Pere-Pau Vázquez – ViRVIG Group, UPC

Percentage Closer Filtering and Percentage Closer Soft Shadows

Overview of Shadow Mapping

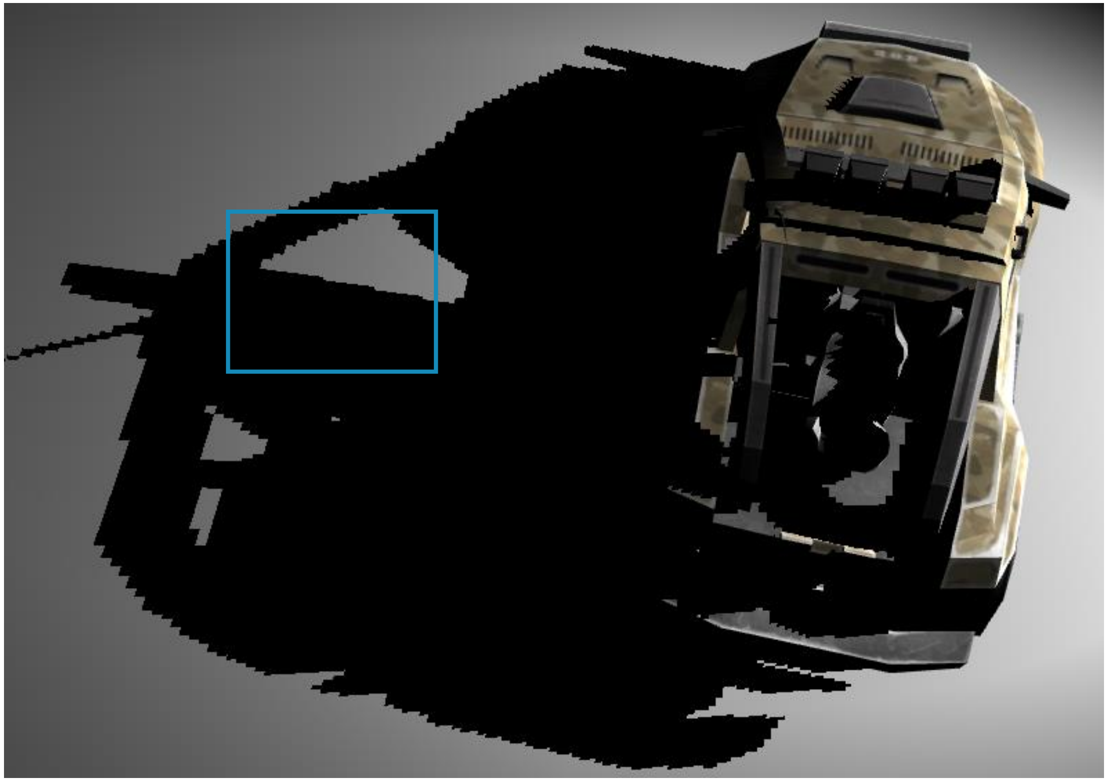
- Introduced by Williams in 1978
- Advantages compared to shadow volumes:
  - Cost less sensitive to geometric complexity
  - Can be queried at arbitrary locations
  - Often easier to implement
- Disadvantages:
  - Aliasing

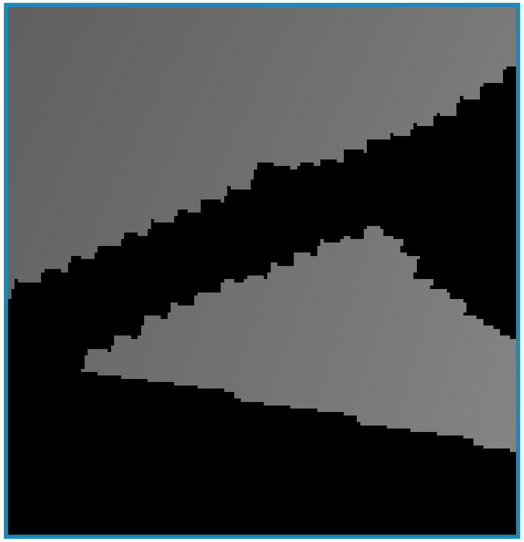
Shadow Mapping Algorithm

- Render scene from light’s point of view
  - Store depth of each pixel
- When shading a surface:
  - Transform surface point into light coordinates
  - Compare current surface depth to stored depth
  - If depth > stored depth, the pixel is in shadow; otherwise the pixel is lit

Aliasing Artifacts

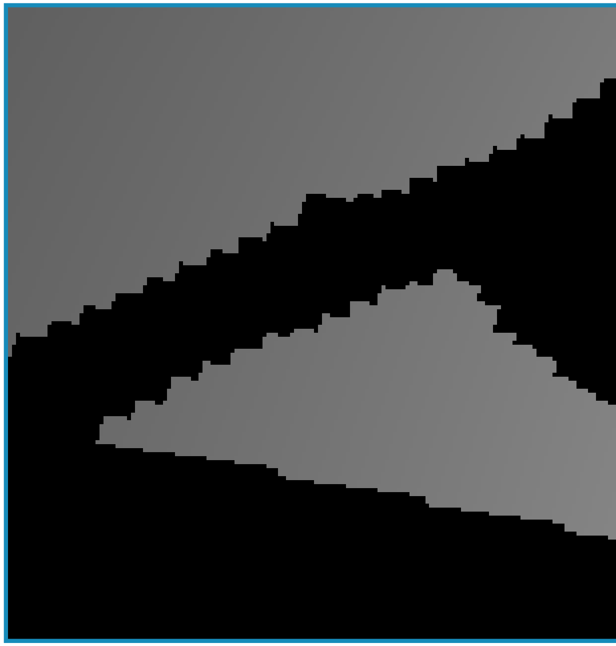
- Magnification artifacts





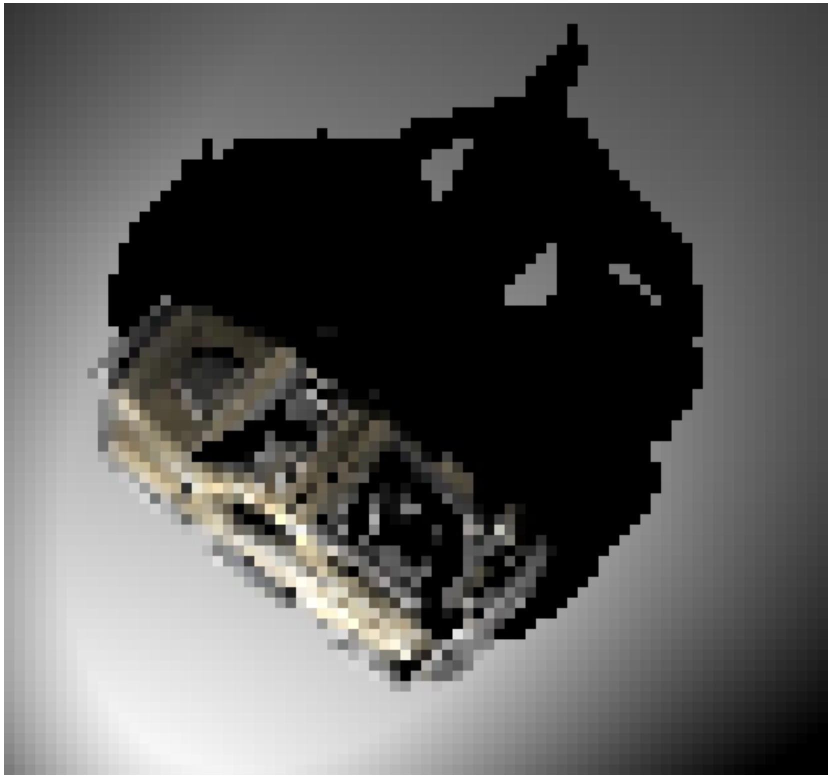
Aliasing Artifacts

- Magnification artifacts
  - Can be addressed to some extent by increasing shadow map resolution. However maximum texture size limits are reached quickly, and are still inadequate (and needlessly expensive) for a large scene.
  - This is a 512x512 shadow map



Aliasing Artifacts

- Minification artifacts



- Typically encountered when viewed from a distance
- Produces ugly and distracting “swimming” effect along shadow edges

Aliasing Artifacts

- Anisotropic artifacts
  - A mix of minification and magnification
  - Encountered at shallow angles

Solutions?

- Also encountered with colour textures
- Reduce aliasing by hardware filtering
  - Magnification artifacts => linear interpolation
  - Minification artifacts => trilinear, mipmapping
  - Anisotropic artifacts => anisotropic filtering

Solutions?

- Can we apply these to shadow maps?
  - Not at the moment
- Interpolating depths is incorrect
  - Gives depth < average(occluder_depth)
  - Want average(depth < occluder_depth)

Percentage Closer Filtering

- Proposed by Reeves et al. in 1987
- Filter result of the depth comparison
  - Sample surrounding shadow map pixels
  - Do a depth comparison for each pixel
  - Percentage lit is the percentage of pixels that pass the depth comparison (i.e. are “closer” than the nearest occluder)
- NVIDIA hardware support for bilinear PCF
- Good results, but can be expensive!

Percentage Closer Filtering

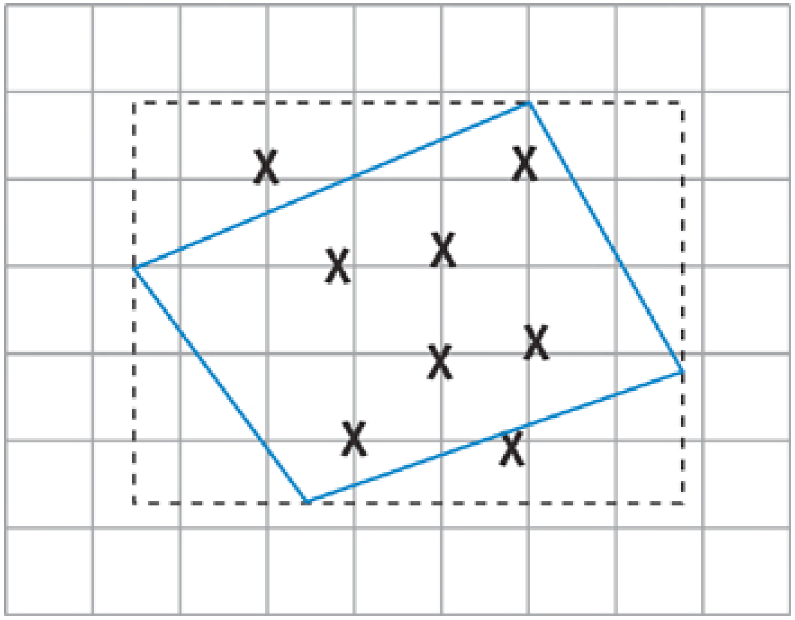
- Approaches
  - Percentage-Closer Filtering
  - Jittered Percentage-Closer Filtering

Percentage Closer Filtering

- Percentage-Closer Filtering:
  - Normal shadow maps present aliasing
    - Shadow map textures cannot be prefiltered
  - Filtering can be achieved averaging multiple shadow map comparisons per pixel
  - It calculates the percentage of the surface that is closer to the light:
    - Not in shadow

Percentage Closer Filtering

- Percentage-Closer Filtering:
  - Original PCF [Reeves et al. 1987] sampled region to be shaded stochastically (≈ randomly)
  - First implemented using the REYES rendering engine



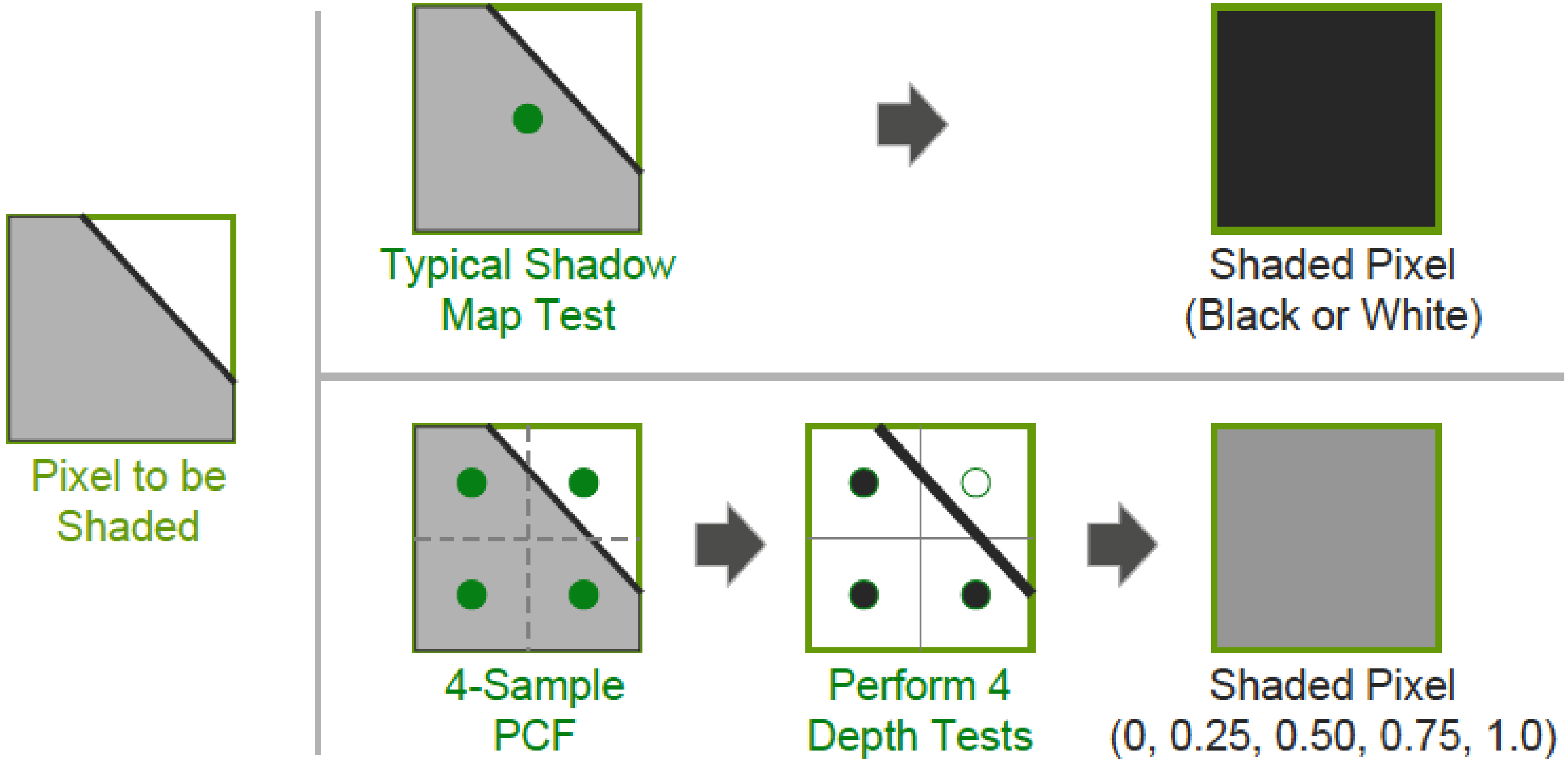
Percentage Closer Filtering

- Percentage-Closer Filtering:
  - GPU implementation means sampling shadow map sampling
    - 4x4 sampling:

0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	1	1
0	0	0	0	0	1	1	1	1
0	0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1	1
0	0	0	0	1	1	1	1	1
1	1	1	1	1	1	1	1	1

Percentage Closer Filtering

- Percentage-Closer Filtering:



Percentage Closer Filtering

- Percentage-Closer Filtering implementation:
  - Size of the sampled region passed through an uniform parameter (*fwidth*).
  - The larger *fwidth* the larger the penumbra region (though computed with the same number of samples)
  - Number of samples now is a constant

Percentage Closer Filtering

```
#define SAMPLES_COUNT 32
#define INV_SAMPLES_COUNT (1.0f / SAMPLES_COUNT)

uniform sampler2D decal;  // decal texture
uniform sampler2D spot;   // projected spotlight image
uniform sampler2DShadow shadowMap;  // shadow map

uniform float fwidth;
uniform vec2 offsets[SAMPLES_COUNT];

// these are passed down from vertex shader
varying vec4 shadowMapPos;
varying vec3 normal;
varying vec2 texCoord;
varying vec3 lightVec;
varying vec3 view;
```

Percentage Closer Filtering

```
#define SAMPLES_COUNT 32
#define INV_SAMPLES_COUNT (1.0f / SAMPLES_COUNT)

uniform sampler2D decal;  // decal texture
uniform sampler2D spot;   // projected spotlight image
uniform sampler2DShadow shadowMap;  // shadow map

uniform float fwidth;
uniform vec2 offsets[SAMPLES_COUNT];

// these are passed down from vertex shader
varying vec4 shadowMapPos;
varying vec3 normal;
varying vec2 texCoord;
varying vec3 lightVec;
varying vec3 view;
```

Percentage Closer Filtering

```
#define SAMPLES_COUNT 32
#define INV_SAMPLES_COUNT (1.0f / SAMPLES_COUNT)

uniform sampler2D decal;  // decal texture
uniform sampler2D spot;   // projected spotlight image
uniform sampler2DShadow shadowMap;  // shadow map

uniform float fwidth;
uniform vec2 offsets[SAMPLES_COUNT];

// these are passed down from vertex shader
varying vec4 shadowMapPos;
varying vec3 normal;
varying vec2 texCoord;
varying vec3 lightVec;
varying vec3 view;
```

Percentage Closer Filtering

```
void main(void)
{
  float shadow = 0;
  float fsize = shadowMapPos.w * fWidth;
  vec4 smCoord = shadowMapPos;

  for (int i = 0; i<SAMPLES_COUNT; i++) {
    smCoord.xy = offsets[i] * fsize + shadowMapPos;
    shadow += texture2DProj(shadowMap, smCoord) * INV_SAMPLES_COUNT;
  }
```

Percentage Closer Filtering

```
void main(void)
{
  float shadow = 0;
  float fsize = shadowMapPos.w * fwidth;
  vec4 smCoord = shadowMapPos;

  for (int i = 0; i<SAMPLES_COUNT; i++) {
    smCoord.xy = offsets[i] * fsize + shadowMapPos;
    shadow += texture2DProj(shadowMap, smCoord) * INV_SAMPLES_COUNT;
  }

  vec3 N = normalize(normal);
```

Percentage Closer Filtering

```
void main(void)
{
  float shadow = 0;
  float fsize = shadowMapPos.w * fwidth;
  vec4 smCoord = shadowMapPos;

  for (int i = 0; i<SAMPLES_COUNT; i++) {
    smCoord.xy = offsets[i] * fsize + shadowMapPos;
    shadow += texture2DProj(shadowMap, smCoord) * INV_SAMPLES_COUNT;
  }

  vec3 N = normalize(normal);
```

Percentage Closer Filtering

```
void main(void)
{
  float shadow = 0;
  float fsize = shadowMapPos.w * fwidth;
  vec4 smCoord = shadowMapPos;

  for (int i = 0; i<SAMPLES_COUNT; i++) {
    smCoord.xy = offsets[i] * fsize + shadowMapPos;
    shadow += texture2DProj(shadowMap, smCoord) * INV_SAMPLES_COUNT;
  }

  vec3 N = normalize(normal);
```

Percentage Closer Filtering

```
void main(void)
{
  float shadow = 0;
  float fsize = shadowMapPos.w * fwidth;
  vec4 smCoord = shadowMapPos;

  for (int i = 0; i<SAMPLES_COUNT; i++) {
    smCoord.xy = offsets[i] * fsize + shadowMapPos;
    shadow += texture2DProj(shadowMap, smCoord) * INV_SAMPLES_COUNT;
  }

  vec3 N = normalize(normal);
```

Percentage Closer Filtering

```
  for (int i = 0; i<SAMPLES_COUNT; i++) {
    smCoord.xy = offsets[i] * fsize + shadowMapPos;
    shadow += texture2DProj(shadowMap, smCoord) * INV_SAMPLES_COUNT;
  }

  vec3 N = normalize(normal);
  vec3 L = normalize(lightVec);
  vec3 V = normalize(view);
  vec3 R = reflect(-V, N);

  // calculate diffuse dot product
  float NdotL = max(dot(N, L), 0);

  // modulate lighting with the computed shadow value
  vec3 color = texture2D(decal, texCoord).xyz;

  gl_FragColor.xyz = (color * NdotL + pow(max(dot(R, L), 0), 64)) *
                     shadow * texture2DProj(spot, shadowMapPos) +
                     color * 0.1;
}
```

Percentage Closer Filtering

```
    for (int i = 0; i<SAMPLES_COUNT; i++) {
      smCoord.xy = offsets[i] * fsize + shadowMapPos;
      shadow += texture2DProj(shadowMap, smCoord) * INV_SAMPLES_COUNT;
    }

    vec3 N = normalize(normal);
    vec3 L = normalize(lightVec);
    vec3 V = normalize(view);
    vec3 R = reflect(-V, N);

    // calculate diffuse dot product
    float NdotL = max(dot(N, L), 0);

    // modulate lighting with the computed shadow value
    vec3 color = texture2D(decal, texCoord).xyz;

    gl_FragColor.xyz = (color * NdotL + pow(max(dot(R, L), 0), 64)) *
                       shadow * texture2DProj(spot, shadowMapPos) +
                       color * 0.1;
}
```

Percentage Closer Filtering

```
for (int i = 0; i<SAMPLES_COUNT; i++) {
  smCoord.xy = offsets[i] * fsize + shadowMapPos;
  shadow += texture2DProj(shadowMap, smCoord) * INV_SAMPLES_COUNT;
}

vec3 N = normalize(normal);
vec3 L = normalize(lightVec);
vec3 V = normalize(view);
vec3 R = reflect(-V, N);

// calculate diffuse dot product
float NdotL = max(dot(N, L), 0);

// modulate lighting with the computed shadow value
vec3 color = texture2D(decal, texCoord).xyz;

gl_FragColor.xyz = (color * NdotL + pow(max(dot(R, L), 0), 64)) *
                   shadow * texture2DProj(spot, shadowMapPos) +
                   color * 0.1;
}
```

Percentage Closer Filtering

- Results:

Percentage Closer Filtering

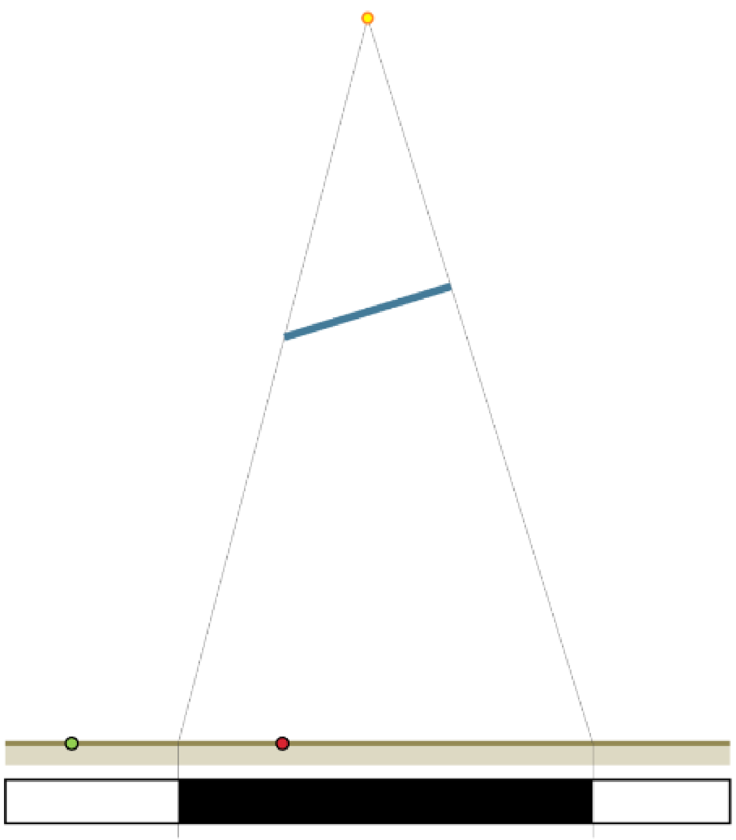
- Percentage-Closer Filtering.
  - Fast due to the spatial coherence of texture texels
    - Many queries benefit from texture cache
  - Modern GPUs inherently implement a 2x2 PCF for shadow map queries
    - Improvement in shadow mapping-based silhouettes

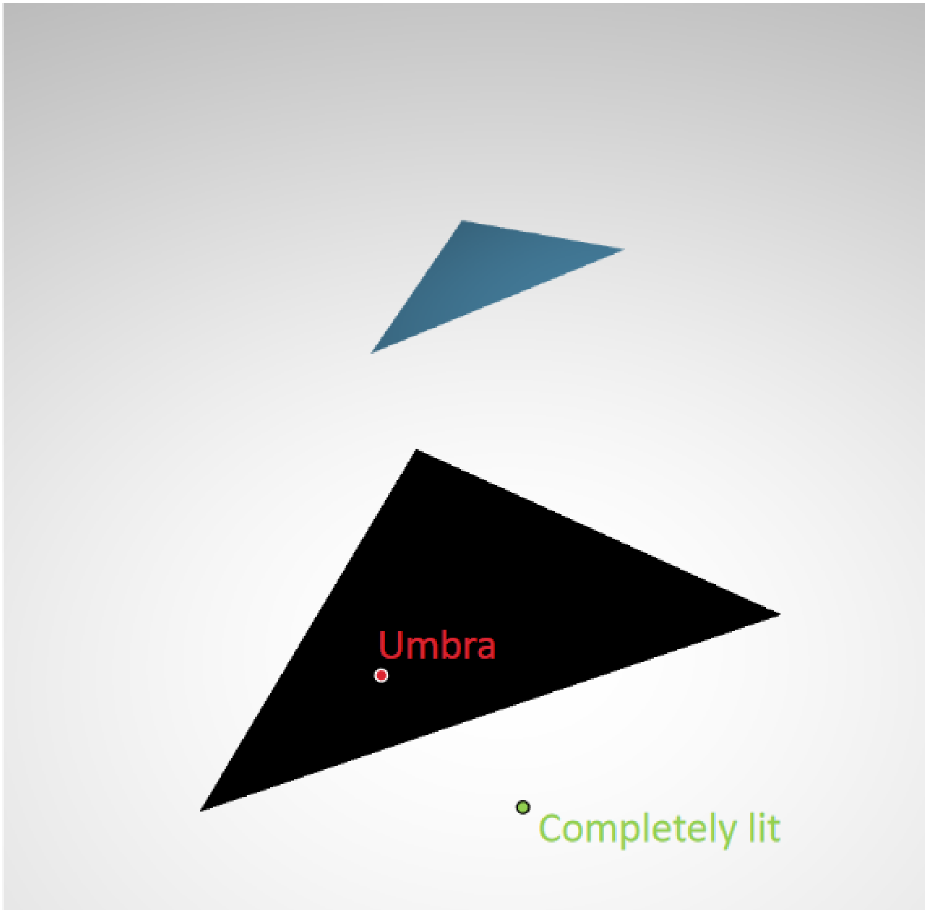
Percentage Closer Filtering

- Percentage-Closer Filtering. Modified region:

Percentage Closer Filtering

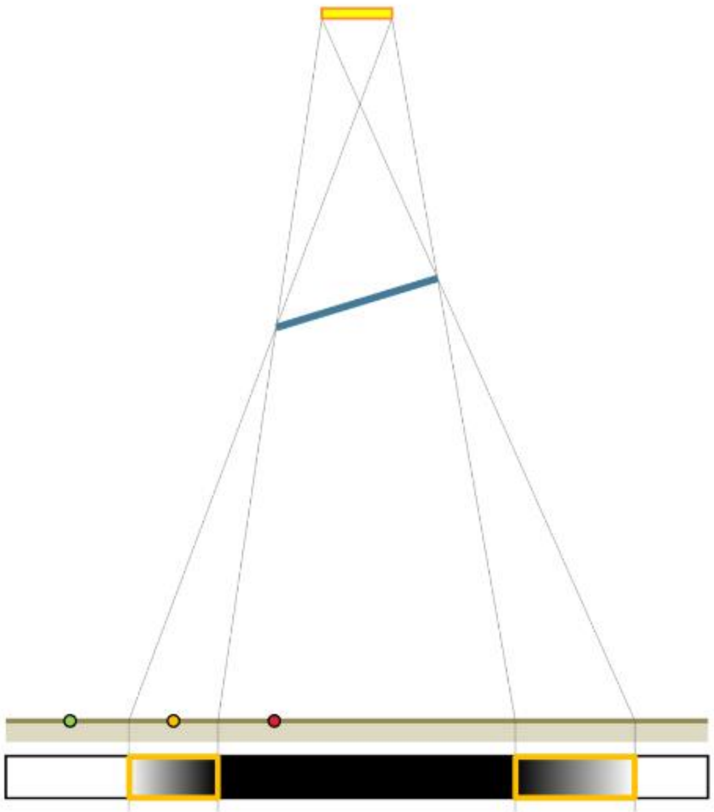
- Recap: hard shadows vs soft shadows

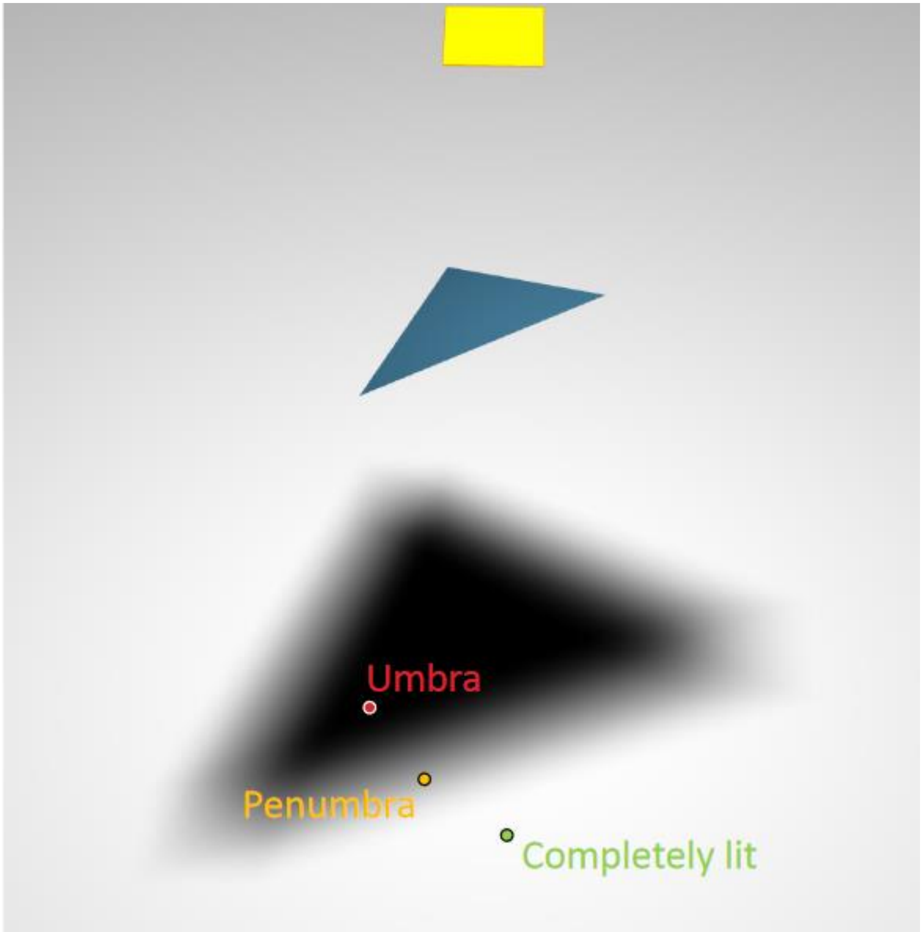




Percentage Closer Filtering

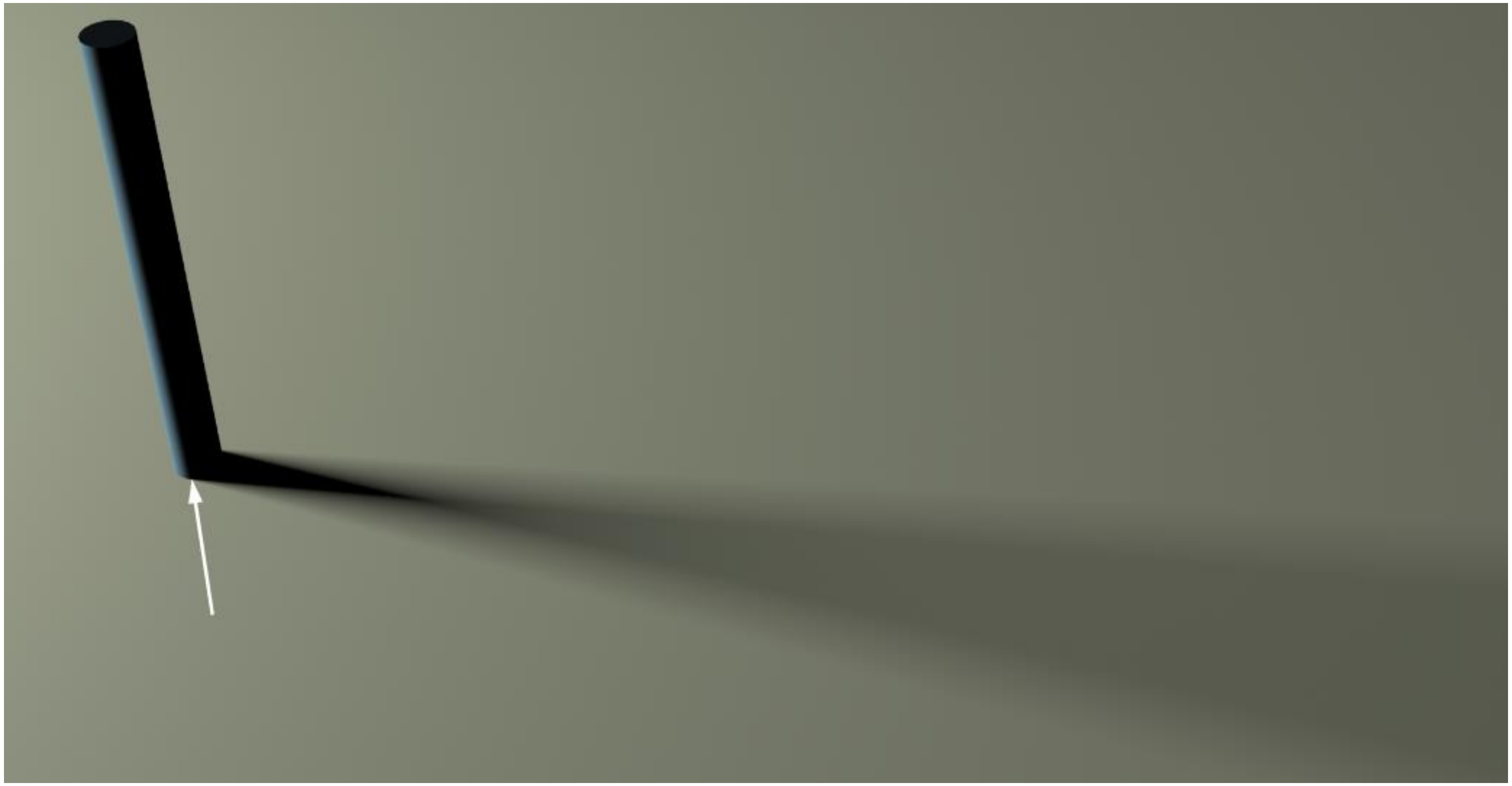
- Recap: hard shadows vs soft shadows





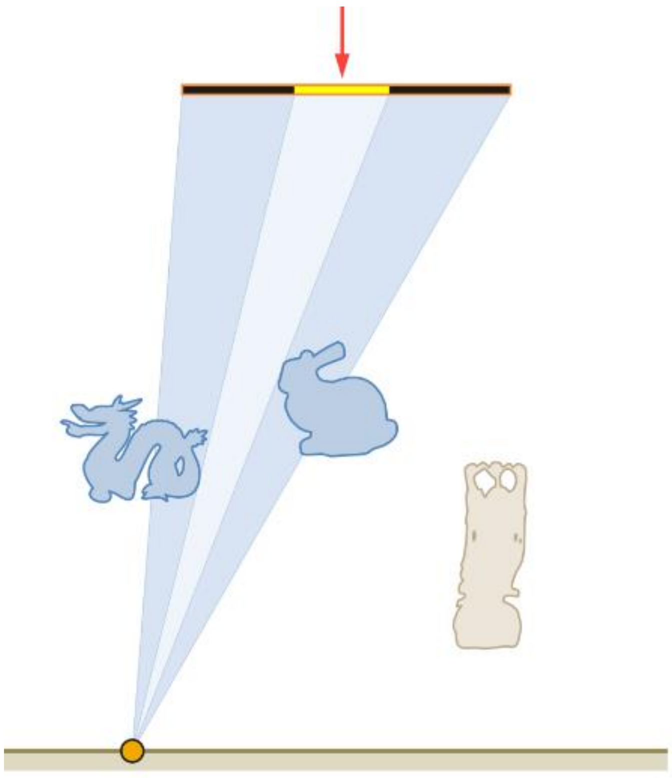
Percentage Closer Filtering

- Recap: hard shadows vs soft shadows
  - Shadow hardening on contact



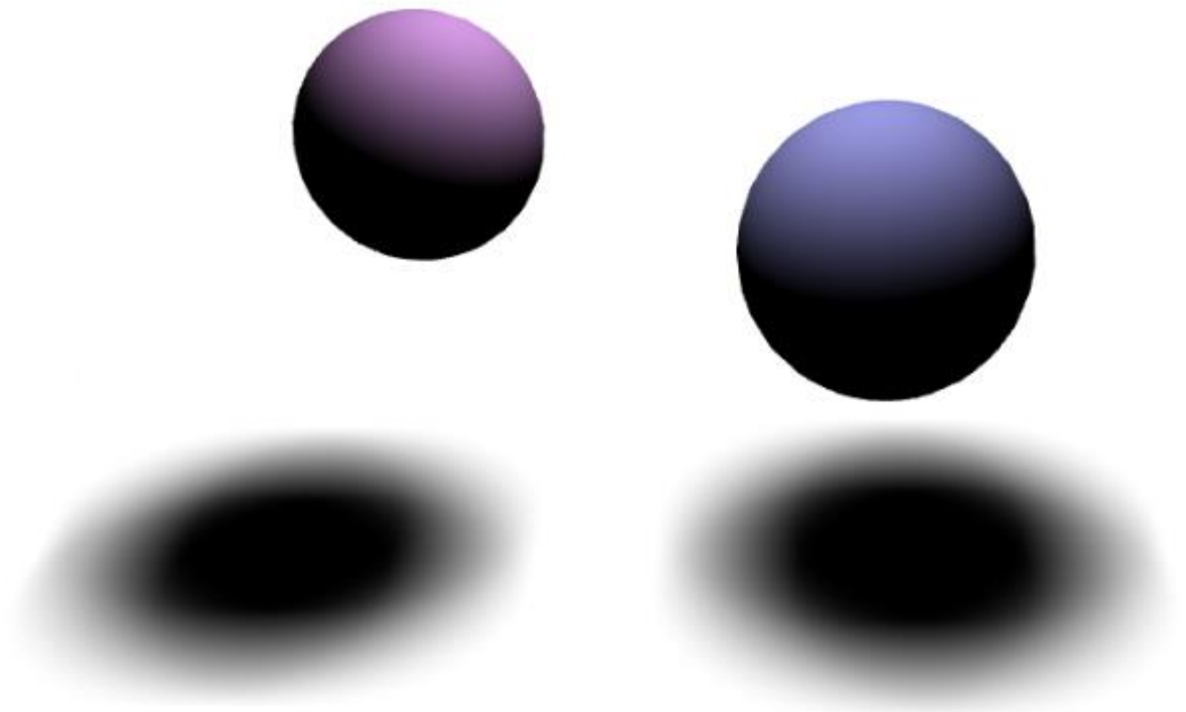
Percentage Closer Filtering

- Recap: soft shadows is a point-region visibility calculation
  - For each receiver sample (point)
  - determine visible fraction of light source(region)



Percentage Closer Filtering

- Original PCF generates a soft-shadow-like appearance
  - But ignoring penumbra width



Percentage Closer Filtering

- PCF implementation shown adapts penumbra width through the use of *fwidth* parameter

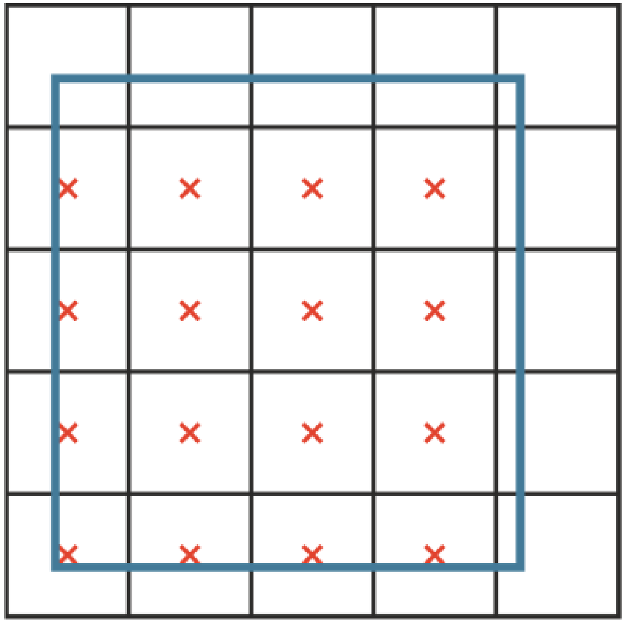
```
void main(void)
{
  float shadow = 0;
  float fsize = shadowMapPos.w * fwidth;
  vec4 smCoord = shadowMapPos;

  for (int i = 0; i<SAMPLES_COUNT; i++) {
    smCoord.xy = offsets[i] * fsize + shadowMapPos;
    shadow += texture2DProj(shadowMap, smCoord) * INV_SAMPLES_COUNT;
  }
```

Percentage Closer Filtering

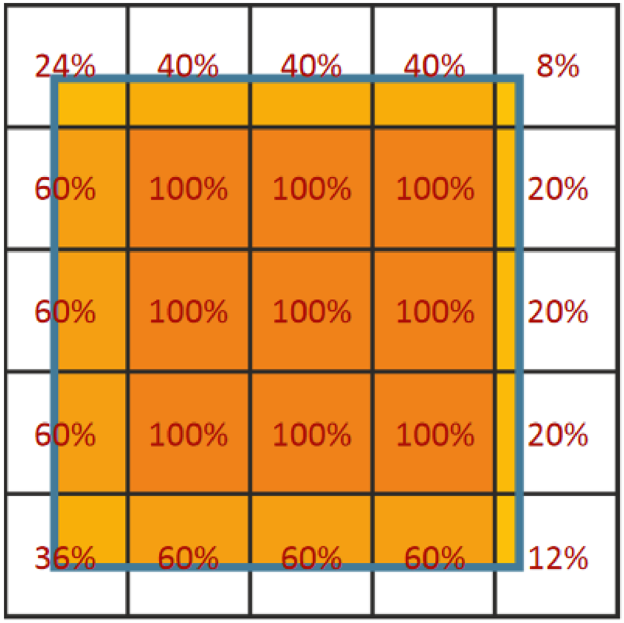
- PCF + large kernel + width dependency = Percentage Closer Soft Shadows (PCSS)

Percentage Closer Filtering

- PCF issues: Oversimplified sampling



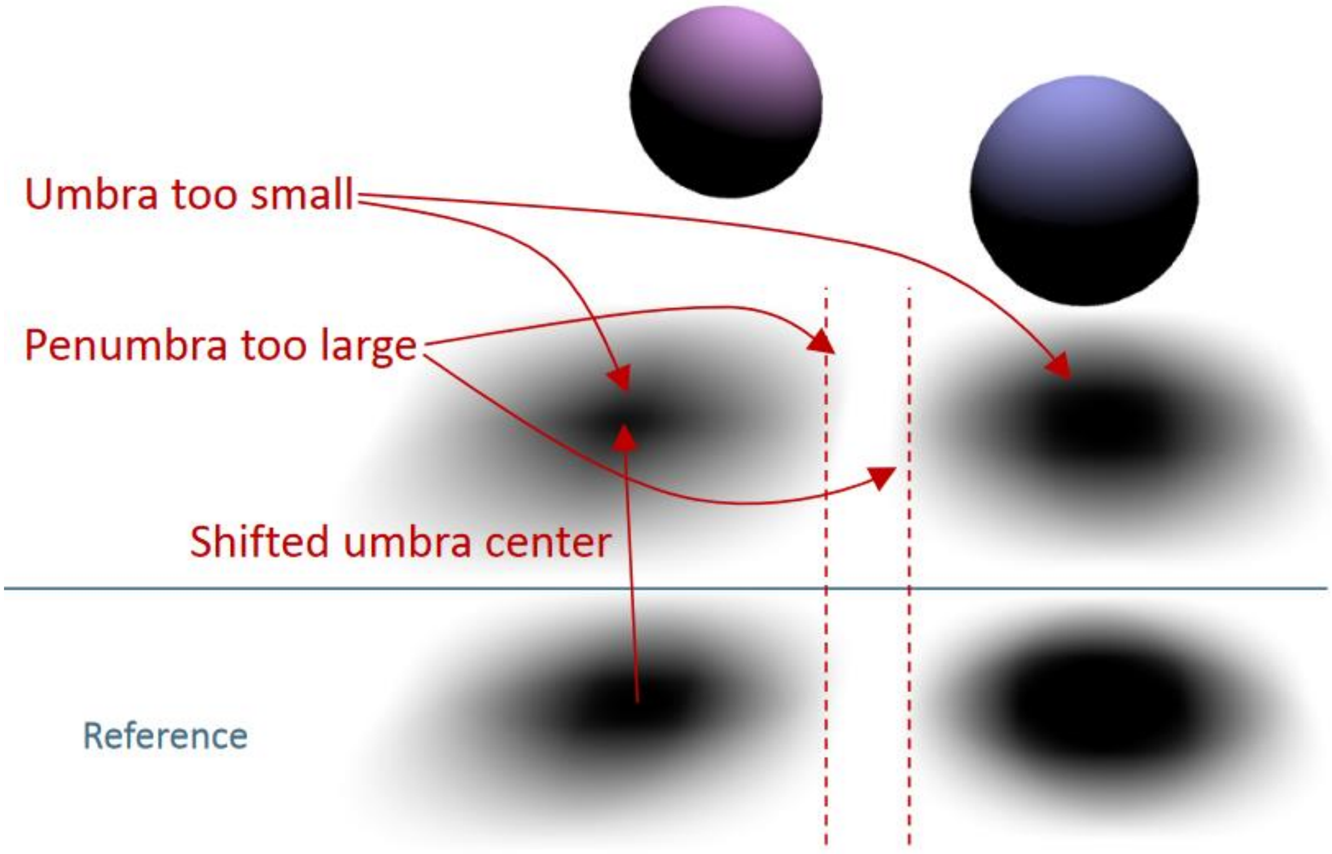
PCF: samples weighted equally



Analytic: actual coverage
[Shen et al., 2011]

Percentage Closer Filtering

- PCSS issues

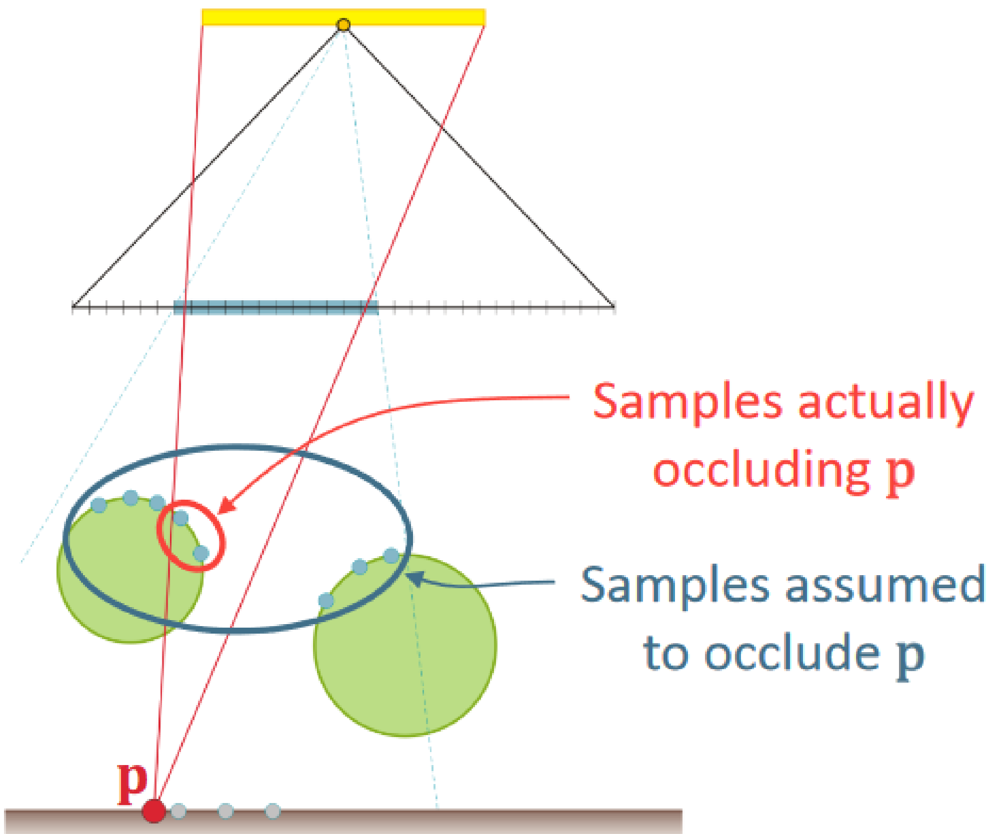


Main sources of incorrectness

- Single planar occluder assumption
- Classification as light blocking solely based on depth test

Percentage Closer Filtering

- PCSS issues



Main sources of incorrectness

- Single planar occluder assumption
- Classification as light blocking solely based on depth test

Percentage Closer Filtering

- PCSS conclusions:
  - Simple and reasonably fast
  - Often visually pleasing results (for smaller light sources)
  - Not really physically correct
  - Only accounts for occluders visible from light source’s center

Percentage Closer Filtering

- Jittered Percentage-Closer Filtering. Motivation:
  - Though PCF achieves soft shadows, banding artifacts still present
  - An arbitrarily large kernel will soften the shadow even further
    - Might require a high number of samples
    - Still might present banding artifacts due to regular sampling

Percentage Closer Filtering

- Jittered Percentage-Closer Filtering:
  - Sample larger regions in an stochastic manner
    - Use fewer samples taking advantage of:
      - PCF hardware
      - Stochastically placing the samples (i. e. dependent on the image position) reduces banding artifacts
  - If texture is magnified the sampling regions of neighboring pixels will overlap:
    - Smooth transition from unlit to lit
    - No banding artifacts

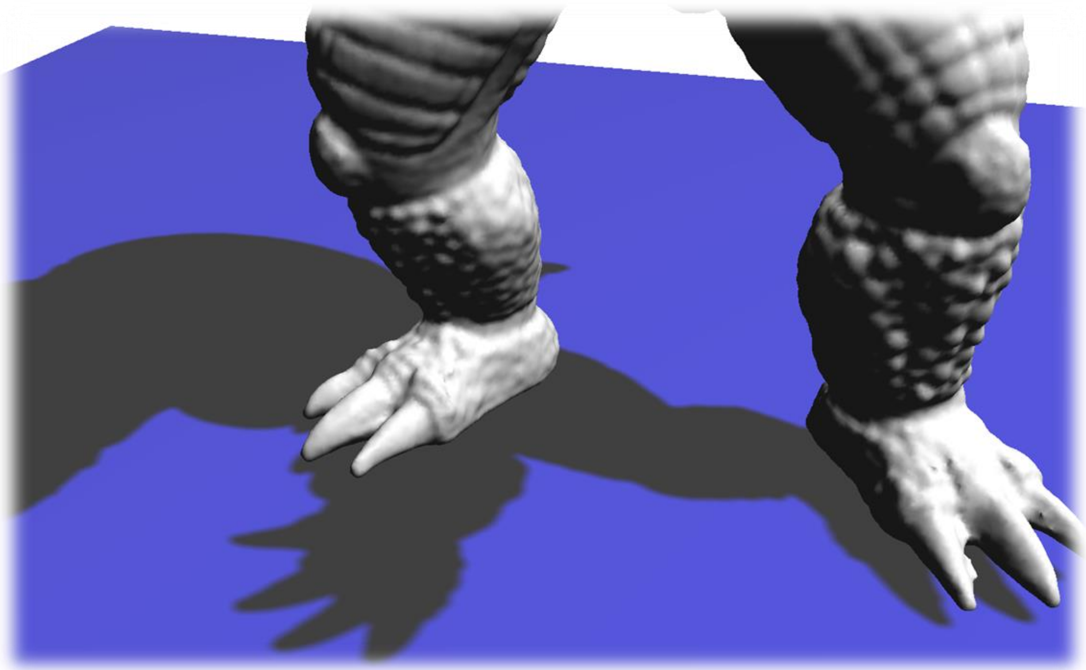
Percentage Closer Filtering

- Jittered Percentage-Closer Filtering. Implementation:
  - Jitter map passed as a 3D texture

```
vec4 smCoord = ShadowCoord;
vec3 jcoord = vec3 (gl_FragCoord.xy*jxyscale, 0.);
float fsize = ShadowCoord.w * fwidth;

for (int i = 0; i < 4; i++)
{
   vec4 offset = texture3D(jitterMap, jcoord)*2. -1.;
   jcoord.z+=1.0f/SAMPLES_COUNT_DIV_2; //0.03125
   smCoord.xy = offset.xy * fsize + ShadowCoord.xy;
   shadow+= lookup_shadowMap(smCoord);
   smCoord.xy = offset.zw * fsize + ShadowCoord.xy;
   shadow+= lookup_shadowMap(smCoord);
 }
```

Percentage Closer Filtering

- Jittered Percentage-Closer Filtering. Results:



Percentage Closer Filtering

- Jittered Percentage-Closer Filtering. PCF vs JPCF: Note the banding artifacts in PCF

PCF vs JPCF

Percentage Closer Filtering

- Sample the result of (d<z) around projected point
  - Filter the binary results in a given kernel

0	1
1	0

- Bilinear PCF
  - NVIDIA and AMD GPUs implement 2x2 PCF in one fetch
  - Using the same location and weights for bilinear filtering

Percentage Closer Filtering

```
Texture2D<float> tDepthMap;

SamplerComparisonState ShadowSampler
{
    ComparisonFunc = LESS;
    Filter = COMPARISON_MIN_MAG_LINEAR_MIP_POINT;
};

    // ...
    sum += tDepthMap.SampleCmpLevelZero(ShadowSampler,
                                        uv + offset, z);
```

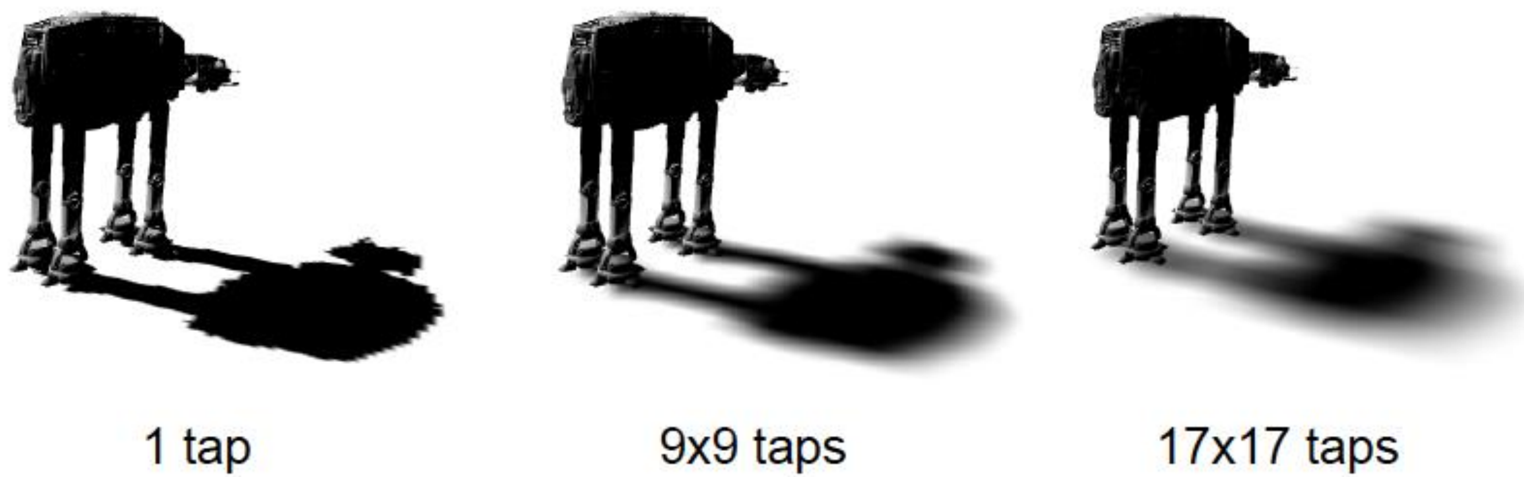
Percentage Closer Filtering

- Texture configuration

```
// Create the FBO
glGenFramebuffers(1, &m_fbo);

// Create the depth buffer
glGenTextures(1, &m_shadowMap);
glBindTexture(GL_TEXTURE_2D, m_shadowMap);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32, WindowWidth, WindowHeight, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_COMPARE_REF_TO_TEXTURE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

glBindFramebuffer(GL_FRAMEBUFFER, m_fbo);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, m_shadowMap, 0);
```

Percentage Closer Filtering

- Increasing the number of PCF taps increases the softness of the shadows



Percentage Closer Filtering

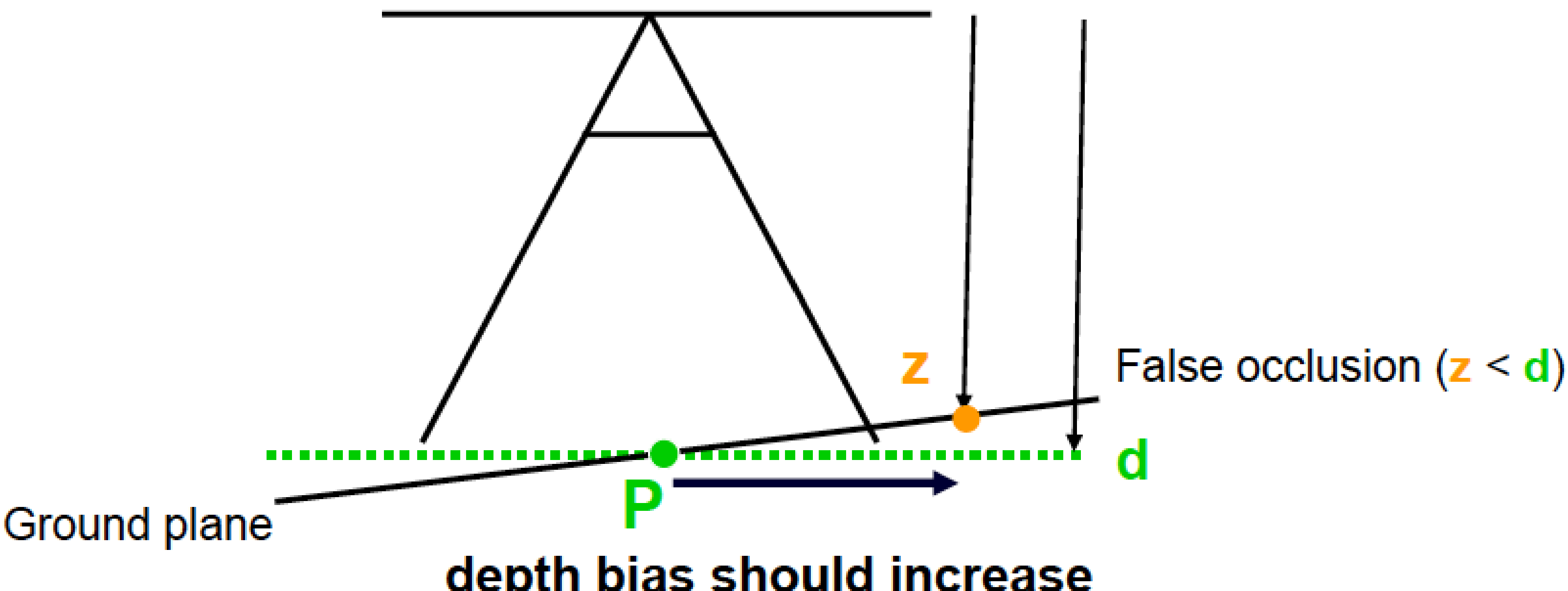
- PCF with large kernels requires many samples
  - Using irregular sampling
    - Trades banding for noise

regular sampling

irregular sampling

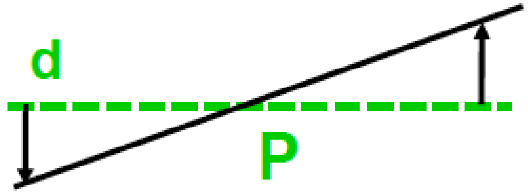
Percentage Closer Filtering. Self-shadowing issue

- Traditional depth bias: write (w+bias) in shadow map
  - Bias = constant bias + slope-based bias
  - Issue: huge depth biases may be required for large PCF kernels

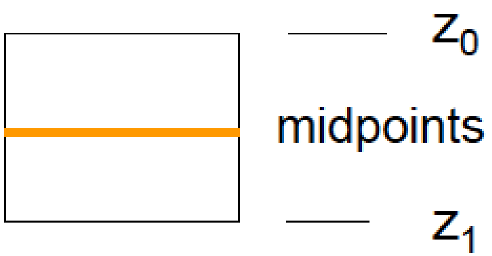


Percentage Closer Filtering

- Use depth gradient = float2(dz/du, dz/dv)
  - Make depth d follow tangent plane
  - d = d0 + dot(uv_offset, gradient)
  - [Schuler06] and [Isidoro 06]



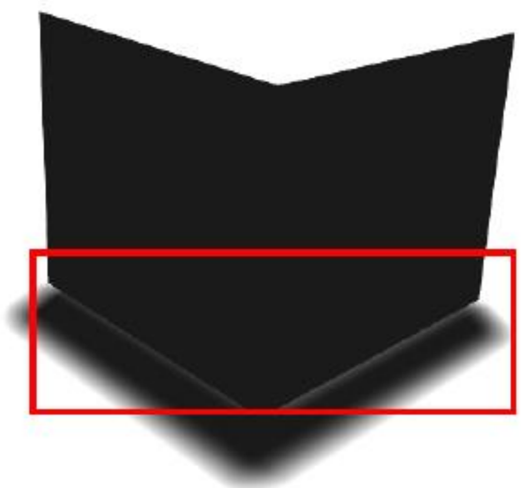
- Render midpoints into shadow map
  - Midpoint z = $(z_0+z_1)/2$
  - Requires two rasterization passes
    - Depth peel two depth layers
  - Still requires a depth bias for thin objects



- Render back faces into shadow map
  - Only works for closed objects
  - Light bleeding for large PCF kernels

Percentage Closer Filtering

- Rendering back faces into shadow map generates light bleeding for large PCF kernels
  - Not due to FP precision or shadow map resolution
  - But reverse of the surface acne issue



Percentage Closer Filtering

- PCF cannot be prefiltered as is
  - Average (d<z) != (Average (z) < d)
  - Filtering the depth buffer would smooth the heighfield of the shadow map
    - Does not generate soft shadows
    - May introduce artifacts
- Solutions: Approximate shadow test by a linear function which can be prefiltered
  - Goal: blurring the shadow map to generate realistic soft shadows

Fast Percentage Closer Soft Shadows using Temporal Coherence

Michael Schwärzler*
VRVis Research Center, Austria

Christian Luksch†
VRVis Research Center, Austria

Daniel Scherzer‡
Max-Planck-Institut für Informatik, Germany

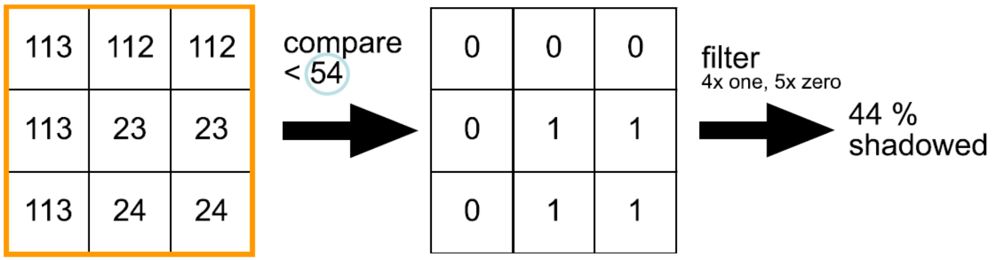
Michael Wimmer§
Vienna University of Technology, Austria

Outline

- Related work
  - Real-time soft shadow mapping
  - Data catching / Temporal coherence
- Their approach
  - Shadow reprojection
  - Detecting moving objects
  - Reconstruction error
- Implementation and evaluation
- Conclusion and future work

Related Work
Real-time soft shadow mapping

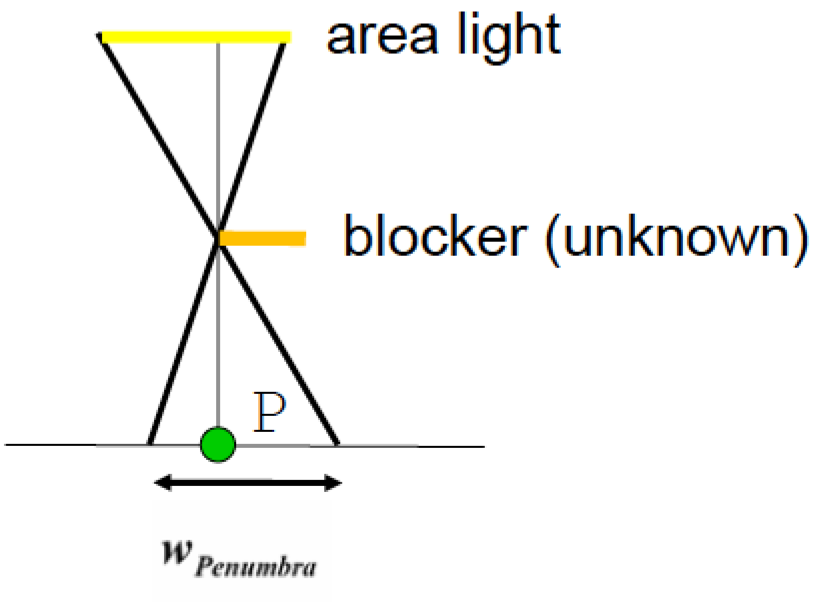
- *Percentage Closer Soft Shadows (PCSS)* [Fernando 2005]
  - Based on *Percentage Closer Filtering* [Reeves et al. 1987]
    - Average comparison results, not depth values



  - But supporting variable kernel sizes
    - To simulate varying penumbra sizes

Percentage Closer Soft Shadows (PCSS)

- PCSS [Fernando05]
- Assume a square light centered at the shadow map center
- Assuming some parallel blocker to receiver
  - Compute penumbra width using similar triangles



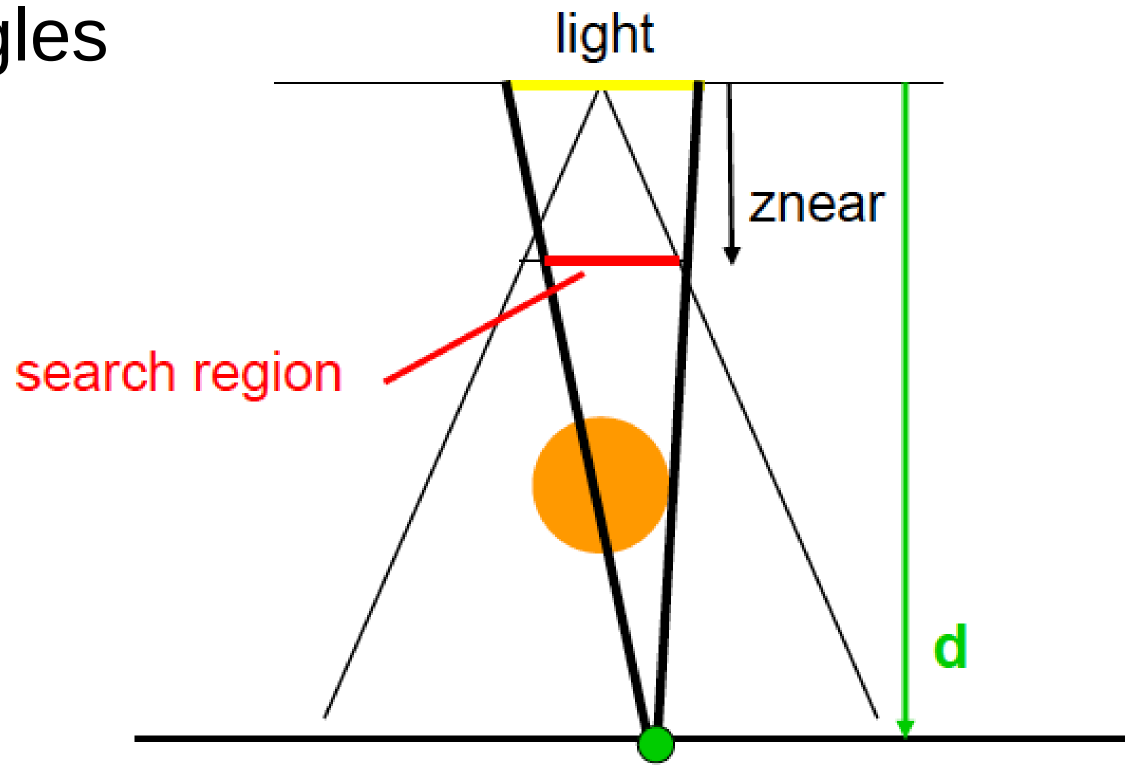
$$w_{Penumbra} = \frac{(d_{Receiver} - d_{Blocker}) \cdot w_{Light}}{d_{Blocker}}$$

Percentage Closer Soft Shadows (PCSS)

- Step 1: Blocker search
  - Sample the depth buffer using point sampling
  - Average all blockers with (depth + bias < receiver) in search region / kernel
  - Early out if no blocker found
- Step 2: Filtering
  - Use filter radius from step 1
  - Clamp filter width to be >= MinRadius for antialiasing
  - Filter the shadow map with PCF or VSM/CSM/ESM

Percentage Closer Soft Shadows (PCSS)

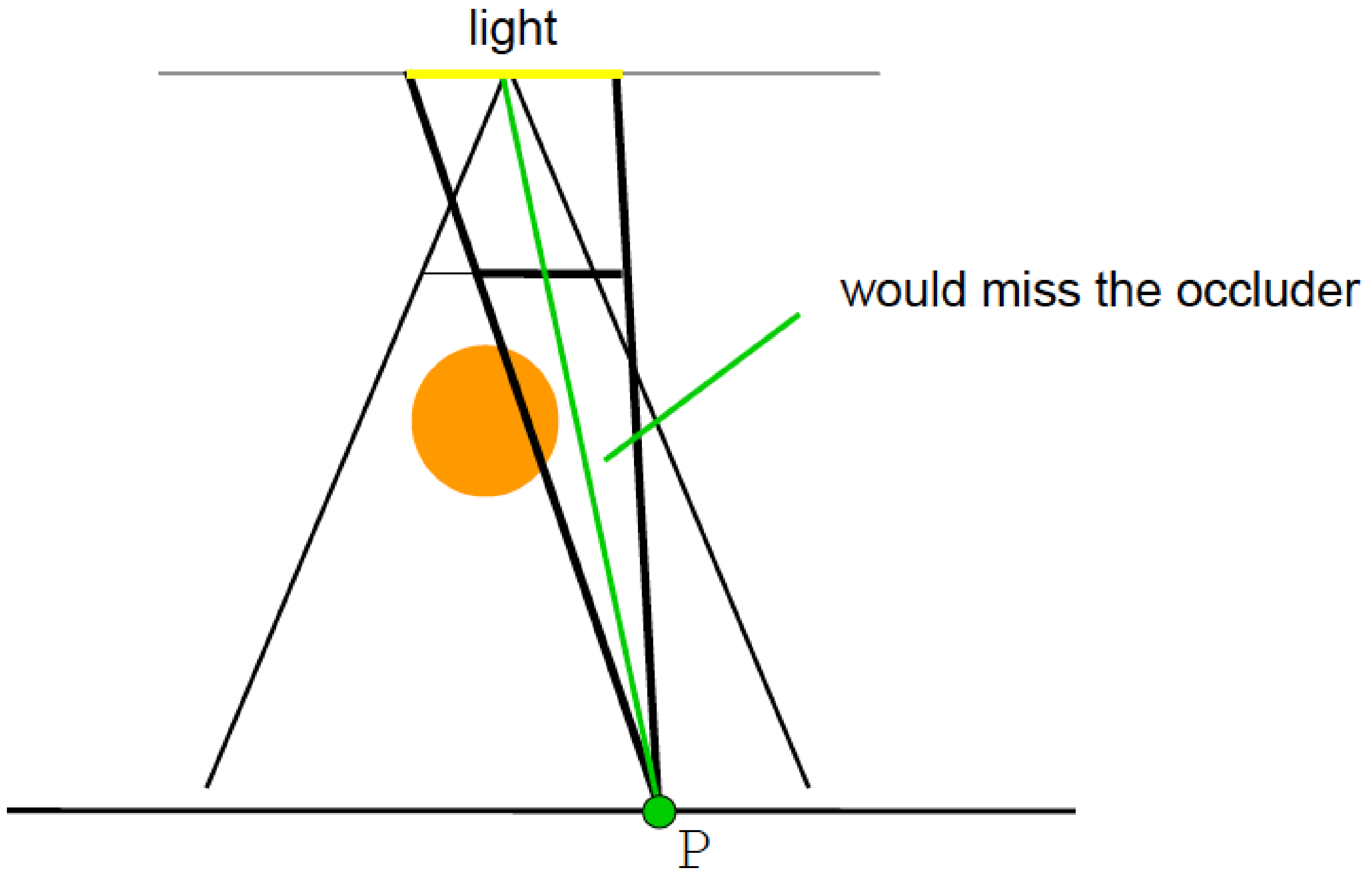
- Where to find blockers?
- Conservative search radius using similar triangles



LightRadius / d = SearchRadius / (d-znear)

Percentage Closer Soft Shadows (PCSS)

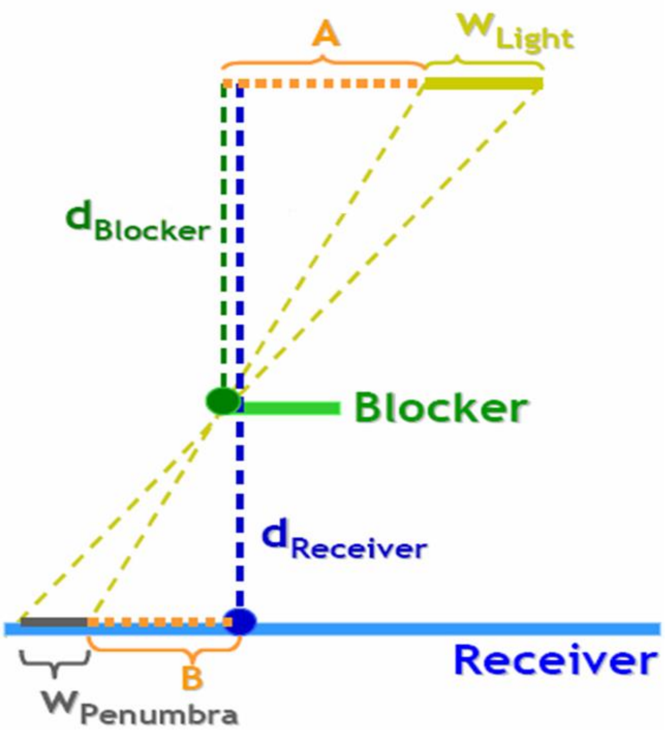
- Why not doing just one sample?



Percentage Closer Soft Shadows (PCSS)

- Step 2: Penumbra estimation
  - assumes that blocker and receiver are planar and in parallel

$$w_{Penumbra} = \frac{(d_{Receiver} - d_{Blocker}) \cdot w_{Light}}{d_{Blocker}}$$



Percentage Closer Soft Shadows (PCSS)

- The more samples in the blocker search, the less noisy artifacts in the soft shadows
  - In practice, 4x4 or 5x5 samples is sufficient

Blocker Search with 3x3 taps

Blocker Search with 5x5 taps

Percentage Closer Soft Shadows (PCSS)

PCSS In Hellgate: London

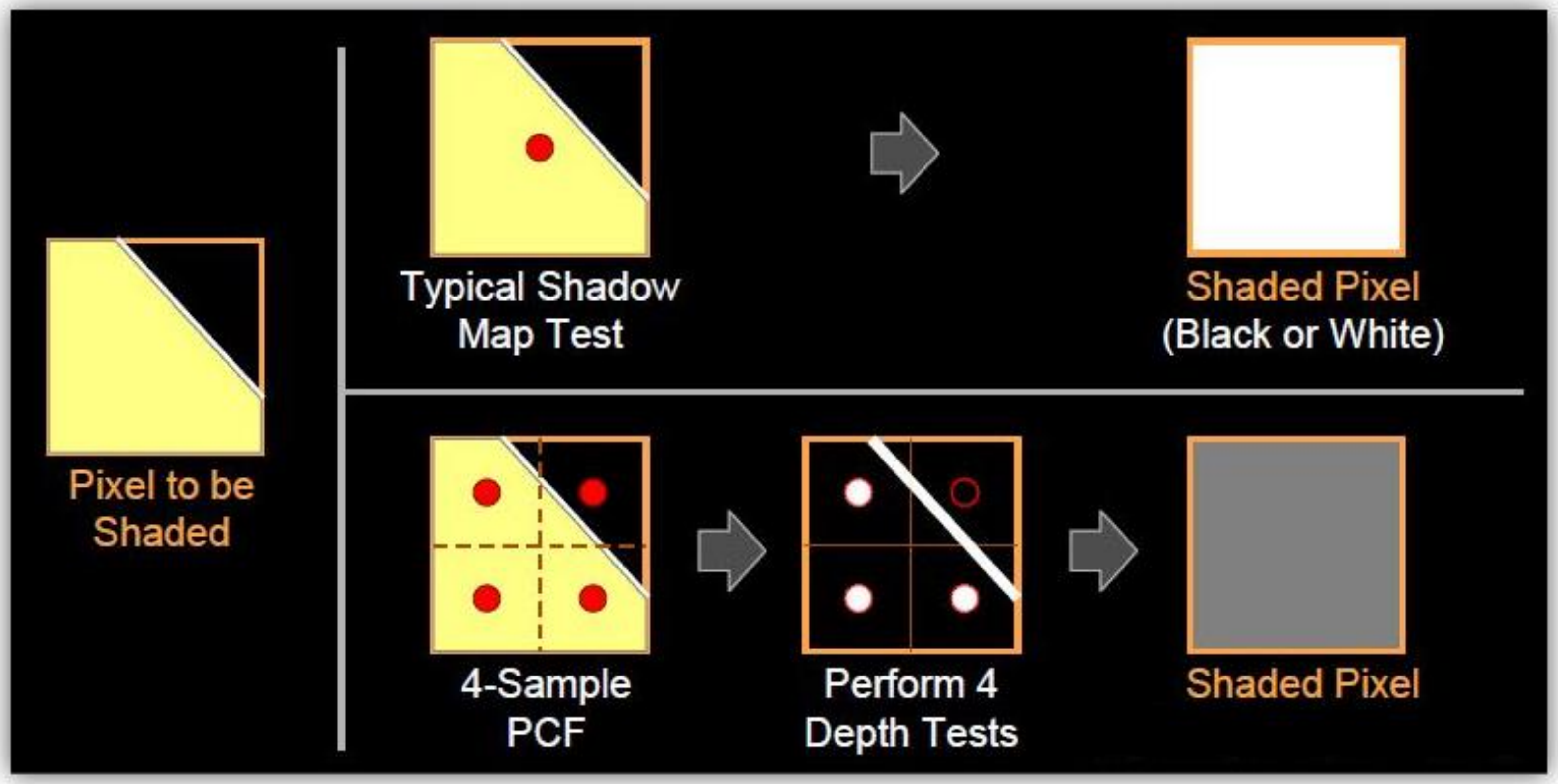
PCSS

PCF

16 POINT taps for the blocker search
16 PCF taps for the PCF filtering

Percentage Closer Soft Shadows (PCSS)

- Step 3: Percentage-Closer Filtering

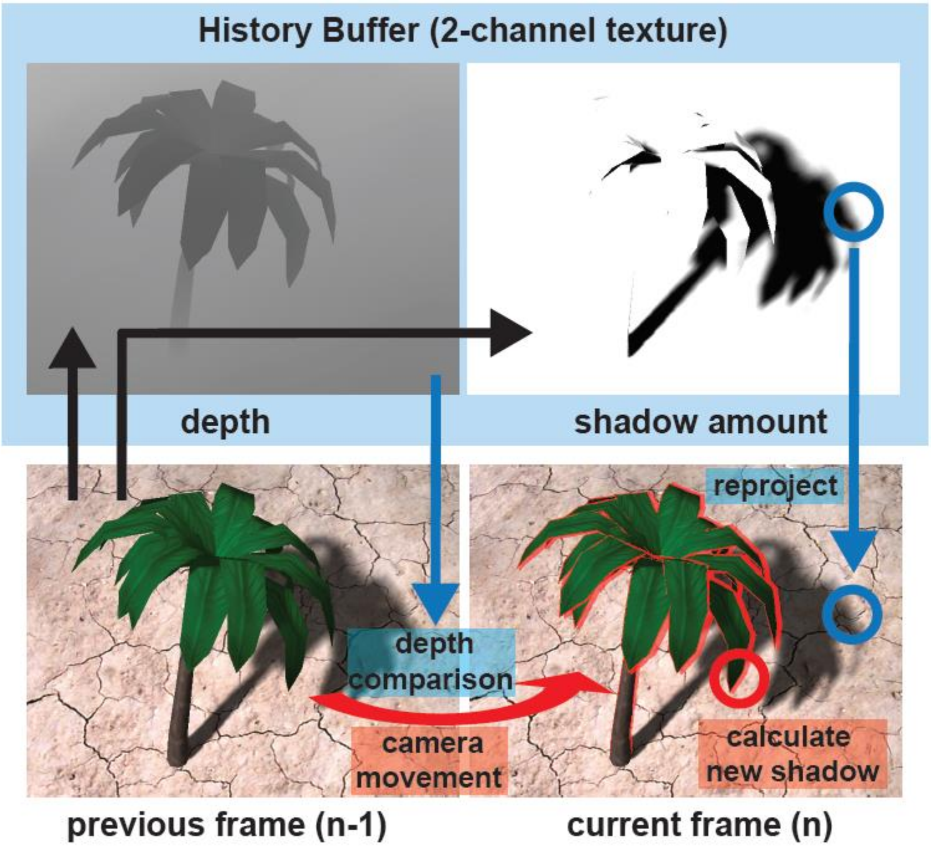


Related Work
Data catching / Temporal coherence

- *Reverse reprojection*
  [Nehab et al. 2007], [Scherzer et al. 2007]
  - storing per pixel info in a *history buffer*
  - compare stored depth with the current one
    - in order to reuse it
    - safely (e.g. not when lights moved)
  - calculate information in disoccluded regions or new areas into frustum
  - some error may be introduced during reprojection

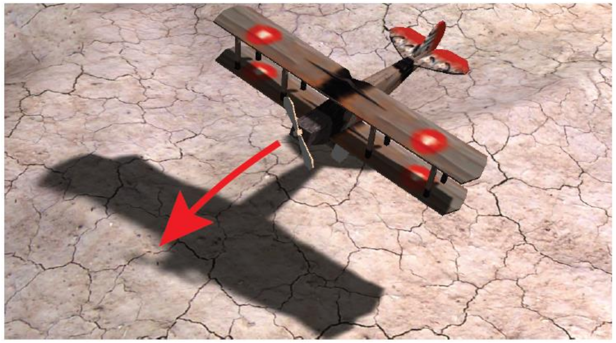
Shadow reprojection

- *History buffer* stores
  - *PCSS* map
  - Depth map
  - In a ping-pong style
- Depth test only in
  - Disoccluded regions
  - New border regions



Detecting moving objects

- Two type shadows on moving objects
  - Shadows cast **on** it
    - Detect areas with depth test
  - Shadows cast **by** it
    - *Movement Map*
      - “1” moving object visible
      - “0” otherwise

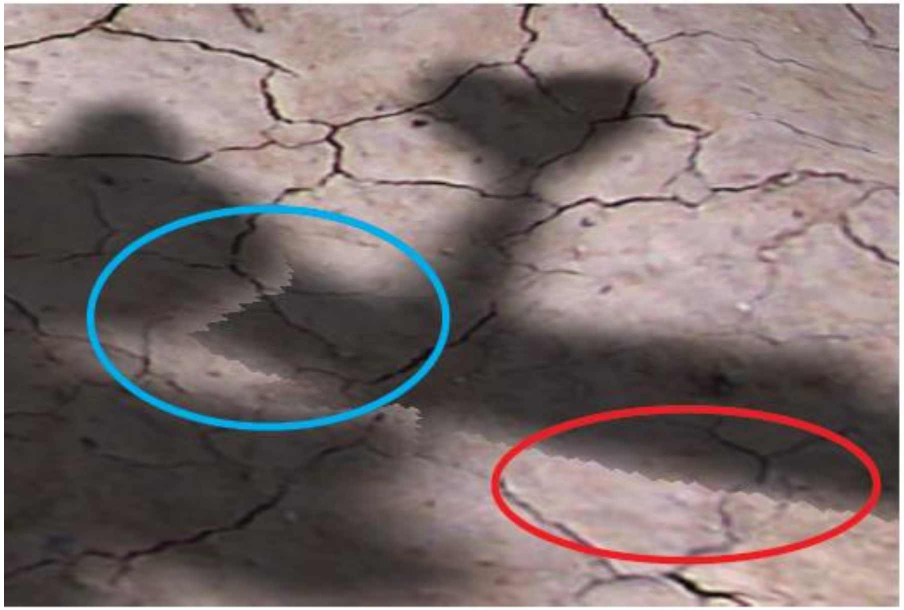


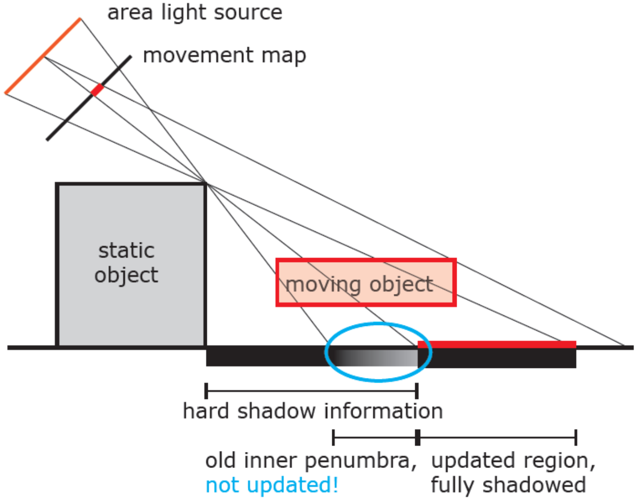
Detecting moving objects

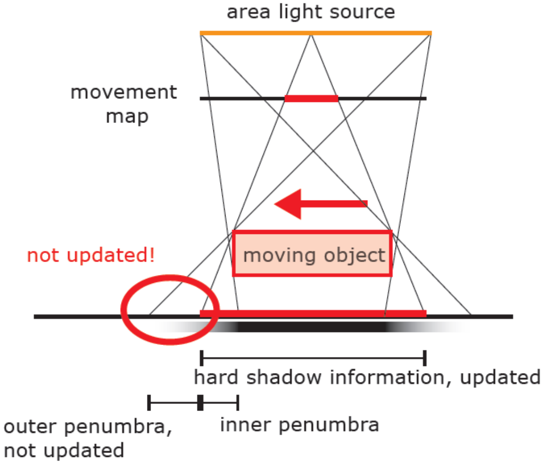
- Movement Map

Detecting moving objects

- Movement map problem:
  - Only have a single hard shadow







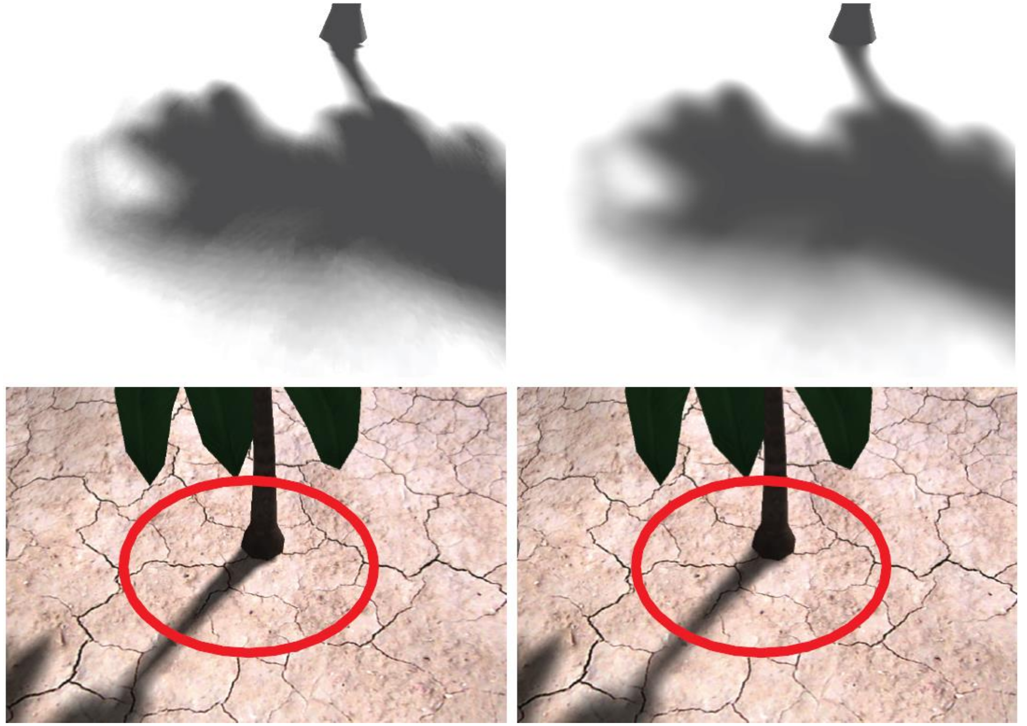
Detecting moving objects

- Solution for the inner penumbra static objects
  - Render moving objects first in both maps
  - Releasing movement map as render target
- Solution for the outer penumbra dynamic objects
  - *Pixel Mipmap* generation procedure to create a pyramid of movement map
    - Mipmap selection equal to occluder search radius PCSS

$$r_{search} = \frac{w_{light} * (z_{receiver} - d_{Nearplane})}{z_{receiver}}$$

Reconstruction error

- With camera movement, reprojecting the history buffer from the previous frame introduces error
  - Bilinear interpolation



Reconstruction error

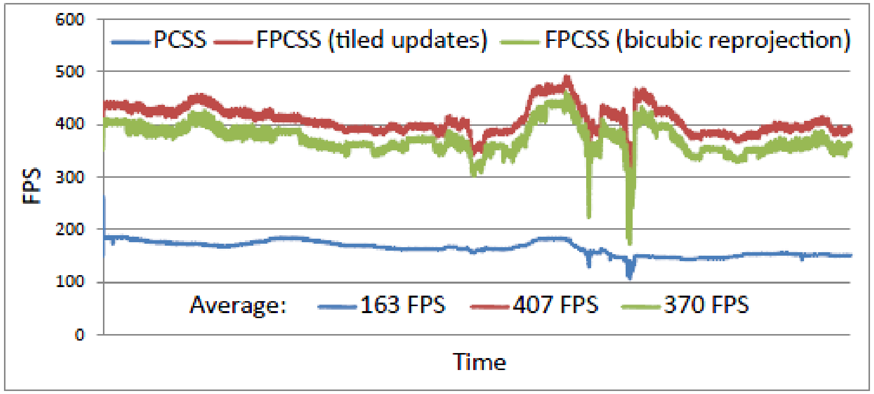
- Accumulated projection error cannot be too large
  - Avoid “oversmoothing”
  - Depends on the scene parameters
  - Two options:
    - Bicubic texture sampling (Catmull-Rom interpolation)
    - Dividing the screen into a groups in a grid
      - Update periodically
      - Parts with too much error recalculated and blended with the old ones

Implementation and evaluation

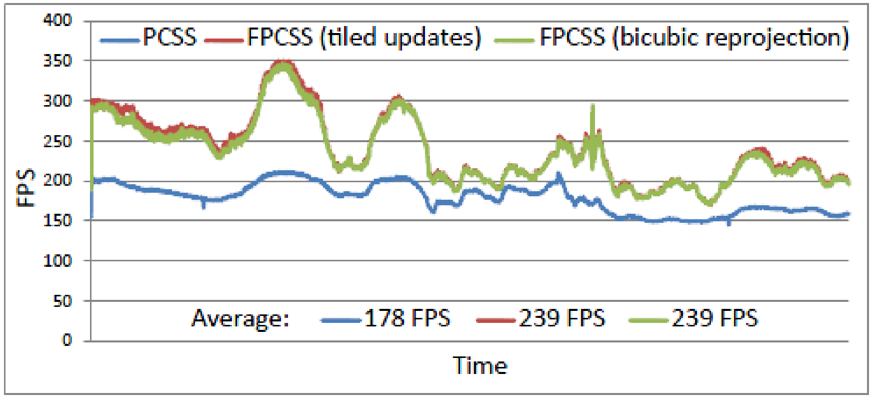
- Implementation
  - in a C++, with shaders using DirectX10
- Tests
  - Intel Core i7 920 with 6GB RAM and GeForce GTX 580
  - Screen resolution 1920x1080
  - Three different scenarios and three different methods

Implementation and evaluation

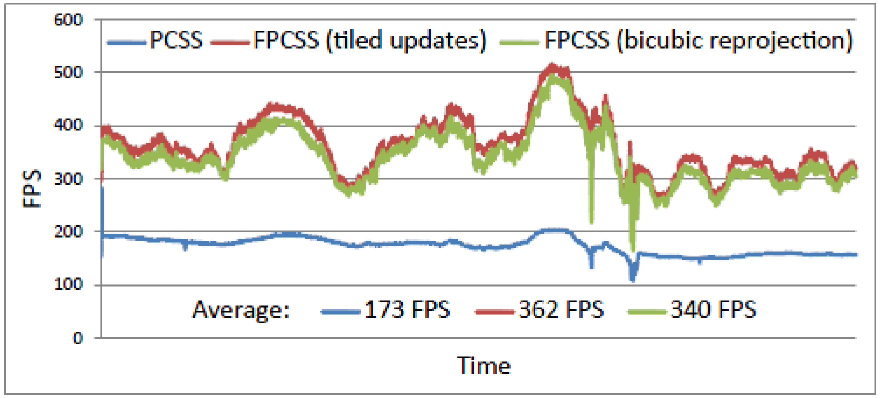
Static Scene



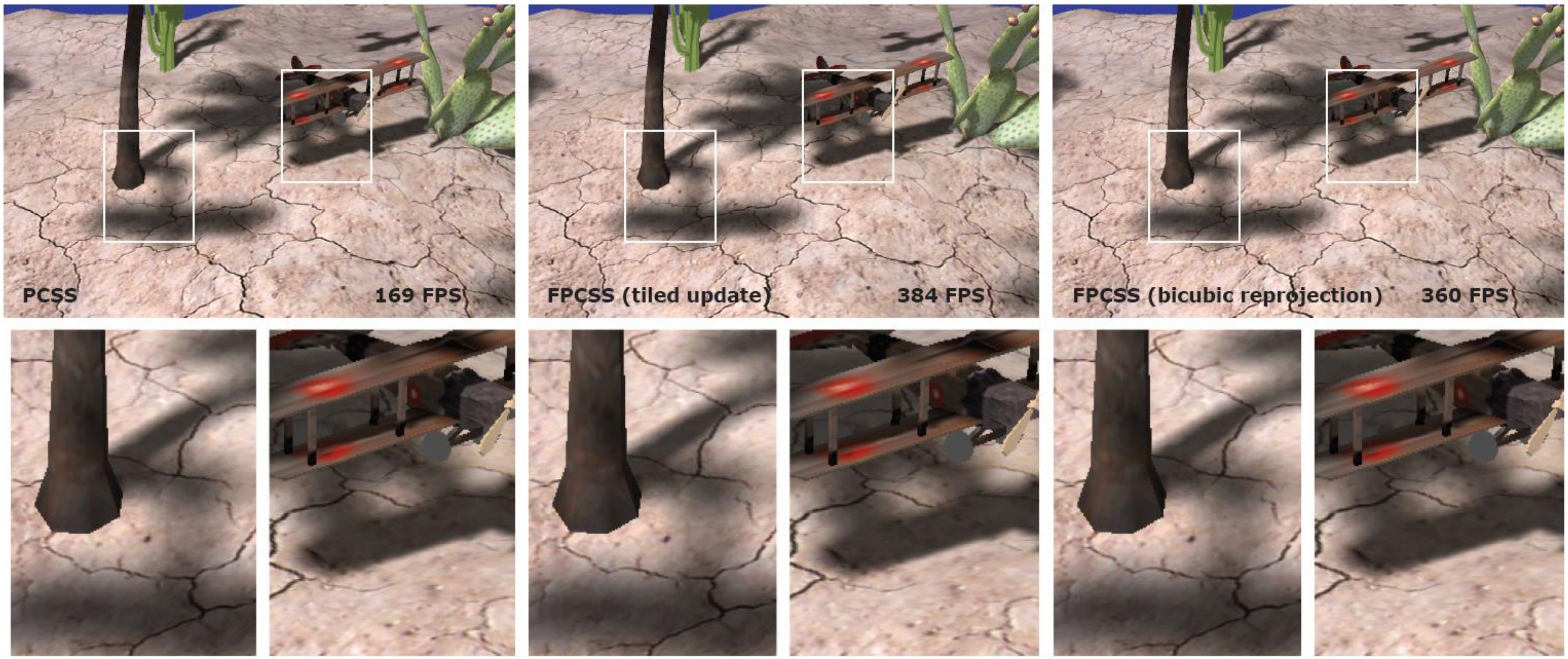
Fully Dynamic Scene



Mixed Scene (dynamic & static objects)



Implementation and evaluation



Conclusions and Future Work

- New improvement method for PCSS
  - Easy to integrate into an existing rendering framework
  - Can be used for all kinds of different scenes
  - The achievable performance gain comes at the cost of memory consumption
- Not for a moving light sources
- FPS variable: may be a problem
- Future work
  - real-time calculation of physically correct soft shadows in dynamic scenes

Fast Percentage Closer Soft Shadows using Temporal Coherence

Michael Schwärzler*
VRVis Research Center, Austria

Christian Luksch†
VRVis Research Center, Austria

Daniel Scherzer‡
Max-Planck-Institut für Informatik, Germany

Michael Wimmer§
Vienna University of Technology, Austria

Pere-Pau Vázquez – ViRVIG Group, UPC

Percentage Closer Filtering and Percentage Closer Soft Shadows