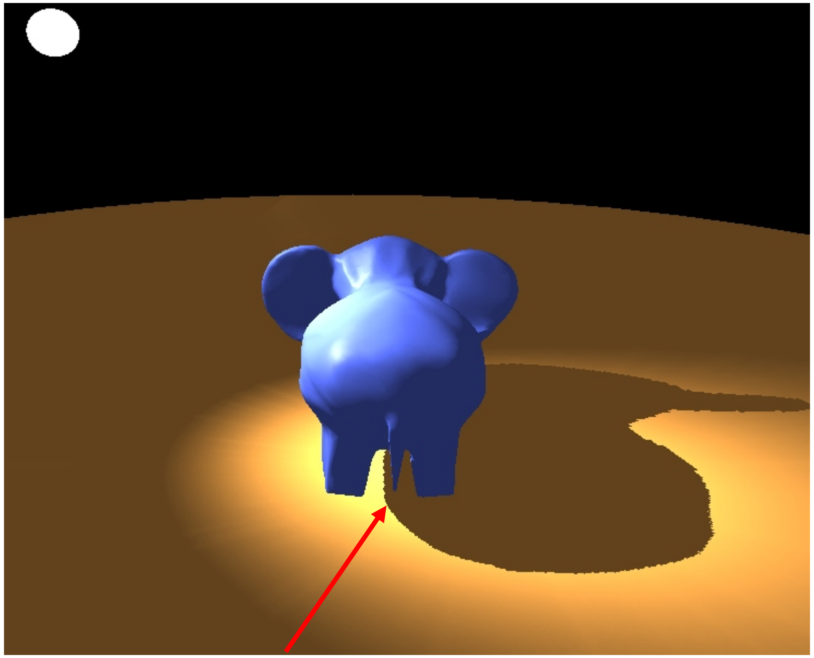
# Depth Extent Masking Example

```
//compute lighting for the point on the surface
lightVal = ComputeLighting(oTex1, dist, oTex2, oTex0);

//if there is no light hitting this surface, then return 0
if (dot(lightVal, float3(1, 1, 1)) == 0) {
   return 0;  //no lighting, return 0
}
else {
   //fetch from depth extent texture
   projCoords.zw = g_fEdgeMaskMipLevel;
   edgeValMinMax = tex2Dlod(EdgeMipSampler, projCoords).rg;

   if ((edgeValMinMax.r < dist) && (edgeValMinMax.g > dist)) {
      //perform high quality PCF filtering here and return
      //  . . . . . . . . .
   }
   else {
      //perform single tap shadow mapping here and return
      //  . . . . . . . . .
   }
}
```

# Shadow Map Bias



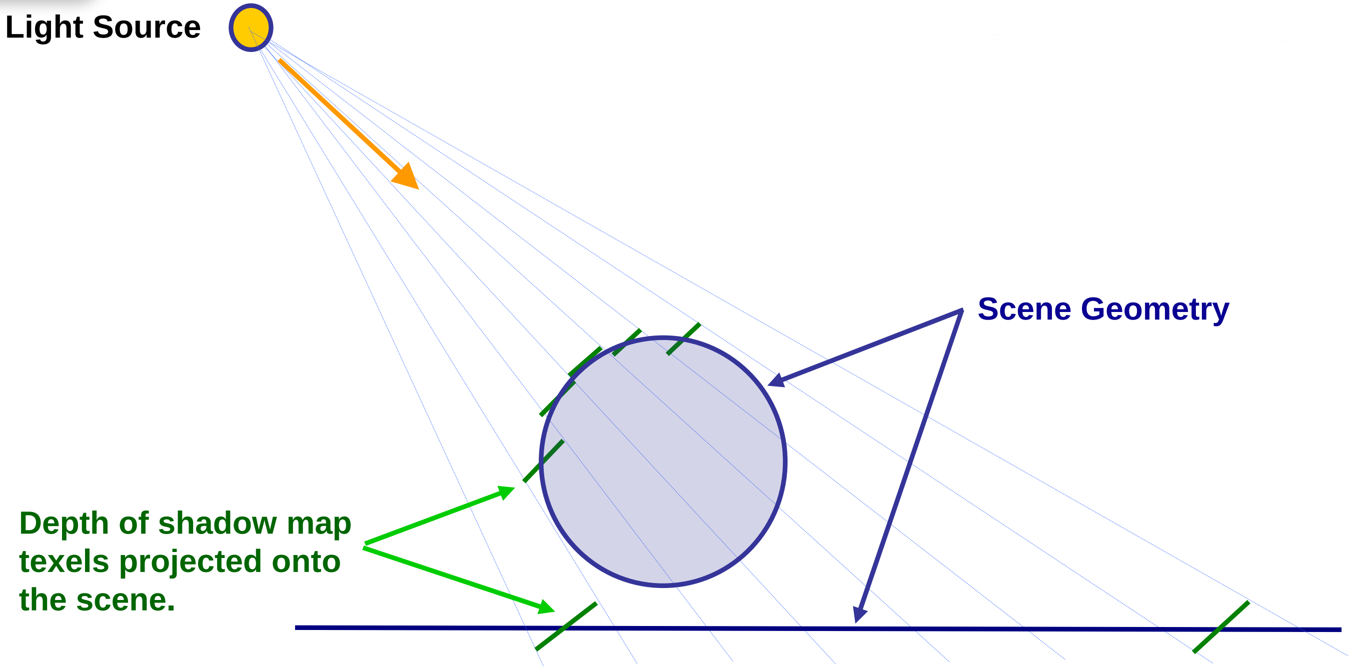
Too Little Bias: Surface Acne



Too Much Bias: Floating Shadow

- Need to bias depth comparison
- Picking right bias value is hard
  - Too little: surface acne
  - Too much: disconnected shadows

# Two Components of Bias



- Numeric: due to the shadow map precision
- Geometric: due to representing an area of texel projection with a single depth value
- Bias depends on the shadow map resolution, slope of the scene to the light source, and precision of depth map

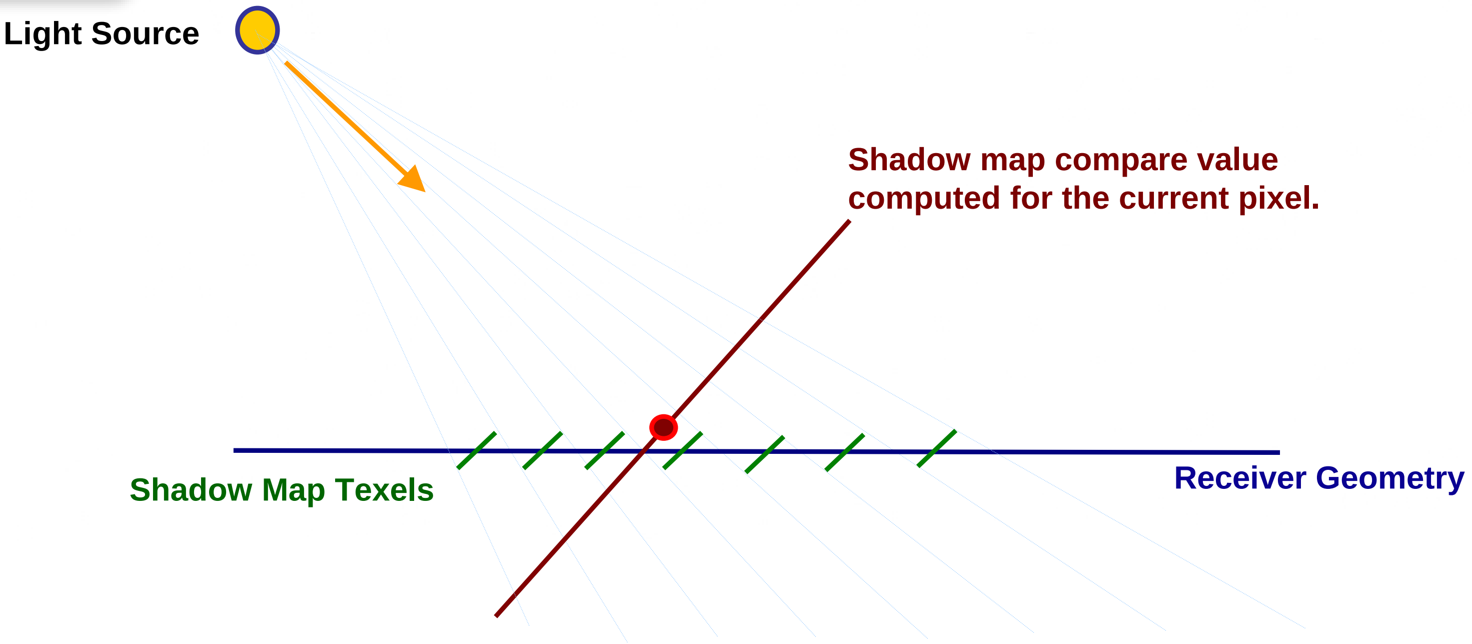
# Slope Based Bias

- Use Z bias for DF16 and DF24 formats
- Couls use gradients to compute bias in PS

```
ddistdx = ddx(dist);
ddistdy = ddy(dist);
dist += g_fSlopeBias * abs(ddistdx);
dist += g_fSlopeBias * abs(ddistdy);
```

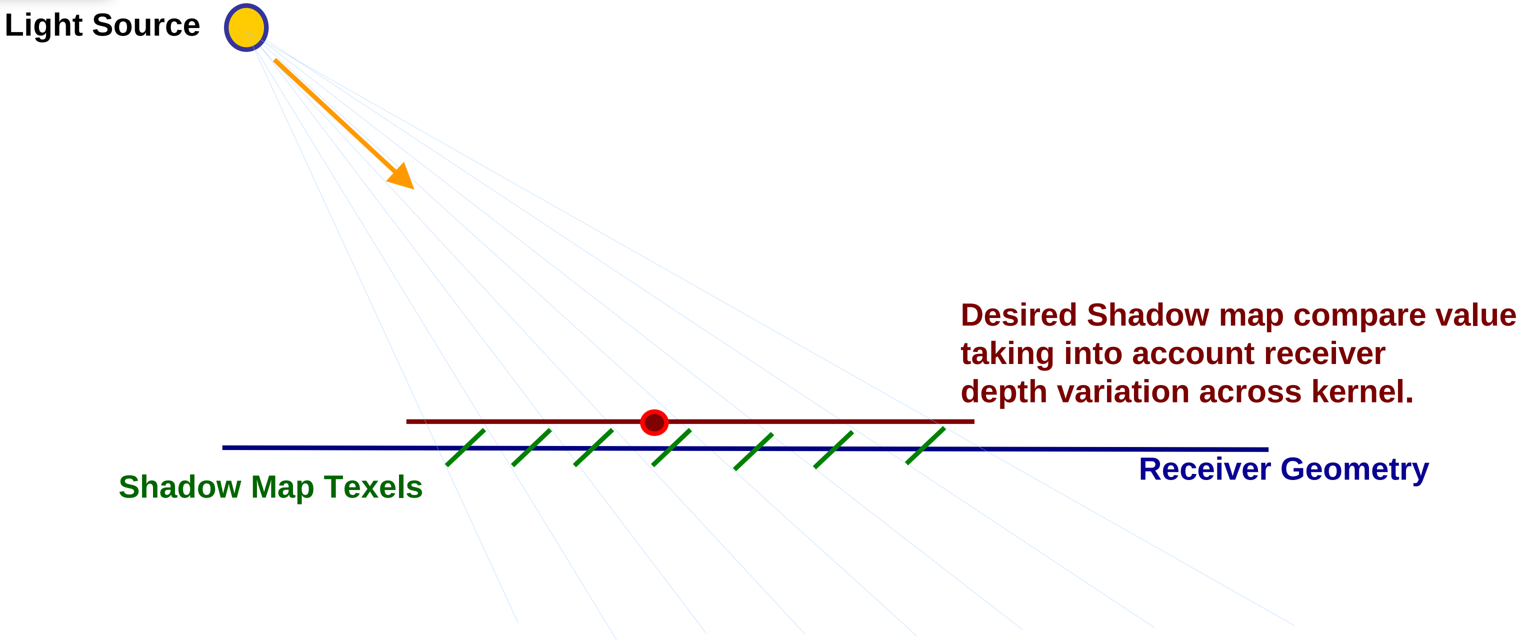
- Large kernels could exhibit surface acne and disconnects at the same time
  - Standard biasing strategy breaks down...

# Receiver Plane Depth Bias



- For large PCF kernel, using a single depth comparison value across the kernel is insufficient

# Receiver Plane Depth Bias



- Vary depth value across the kernel to match the receiver plane
- Need to know how much the depth changes with respect to shadow map texture coordinates

# Receiver Plane Depth Bias

- Compute texture space Jacobian:
  - Derivative of texture coordinates with respect to screen coordinates
- Use as a transform matrix to find derivative of distance to light source w.r.t. texture coordinates

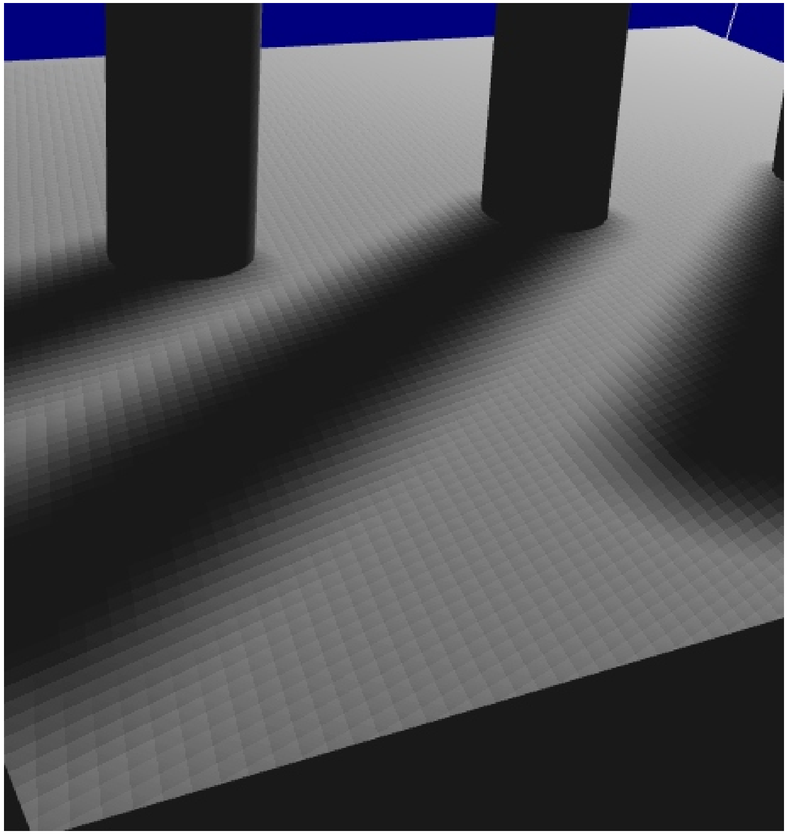
Derivative of distance to light source w.r.t. texture coordinates

Texture space Jacobian (inverse-transpose)

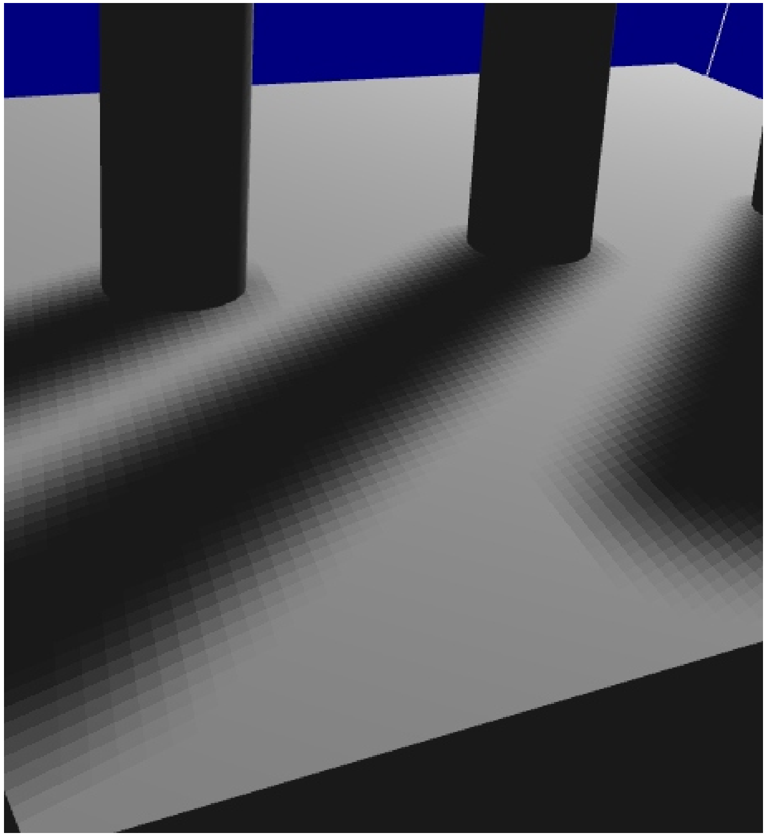
Derivative of distance to light source w.r.t. screen coordinates

$$\begin{bmatrix} \frac{\partial d}{\partial u} \\ \frac{\partial d}{\partial v} \end{bmatrix} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} \end{bmatrix}^{-T} \begin{bmatrix} \frac{\partial d}{\partial x} \\ \frac{\partial d}{\partial y} \end{bmatrix}$$

# Receiver Plane Depth Bias



8x8 PCF without adjustment



8x8 PCF with receiver plane depth bias

# Implementation

```
//Packing derivatives of u,v, and distance to light source w.r.t. screen space x, and y
duvdist_dx = ddx(projCoords);
duvdist_dy = ddy(projCoords);

//Invert texture Jacobian and use chain rule to compute ddist/du and ddist/dv
//  |ddist/du| = |du/dx  du/dy|-T  * |ddist/dx|
//  |ddist/dv|   |dv/dx  dv/dy|      |ddist/dy|

//Multiply ddist/dx and ddist/dy by inverse transpose of Jacobian
float invDet = 1 / ((duvdist_dx.x * duvdist_dy.y) - (duvdist_dx.y * duvdist_dy.x) );

//Top row of 2x2
ddist_duv.x = duvdist_dy.y * duvdist_dx.w ;   // invJtrans[1][1] * ddist_dx
ddist_duv.x -= duvdist_dx.y * duvdist_dy.w ;  // invJtrans[1][2] * ddist_dy

//Bottom row of 2x2
ddist_duv.y = duvdist_dx.x * duvdist_dy.w ;   // invJtrans[2][2] * ddist_dy
ddist_duv.y -= duvdist_dy.x * duvdist_dx.w ;  // invJtrans[2][1] * ddist_dx
ddist_duv *= invDet;

//compute depth offset and PCF taps 4 at a time
for(int i=0; i<9; i++)
{
   //offset of texel quad in texture coordinates;
   texCoordOffset = (g_vFullTexelOffset * quadOffsets[i] );
   //shadow map values
   shadowMapVals = tex2D(ShadowSampler, projCoords.xy + texCoordOffset.xy );

   //Apply receiver plane depth offset
   dist = projCoords.w + (ddist_duv.x * texCoordOffset.x) + (ddist_duv.y * texCoordOffset.y);

   inLight = ( dist < shadowMapVals );
   percentInLight += dot(inLight, invNumTaps);
}
```

# Conclusions

- Radeon X1000 tricks and optimizations
- HDR on Radeon X1000
- Shadow mapping on Radeon X1000

# Questions