Real Time Skin Rendering

David Gosselin
3D Application Research Group

ATI Research, Inc.

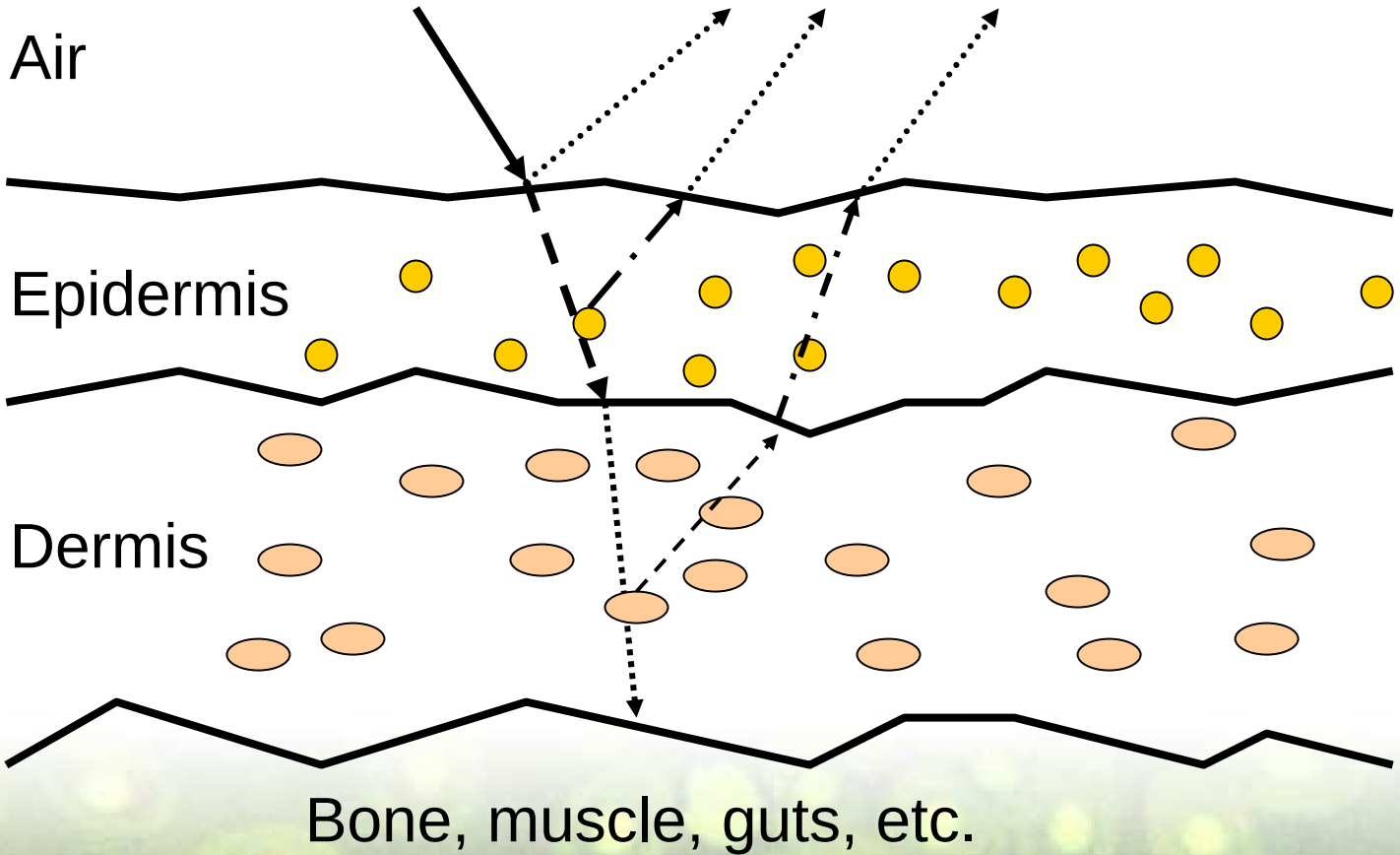
Overview

- Background
- Texture space lighting
- Spatially varying blur
- Dilation
- Adding shadows
- Specular with shadows

Why Skin is Hard

- Most diffuse lighting from skin comes from sub-surface scattering
- Skin color mainly from epidermis
- Pink/red color mainly from blood in dermis
- Lambertian model designed for “hard” surfaces with little sub-surface scattering so it doesn’t work real well for skin

Rough Cross Section



Research

- There are several good mathematical models available
- We looked at using Hanrahan/Krueger (SIGGRAPH 93) based model
  - Good but expensive for current technology
  - Over 100 instructions per light

Basis for Our Approach

- SIGGRAPH 2003 sketch **Realistic Human Face Rendering for "The Matrix Reloaded"** by George Borshukov and J. P. Lewis
- Rendered a 2D light map
- Simulate subsurface diffusion in image domain (different for each color component)
- Used traditional ray tracing for areas where light can pass all the way through (e.g. ears)
- Also capture fine detail normal maps and albedo maps

Texture Space Subsurface Scattering

- From **Realistic Human Face Rendering for "The Matrix Reloaded"** @ SIGGRAPH 2003:

From *Matrix: Reloaded* sketch

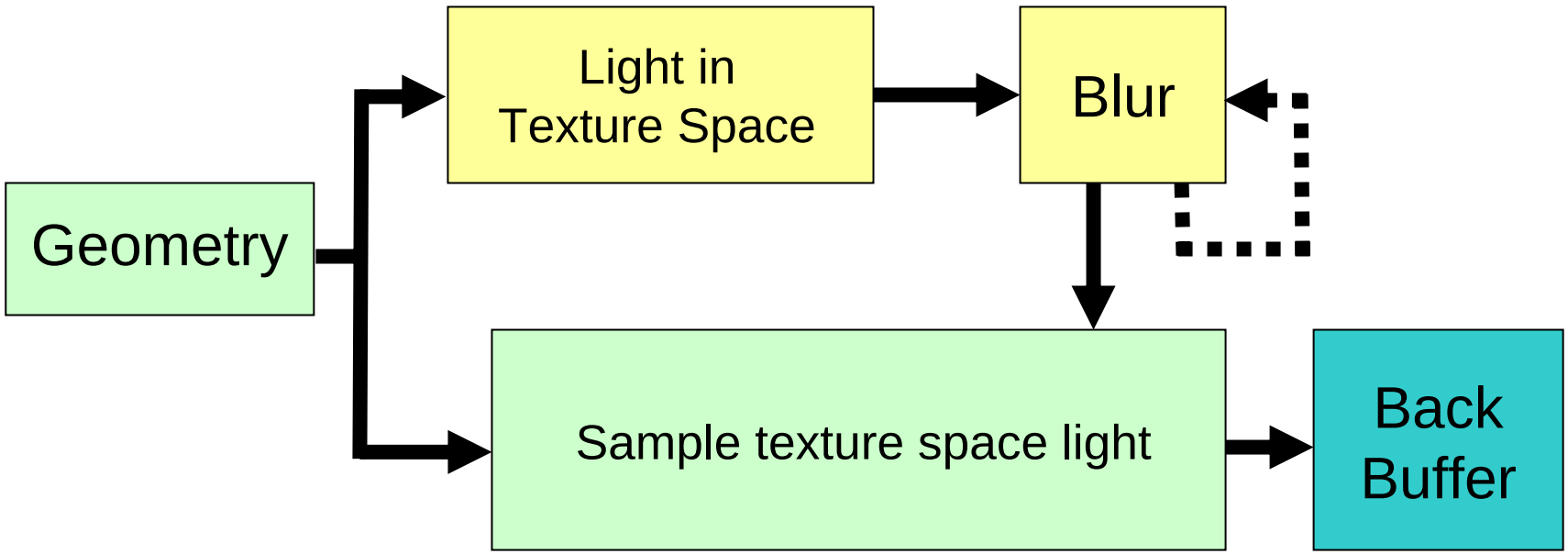
- Our results:

Current skin in Real Time

Texture Space Lighting for Real Time

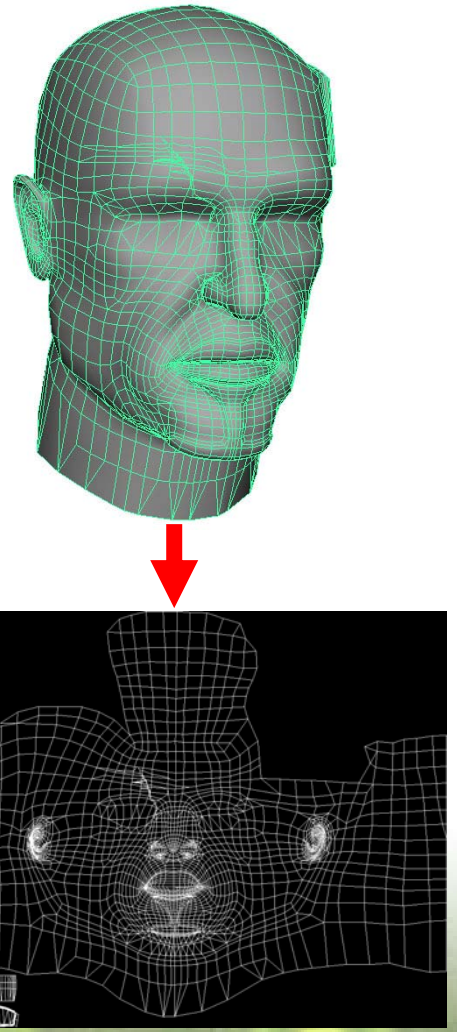
- Render diffuse lighting into an off-screen texture using texture coordinates as position
- Blur the off-screen diffuse lighting
- Read the texture back and add specular lighting in subsequent pass
- We only used bump map for the specular lighting pass

Basic Approach



Texture Coordinates as Position

- Need to light as a 3D model but draw into texture
- By passing texture coordinates as “position” the rasterizer does the unwrap
- Compute light vectors based on 3D position and interpolate



Texture Lighting Vertex Shader

```
VsOutput main (VsInput i)
{
    // Compute output texel position
    VsOutput o;
    o.pos.xy = i.texCoord*2.0-1.0;
    o.pos.z = 1.0;
    o.pos.w = 1.0;

    // Pass along texture coordinates
    o.texCoord = i.texCoord;

    // Skin
    float4x4 mSkinning = SiComputeSkinningMatrix (i.weights, i.indices);
    float4 pos = mul (i.pos, mSkinning);
    pos = pos/pos.w;
    o.normal = mul (i.normal, mSkinning);

    // Compute Object light vectors
    // etc.
    . . .
```

Texture Lighting Pixel Shader

```
float4 main (PsInput i) : COLOR
{
   // Compute Object Light 0
   float3 vNormal = normalize (i.normal);
   float3 lightColor = 2.0 * SiGetObjectAmbientLightColor(0);
   float3 vLight = normalize (i.oaLightVec0);
   float NdotL = SiDot3Clamp (vNormal, vLight);
   float3 diffuse = saturate (NdotL * lightColor);

   // Compute Object Light 1 & 2
   . . .

   float4 o;
   o.rgb = diffuse;
   float4 cBump = tex2D (tBump, i.texCoord);
   o.a = cBump.a; // Save off blur size
   return o;
}
```

Texture Lighting Results

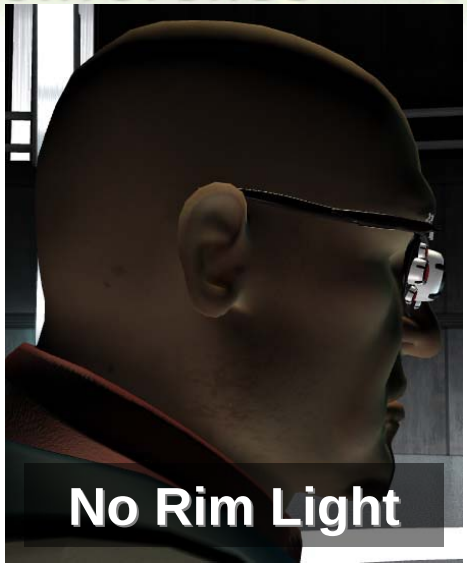
Rim light

- We wanted to further emphasize the light that bleeds through the skin when backlit
- Compute the dot product between the negative light vector and the view vector
- Multiply result by Fresnel term
- Only shows up if there is a light roughly “behind” the object

Pixel Shader

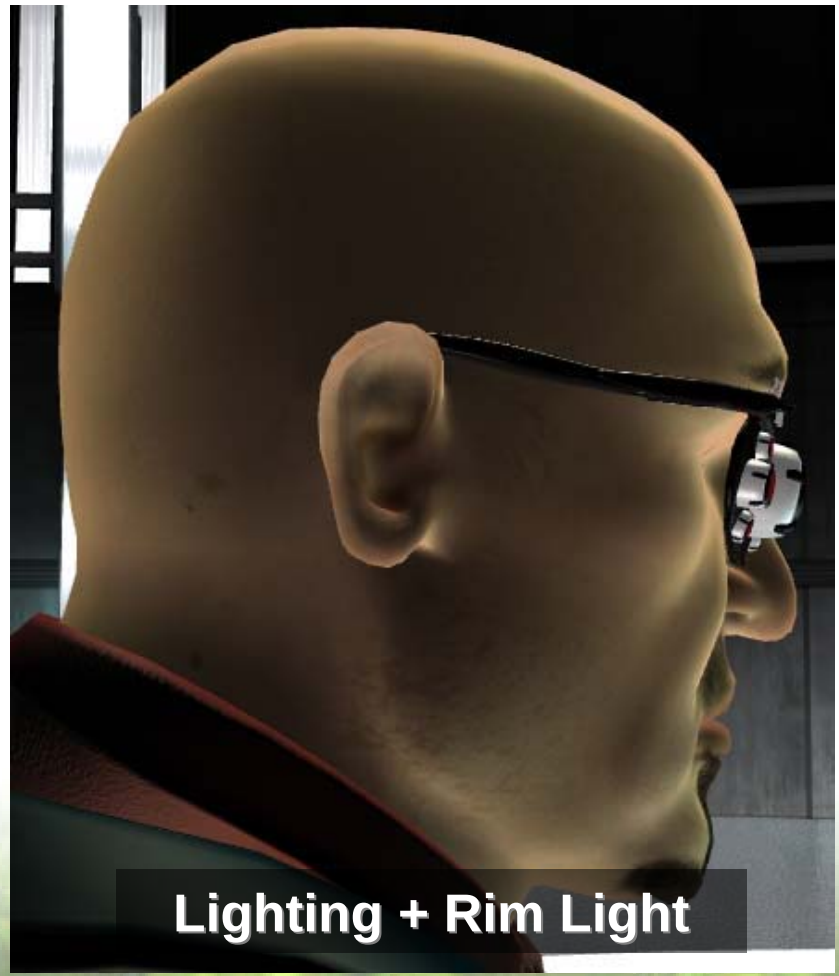
```
float4 main (PsInput i) : COLOR
{
   // Normalize interpolated vectors.
   float3 vNormal = normalize (i.normal);
   float3 vView = normalize (i.viewVec);
   float NdotV = SiDot3Clamp (vNormal, vView);
   float fresnel = (1.0f - NdotV);

   // Compute Object Light 0
   float3 lightColor = 2.0 * SiGetObjectAmbientLightColor(0);
   float3 vLight = normalize (i.oaLightVec0);
   float NdotL = SiDot3Clamp (vNormal, vLight);
   float VdotL = SiDot3Clamp (-vLight, vView);
   float3 diffuse = saturate ((fresnel*VdotL+NdotL)*lightColor);

   // Compute Object Light 1 & 2 in the same way
   // Output diffuse and alpha from bump map (blur size)
   . . .
```

Added Rim Light Result



+

=

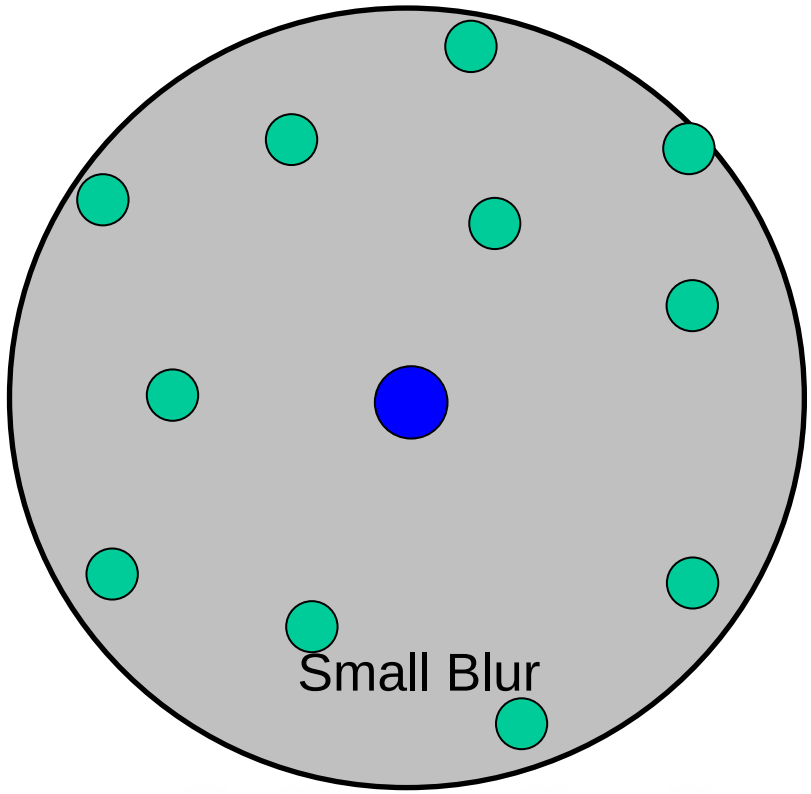


Spatially Varying Blur

- Used to simulate the subsurface component of skin lighting
- Used a grow-able Poisson disc filter
- Read the kernel size from a texture
- Allows varying the subsurface effect
  - Higher for places like ears/nose
  - Lower for places like cheeks

Growable Filter Kernel

- Stochastic sampling
- Poisson distribution
- Samples stored as 2D offsets from center



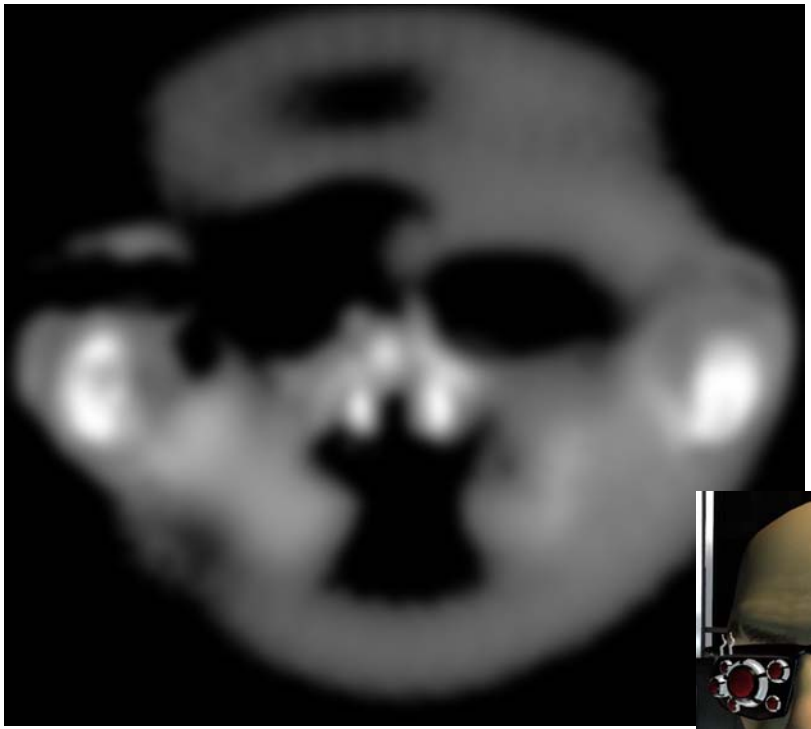
Center Sample

Outer Samples

Spatially Varying Blur Pixel Shader

```
float4 main (PsInput i) : COLOR
{
   float2 poisson[12] = . . . // Texel offsets from center

   // Figure out blur size
   float4 center = tex2D(tRenderedScenePong, i.texCoord);
   float blurSize = center.a*vBlurScale.x + vBlurScale.y;

   // Loop over the taps summing contributions
   float3 cOut = center.rgb;
   for (int tap = 0; tap < 12; tap++)
   {
      // Sample using Poisson taps
      float2 coord = i.texCoord.xy+(vPixelSize*poisson[tap]*blurSize);
      float4 sample = tex2D (tRenderedScenePong, coord);
      cOut += sample.rgb;
   }
   return float4(cOut / 13.0f, center.a);
}
```

Blur Size Map and Blurred Lit Texture



Blur Kernel Size Map

Result

Texture Space Lighting

Dilation

- Texture seams can be a problem (unused texels, bilinear blending artifacts)
- During the blur pass we need to dilate
- Use the alpha channel of off-screen texture to determine where we wrote
- If any sample has 1.0 alpha, just copy the sample with the lowest alpha

Dilation + Blur Pixel Shader Code

```
float4 main (PsInput i) : COLOR
{
   float2 poisson[12] = // Texel offsets from center

   // Figure out blur size
   float4 center = tex2D(tRenderedScenePong, i.texCoord);
   float blurSize = center.a*vBlurScale.x + vBlurScale.y;

   // flag is the max alpha value. If it is 1.0f, then sample is
   // close to the boundary since we clear alpha to 1.0
   float flag = center.a;
```

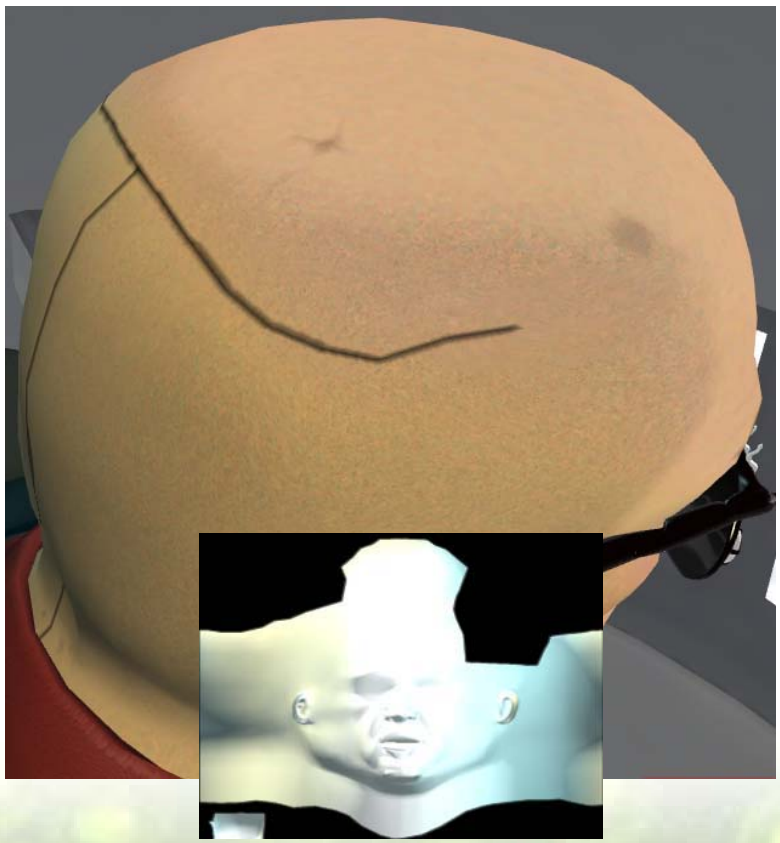
Main Dilate/Blur Pixel Shader Loop

```
// Loop over the taps summing contributions
float3 cOut = center.rgb;
for (int tap = 0; tap < 12; tap++)
{
   // Sample using Poisson distribution
   float2 coord = i.texCoord.xy + (vPixelSize*poisson[tap]*blurSize);
   float4 sample = tex2D (tRenderedScenePing, coord);
   cOut += sample.rgb;

   // Figure out if we need to change the flag
   flag = max (sample.a, flag);
   if (sample.a < center.a)
   {
      // Store texel with lowest alpha; will be used if close to
      // the boundary to "dilate" by picking a more "inside" texel
      center = sample;
   }
}
```

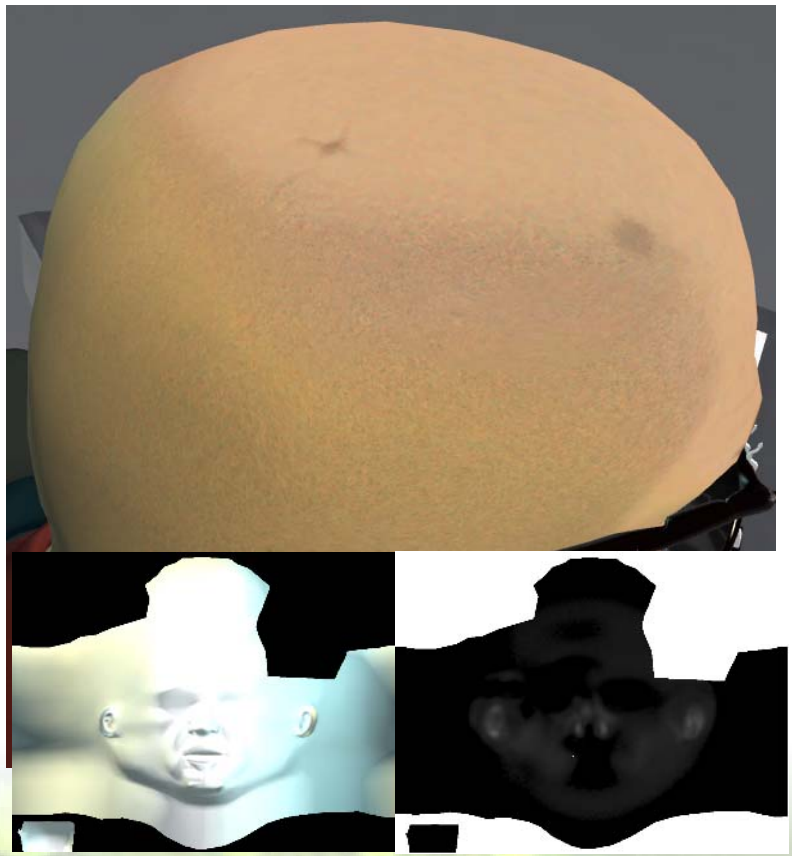
Dilate Test Pixel Shader

```
    // Test the flag to see if we are on a boundary texel
    if (flag == 1.0f)
    {
        // On a boundary pick the texel with the lowest alpha
        return float4 (center.rgb, 1.0f);
    }
    else
    {
        // Not on a boundary same blur as before.
        return float4(cOut / 13.0f, 0.0f);
    }
}
```

Dilation Results



Without Dilation



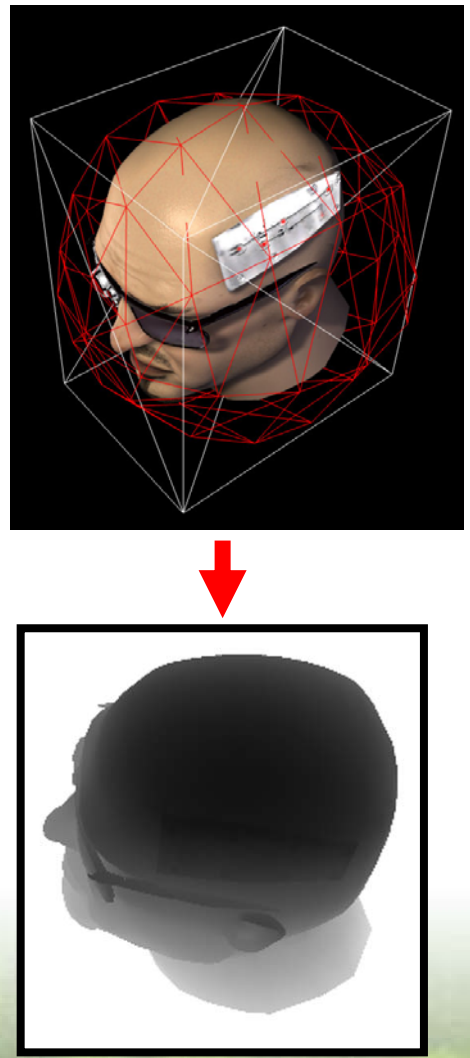
With Dilation

Shadows

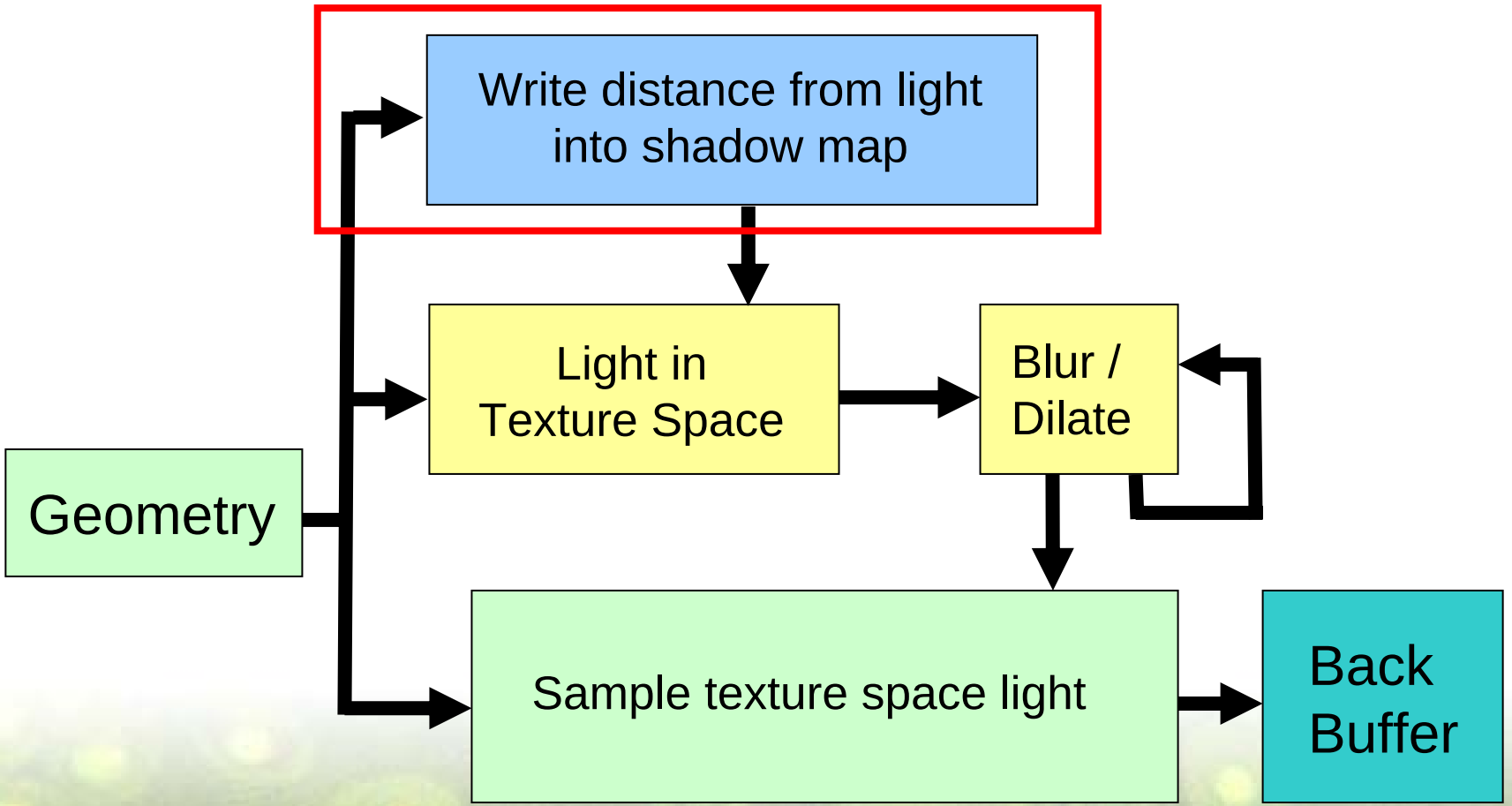
- Used shadow maps
  - Apply shadows during texture lighting
  - Get “free” blur
    - Soft shadows
    - Simulates subsurface interaction
    - Lower precision/size requirements
    - Reduces artifacts
- Only doing shadows from one key light

Shadow Maps

- Create projection matrix to generate map from the light's point of view
- Used bounding sphere of head to ensure texture space is used efficiently
- Write depth from light into off-screen texture
- Test depth values in pixel shader



Texture Lighting With Shadows



Shadow Map Vertex Shader

```
float4x4 mSiLightProjection; // Light projection matrix
VsOutput main (VsInput i)
{
   VsOutput o;

   // Compose skinning matrix
   float4x4 mSkinning = SiComputeSkinningMatrix(i.weights, i.indices);

   // Skin position/normal and multiply by light matrix
   float4 pos = mul (i.pos, mSkinning);
   o.pos = mul (pos, mSiLightProjection);

   // Compute depth (Pixel Shader is just pass through)
   float dv = o.pos.z/o.pos.w;
   o.depth = float4(dv, dv, dv, 1);
   return o;
}
```

Texture Lighting Vertex Shader with Shadows

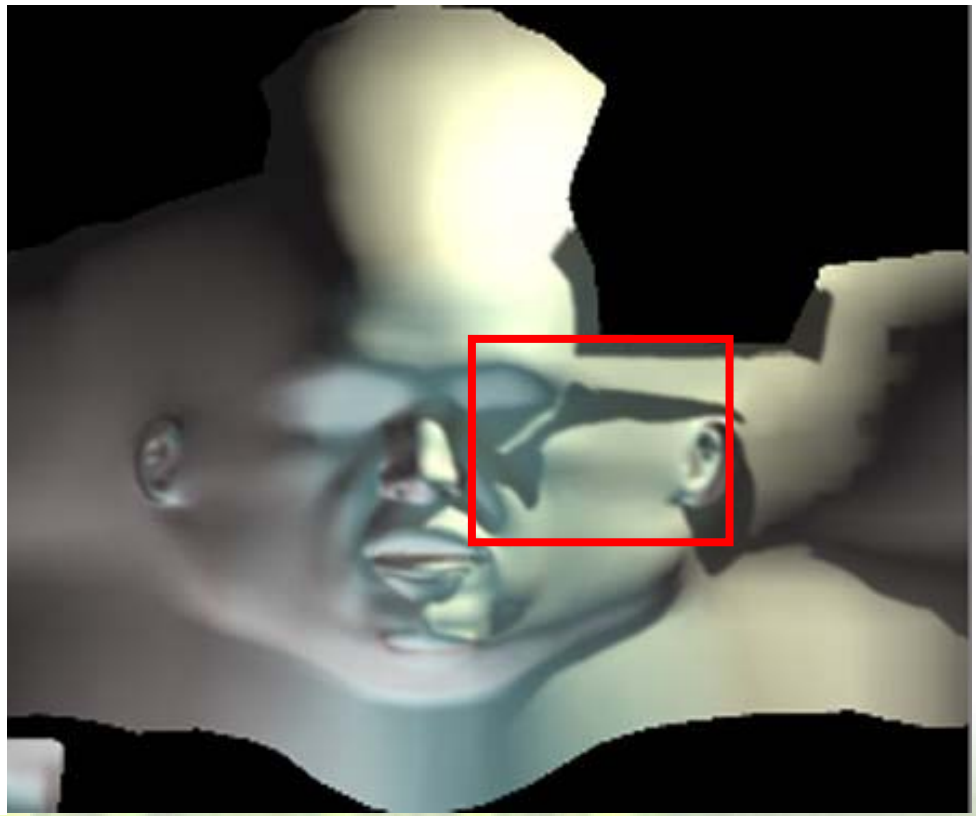
```
VsOutput main (VsInput i)
{
   // Same lead in code as before
   . . .

   // Compute texture coordintates for shadow map
   o.posLight = mul(pos, mSiLightKingPin);
   o.posLight /= o.posLight.w;
   o.posLight.xy = (o.posLight.xy + 1.0f)/2.0f;
   o.posLight.y = 1.0f-o.posLight.y;
   o.posLight.z -= 0.01f;
   return o;
}
```

Texture Lighting Pixel Shader with Shadows

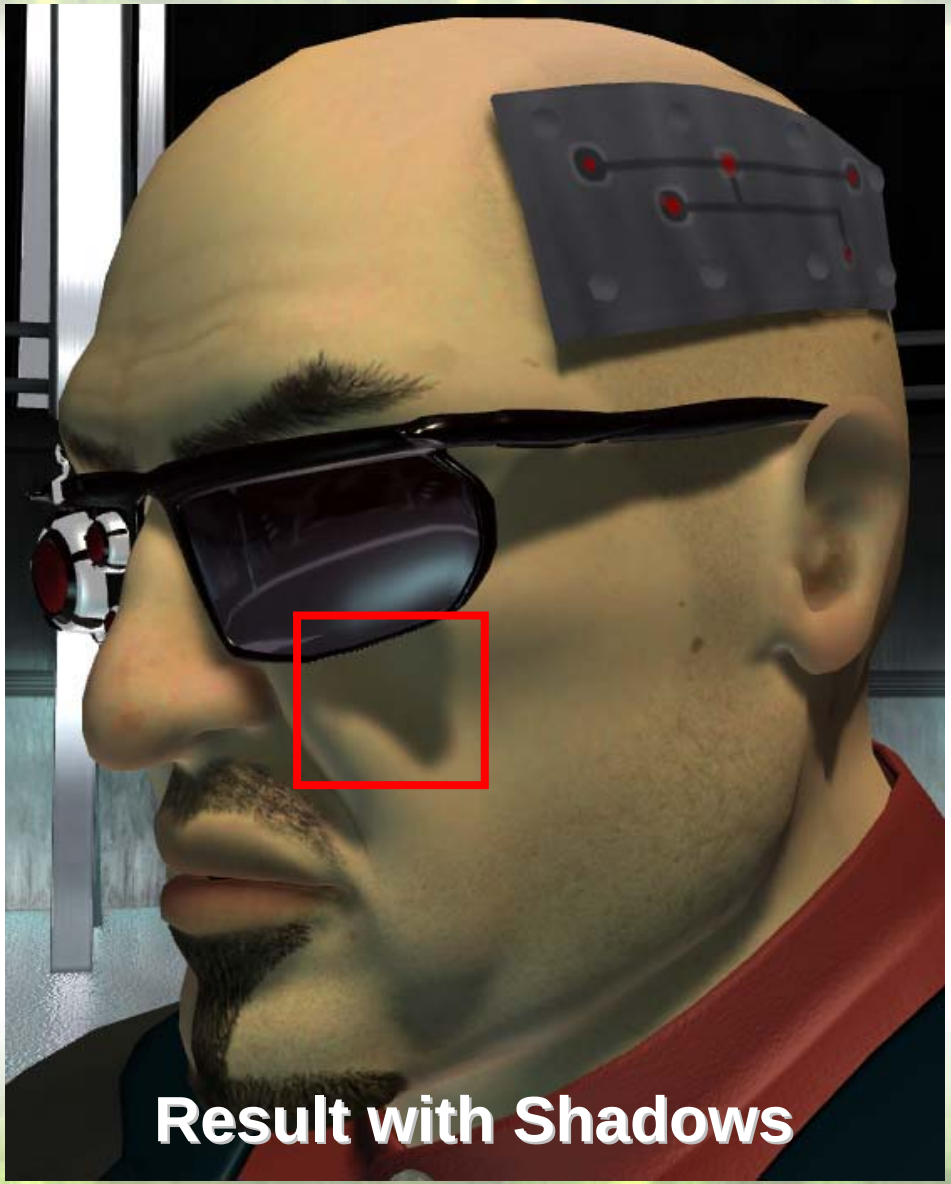
```
sampler tShadowMap;
float faceShadowFactor;
float4 main (PsInput i) : COLOR
{
   // Same lead in code
   . . .

   // Compute Object Light 0
   float3 lightColor = 2.0 * SiGetObjectAmbientLightColor(0);
   float3 vLight = normalize (i.oaLightVec0);
   float NdotL = SiDot3Clamp (vNormal, vLight);
   float VdotL = SiDot3Clamp (-vLight, vView);
   float4 t = tex2D(tShadowMap, i.posLight.xy);
   float lfac = faceShadowFactor;
   if (i.posLight.z < t.z) lfac = 1.0f;
   float3 diffuse = lfac * saturate ((fresnel*VdotL+NdotL)*lightColor);

   . . .// The rest of the shader is the same as before
}
```

Shadow Map and Shadowed Lit Texture

Shadow Map (depth)



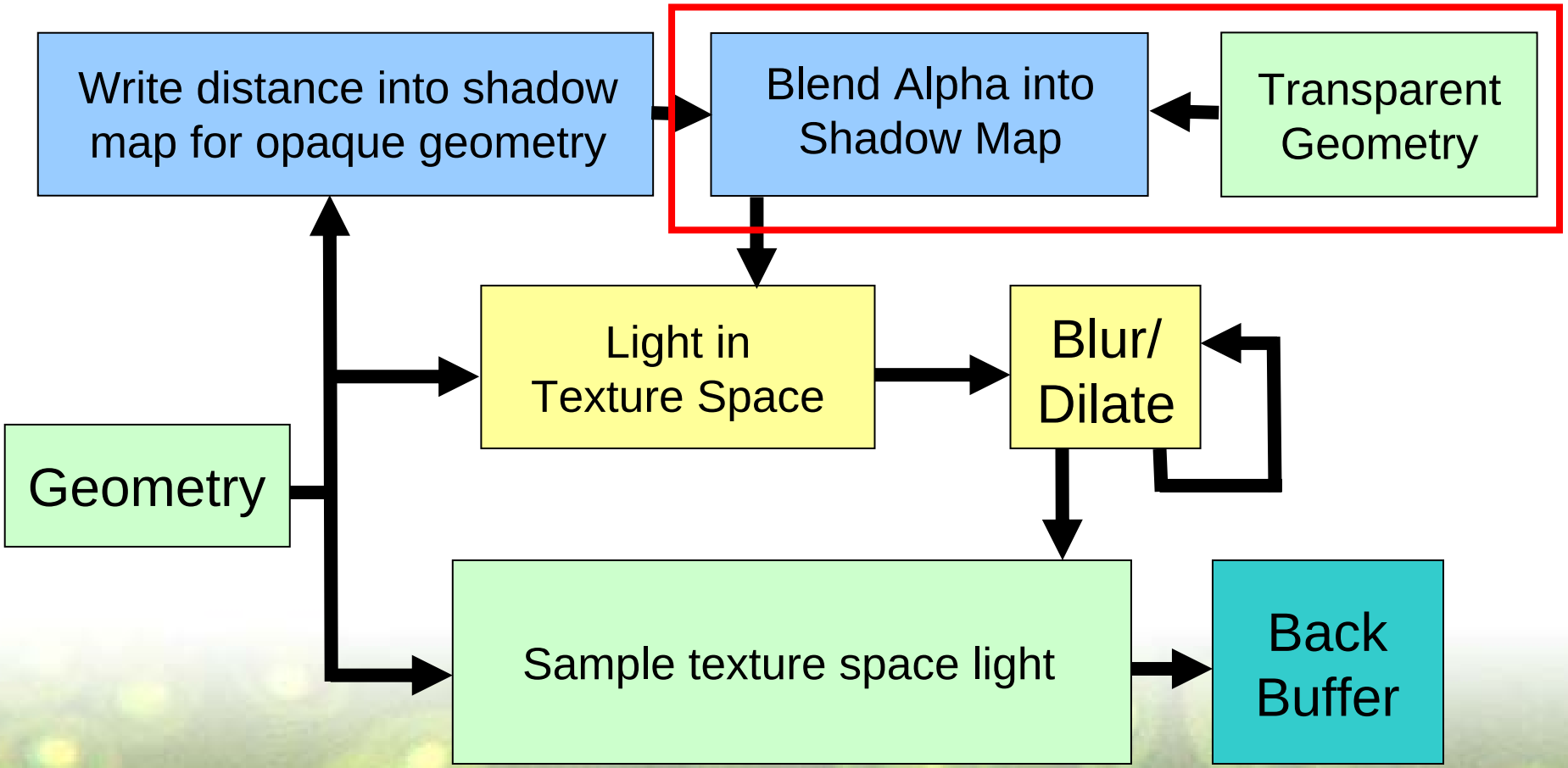
Shadows in Texture Space


Result with Shadows

Shadows From Translucent Objects

- Allow multiple translucent objects that combine to form opaque shadow (hair)
- Draw opaque shadow geometry first
- Blend alpha of translucent shadow geometry into shadow buffer alpha. Don’t write depth!
- In pixel shader: non-shadowed pixels lerp between shadow term and 1.0 based on alpha in shadow map

Texture Lighting With Translucent Shadows

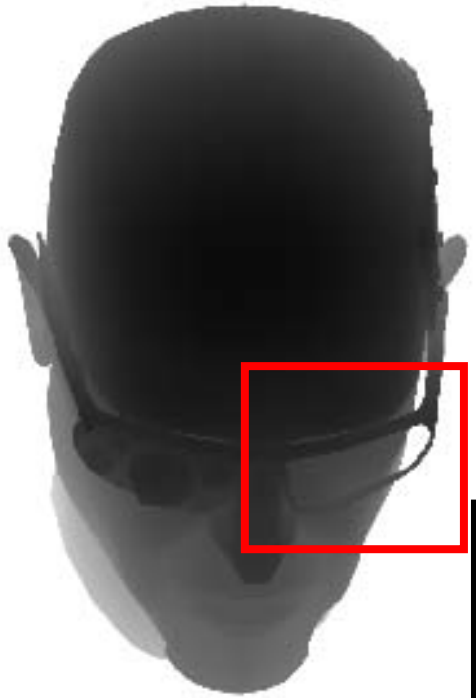


Translucent Shadow Pixel Shader

```
float shadowAlpha;
float4 main (PsInput i) : COLOR
{
   // Same lead in
   . . .

   // Usual light 0 code
   . . .
   float4 t = tex2D(tShadowMap, i.posLight.xy);
   float lfac = faceShadowFactor;
   if (i.posLight.z < t.z)
   {
      float alpha = pow(t.a, shadowAlpha);
      lfac = lerp(faceShadowFactor, 1.0f, alpha);
   }
   float3 diffuse = lfac * saturate((fresnel*VdotL+NdotL)*lightColor);

   . . . // Rest of the shader is the same as well
```

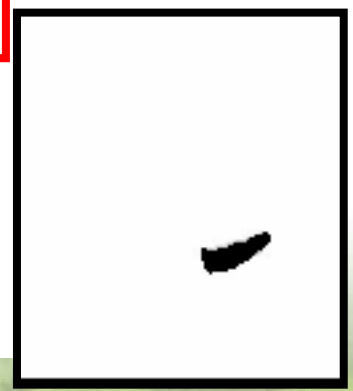
Shadow Map for Transparent Shadows

Shadow Map Fully Opaque

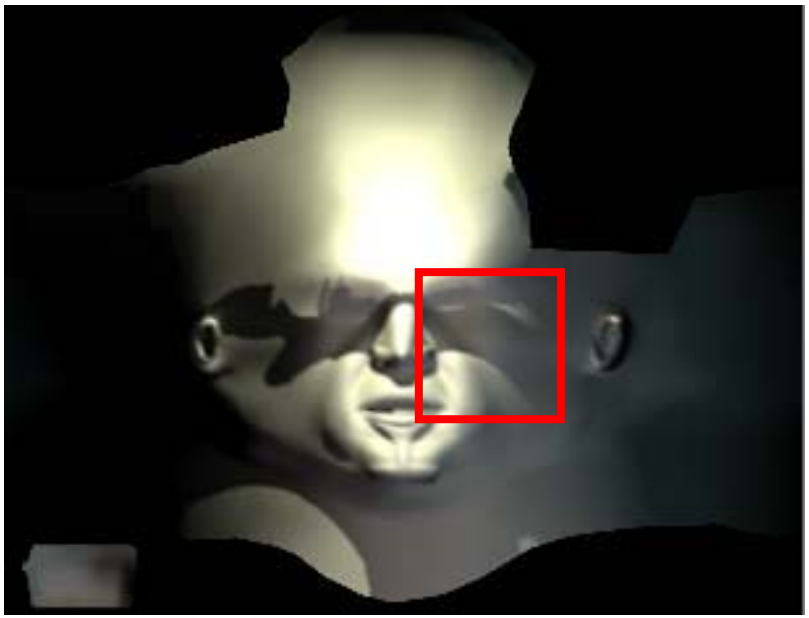


Shadow Map With Translucency

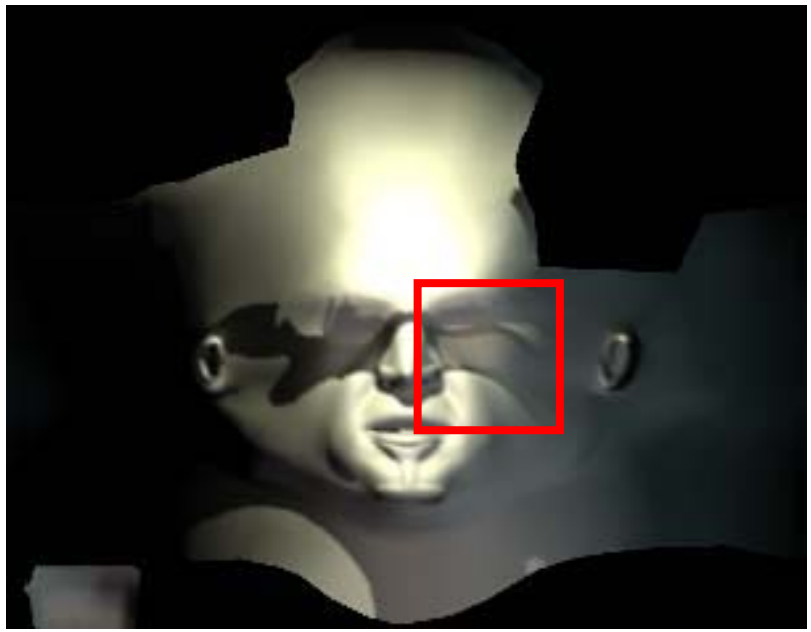
Alpha



Off Screen Light Textures with Translucent Shadows



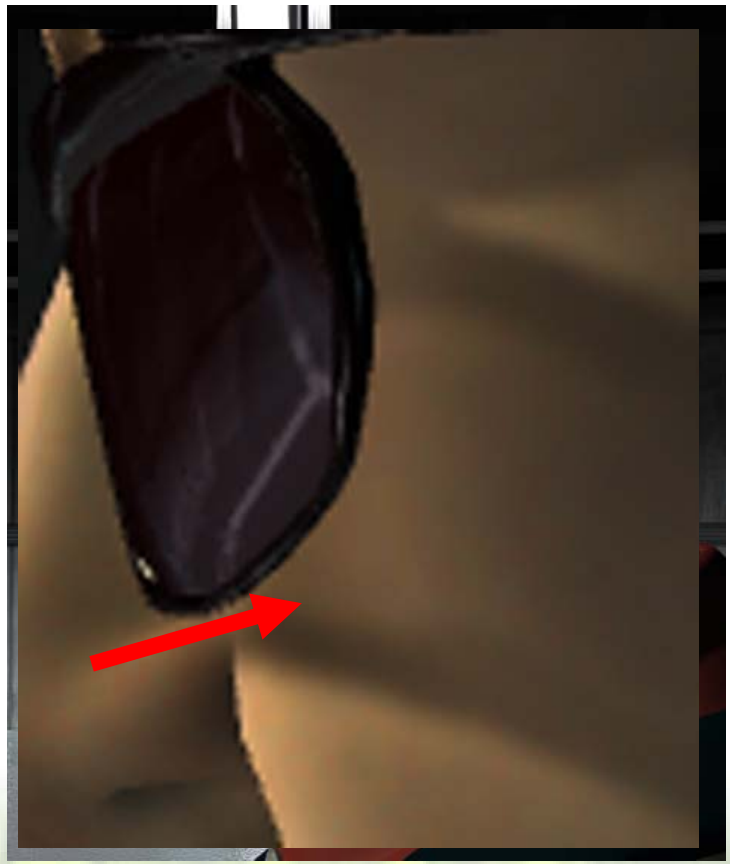
Opaque Shadows



Translucent Shadows

Translucent Shadows Results

Opaque Shadows



Translucent Shadows

Specular

- Use bump map for specular lighting
- Per-pixel exponent
- Need to shadow specular
  - Hard to blur shadow map directly
  - Expensive to do yet another blur pass for shadows
  - Modulate specular from shadowing light by luminance of texture space light
  - Darkens specular in shadowed areas but preserves lighting in unshadowed areas
- Shadow only dims one light (2 other un-shadowed)

Final Pixel Shader (with specular)

```
sampler tBase;
sampler tBump;
sampler tTextureLit;
float4 vBumpScale;
float specularDim;
float4 main (PsInput i) : COLOR
{
   // Get base and bump map
   float4 cBase = tex2D (tBase, i.texCoord.xy);
   float3 cBump = tex2D (tBump, i.texCoord.xy);

   // Get bumped normal
   float3 vNormal = SiConvertColorToVector (cBump);
   vNormal.z = vNormal.z * vBumpScale.x;
   vNormal = normalize (vNormal);
```

Final Pixel Shader

```
// View, reflection, and specular exponent
float3 vView = normalize (i.viewVec);
float3 vReflect = SiReflect (vView, vNormal);
float exponent = cBase.a*vBumpScale.z + vBumpScale.w;
```

```
// Get "subsurface" light from lit texture.
float2 iTx = i.texCoord.xy;
iTx.y = 1-i.texCoord.y;
float4 cLight = tex2D (tTextureLit, iTx);
float3 diffuse = cLight*cBase;
```

Final Pixel Shader

```
    // Compute Object Light 0
    float3 lightColor = 2.0 * SiGetObjectAmbientLightColor(0);
    float3 vLight = normalize (i.oaLightVec0);
    float RdotL = SiDot3Clamp (vReflect, vLight);
    float shadow = SiGetLuminance (cLight.rgb);
    shadow = pow(shadow, 2);
    float3 specular = saturate(pow(RdotL,exponent)*lightColor)*shadow;

    // Compute Object Light 1 & 2 (same as above but no shadow term)
    . . .

    // Final color
    float4 o;
    o.rgb = diffuse + specular*specularDim;
    o.a = 1.0;
    return o;
}
```

Specular Shadow Dim Results

Specular Without Shadows

Specular With Shadows

Demo

Questions?