2.8

Real-Time Texture-Space Skin Rendering

David R. Gosselin, Pedro V. Sander, and Jason L. Mitchell

Introduction

Rendering of human skin is an important research problem in Computer Graphics. Obtaining realistic results can be very challenging, especially when it has to be accomplished in real time. We present an overview of methods used to simulate the appearance of skin, and describe their implications to real-time rendering. Finally, we describe a real-time algorithm that approximates the appearance of subsurface scattering by performing a blur operation in texture-space using graphics hardware. The advantages of this approach are its simplicity and efficiency, while still achieving a realistic result. We also present additional post-processing techniques to prevent texture seams from being generated by the blur operation, as well as methods to efficiently compute soft shadows using this framework.

Skin Rendering in Real Time

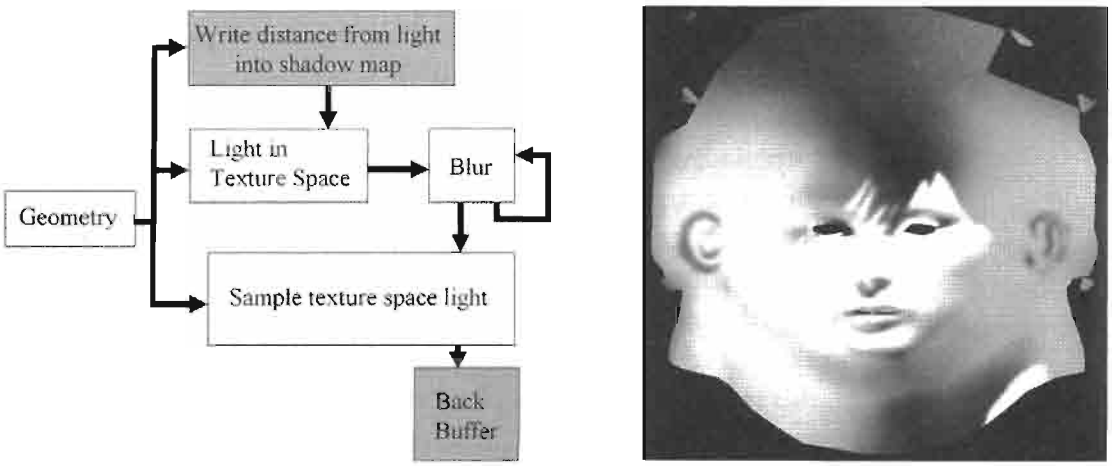
Historically, skin has been shaded much like other materials in real-time games due to a lack of sophistication in hardware shading models. Developers rendering real-time skin have traditionally used tweaked Phong and environment map-based shading models with rim lighting to try to approximate the look of skin [Beeson04]. Most games today don't even go that far, resulting in a very plastic and unrealistic look. Even offline rendering techniques used in films such as *Final Fantasy: The Spirits Within* use Phong models with fairly arbitrary tweaks to select between different textures based upon viewing angle [Bjorke01]. While this gives a decent look and has the advantage of being artist-driven, even this technique does not explicitly attempt to simulate subsurface scattering. A more recent offline rendering technique, used in *The Matrix Reloaded*, simulated subsurface scattering with texture-space lighting and blurring steps [Borshukov03]. In this technique, the illumination of the skin is rendered to a texture map and then blurred to simulate the effect of subsurface scattering. As it turns out, this technique can be implemented using graphics hardware and is the basis for the approach to skin rendering discussed in this article.

Texture-Space Skin Rendering Overview

(a) Without texture blur (b) Split screen (w/o and w/ blur) (c) With texture blur

FIGURE 2.8.1 *Comparison of skin rendering with and without the texture-space blur operation. Images from ATI's demo* Ruby: The Double Cross. *© ATI Technologies 2004.*

The basic idea of the texture-space skin rendering algorithm is to render the diffuse illumination to a light map, blur this light map (see Figure 2.8.2b) in order to approximate subsurface scattering, and finally use the blurred light map to render the final image [Borshukov03]. The specular illumination does not use the light map and thus is not blurred. In addition, prior to rendering the light map, we create a shadow map which is then used to attenuate shadowed regions when rendering the diffuse illumination to the light map.



(a) Outline of the algorithm (b) Rendered light map

FIGURE 2.8.2 *The structure of the texture-space algorithm (left), and the texture-space lightmap used to render Figure 2.8.1 (right).*

In summary, the algorithm proceeds as follows:

1. Create shadow map from the point of view of the key light.
2. Render diffuse illumination to a 2D light map (z test against shadow map).
3. Dilate the boundaries and blur the light map.
4. Render final mesh (use the blurred light map for diffuse illumination).

Next we describe each of the above steps in detail. Then we describe some additional acceleration techniques.

Create Shadow Map

Using this texture-space skin rendering technique, the computation of soft shadows is relatively inexpensive. A shadow map algorithm is used to determine visibility from the light and attenuate the samples when rendering to the light map. The blur operation performed in Step 3 will not only create the appearance of subsurface scattering, but it will also provide soft shadows at no additional cost. The blur pass also significantly reduces aliasing, making it practical to use the shadow map algorithm.

First, for each frame, we need to compute the view-projection matrix in order to render the scene from the point of view of the light. To make the best use of the samples in the shadow map, we use a frustum that tightly bounds the bounding sphere of the geometry (see Figure 2.8.3). This is standard practice when creating shadow maps and can be accomplished with a call to `D3DXMatrixLookAtLH()` to compute the view matrix, and a call to `D3DXMatrixPerspectiveLH()` to compute the projection matrix.

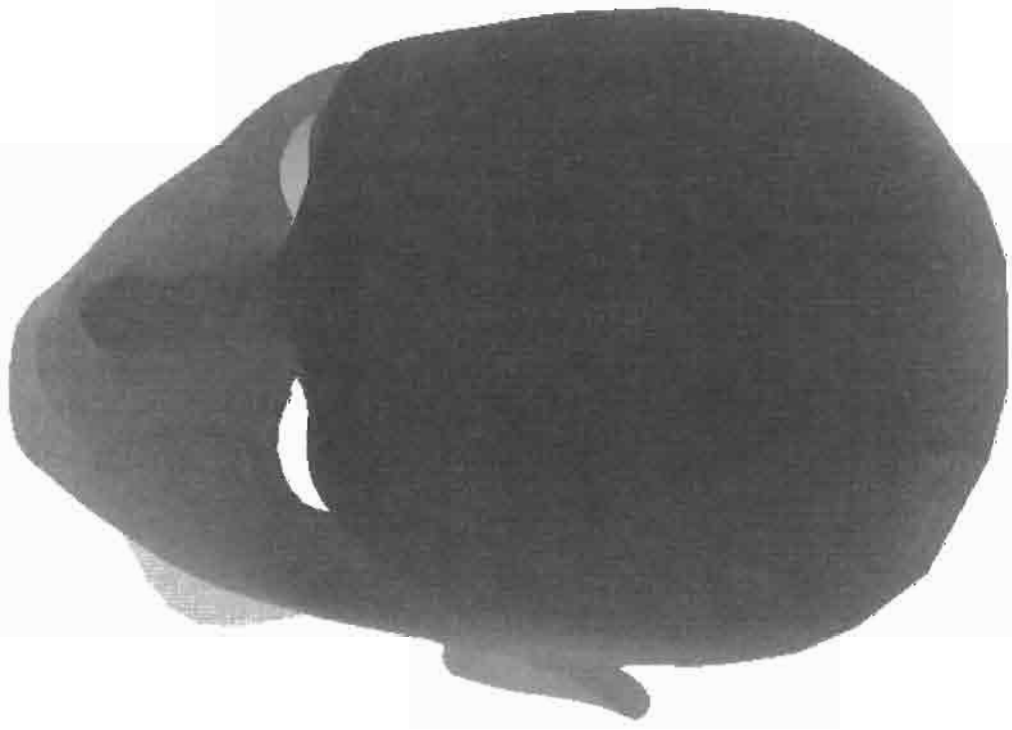


FIGURE 2.8.3 *Shadow map.*

When rendering into the shadow map, we use a shader that stores the depth on the alpha component of an RGBA texture. Figure 2.8.3 shows the depth stored on the alpha channel.

The vertex shader simply has to pass a float containing the depth to the pixel shader in one of the color interpolators:

```
VsOutput vsmain (VsInput i)
{
    VsOutput o;

    //multiply position by light matrix
    o.pos = mul (i.pos, mSiLightAgent);

    //pass depth to pixel shader
    o.depth = o.pos.z/o.pos.w;

    return o;
}
```

And the pixel shader just outputs the interpolated depth:

```
float4 psmain (PsInput i) : COLOR
{
    //output the interpolated depth
    return i.depth;
}
```

Note that since we only write to the alpha channel, the RGB channels should remain set to zero.

Translucent Shadows

Translucent shadows can be computed using a hybrid shadow algorithm. The basic idea is to first render the opaque geometry as described above, and then, on a second pass, render the translucent objects (e.g., the glass lens on Figure 2.8.4) with z testing turned on and z writing turned off. On the second pass, we accumulate the opacity of the samples on the RGB channels of the shadow map texture using additive blending. Then, when computing the shadows from the shadow map, if the sample is not shadowed by an opaque object, its diffuse contribution is attenuated by the value in the alpha channel. As a result, the opaque geometry shadows itself, and the translucent geometry shadows the opaque geometry. The alpha channel, which stores the depth for the opaque geometry should not be changed in the translucent geometry pass. To ensure this, we only enable writing to RGB. Note that translucent objects that are behind the opaque geometry are z culled and thus not rendered.

The vertex shader is similar to the one for the previous pass, except that instead of computing depth, it just passes the texture coordinates for the opacity map lookup to

(a) Without transparency (b) With transparency

FIGURE 2.8.4 *Comparison of rendering the glass lens with and without translucent shadows. Images from ATI's demo* Ruby: The Double Cross. *© ATI Technologies 2004.*

the pixel shader in one of the interpolators. The pixel shader returns the result of the texture lookup:

```
return tex2D(tOpacity, i.texCoord);
```

Note that the opacity map is an RGB texture in order to allow for colored translucent materials, such as purple shades casting a purple shadow (Figure 2.8.4(b)). To use a single-channel opacity map, just output the red channel of the opacity map:

```
return tex2D(tOpacity, i.texCoord).rrrr;
```

Remember that the alpha channel, which contains the depth values for the opaque geometry is not touched during this pass.

Render Diffuse Illumination to Light Map

The next step of the algorithm is to render the diffuse illumination of the opaque geometry to the light map. The vertex shader sets the output positions to be the texture coordinates of the vertices, thus rendering to a 2D texture map, which consistently samples the entire surface, independent of the camera position. This is accomplished by the following vertex shader code fragment, which maps the texture coordinates to positions in the [–1, 1] range:

```
o.pos.xy = i.texCoord*2.0-1.0;
o.pos.z = 0.0;
o.pos.w = 1.0;
```

The vertex shader also needs to compute the position of the vertex from the point of view of the light and pass that in to the pixel shader for the depth test. This is accomplished as follows:

```
o.posLight = mul(pos, mSiLightAgent);
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;
```

The above code computes the vertex position from the point of view of the light, and puts it on the [0, 1] range, in order to do the lookup on the shadow map. The z value is slightly biased to prevent z fighting when the z value in the shadow map is equal to the current z value. If that is the case, the shadow test should determine that the pixel is not in shadow.

The pixel shader does the diffuse light computation for all of the scene lights. We only do shadow computation for one light (the key light, which provides most of the lighting contribution for our scene). However, the algorithm can be easily adapted to handle shadows for multiple source lights by using multiple shadow maps. One may be able reuse the same shadow map when rendering a shadow map for a second light, since this algorithm only makes use of two out of the four channels of the RGBA texture when using a single-channel opacity map. Here is the code to compute the diffuse contribution of the light that is attenuated by the shadow:

```
//diffuse lighting computation
float NdotL = dot(vNormal, vLight);

//read from shadow map
float4 t = tex2D(tShadowMap, i.posLight.xy);

//set shadow factor to the value it should attenuate light
//if it is in shadow (e.g., 0.0)
float3 vShadowFac = fOccluded;

//if light is NOT in shadow
if(i.posLight.z < t.a && NdotL > 0)
{
    //compute translucency.
    //Large values of transShadowAlpha make shadow darker
    float3 alpha = pow(t.rgb, transShadowAlpha);

    //set shadow factor to be a lerp between
    //shadowed color and white
    vShadowFac = lerp(fOccluded.xxx, float3(1.,1.,1.), alpha);
}

//attenuate light by the shadow factor
float3 diffuse = vShadowFac * saturate (NdotL * vLightColor);
```

Now, the light map contains the diffuse contribution of all lights and is ready to be blurred. During this pass, we also render the blur kernel size to the alpha com-

ponent of the light map. Therefore, only one texture needs to be accessed during the texture blur pass.

The Texture Blur

The next step in the process is to blur the 2D light map texture. This operation is performed in hardware by using a pixel shader. A number of filter kernels were examined to determine which gave acceptable results: Box, Gaussian, Poisson, and Kwasi. The Poisson disc filter was chosen because it gave good results for a reasonable cost and also allows a variable kernel size to be specified. The reason why this is important is described below.

The vertex shader for blurring is just a simple pass through; the pixel shader is where the main work is accomplished. The first part of the pixel shader is just filter kernel setup. In this case the filter kernel is composed of twelve samples generated via the Poisson distribution and the center sample.

```
float2 poisson[12] = {float2(-0.326212f, -0.40581f),
                      float2(-0.840144f, -0.07358f),
                      float2(-0.695914f, 0.457137f),
                      float2(-0.203345f, 0.620716f),
                      float2(0.96234f, -0.194983f),
                      float2(0.473434f, -0.480026f),
                      float2(0.519456f, 0.767022f),
                      float2(0.185461f, -0.893124f),
                      float2(0.507431f, 0.064425f),
                      float2(0.89642f, 0.412458f),
                      float2(-0.32194f, -0.932615f),
                      float2(-0.791559f, -0.59771f)};
```

The next part of the pixel shader reads from the center sample as well as the artist generated blur kernel size, which is stored in the alpha component of the light map. The other samples will be scaled by this kernel size in order to grow or shrink the blur kernel per texel. The `vPixelSize` variable represents the reciprocal of the width and height of the off-screen texture. The `vBlurScale` variable represents the scale and bias for the blur kernel texture.

```
    float2 vPixelSize (0.001953125, 0.001953125);
    float2 vBlurScale (3, 2);
    float4 ss = tex2D (tRenderedScenePong, i.texCoord);
    float2 pixelRadius = vPixelSize *
(ss.a*vBlurScale.x+vBlurScale.y);
```

The final part of the shader sums up the samples and divides by the number of samples, in this case thirteen. In addition, the pixel shader outputs the blur kernel size read from the center sample. This allows for further blurring passes if desired.

```
float3 cOut = ss.rgb;
for (int tap = 0; tap < 12; tap++)
{
```

```
        float4 s = tex2D (tRenderedScenePong, i.texCoord +
                          (poisson[tap] * pixelRadius));
        cOut += s.rgb;
    }
    return float4(cOut / 13.0f, ss.a);
```

By using two temporary buffers, we are able to perform several blurring passes on the diffuse illumination in order to achieve a soft, realistic look.

Variable Kernel Size

Borshukov and Lewis [Borshukov03] addressed translucency on the model's ears by ray-tracing. Unfortunately, in real time this is not currently an option. This visual cue is still important for achieving realistic-looking skin. When looking at a person who is backlit, typically you will see a lot of light passing through areas like the ears. In order to try to address this issue, our technique uses a blur kernel size texture that is artist generated. While it is theoretically possible to rearrange the texture coordinates to achieve a similar effect, this would require quite a bit of either artist or programmer time. The kernel scalar texture which was artist generated was a more time efficient approach. A blur kernel texture is shown in Figure 2.8.5.

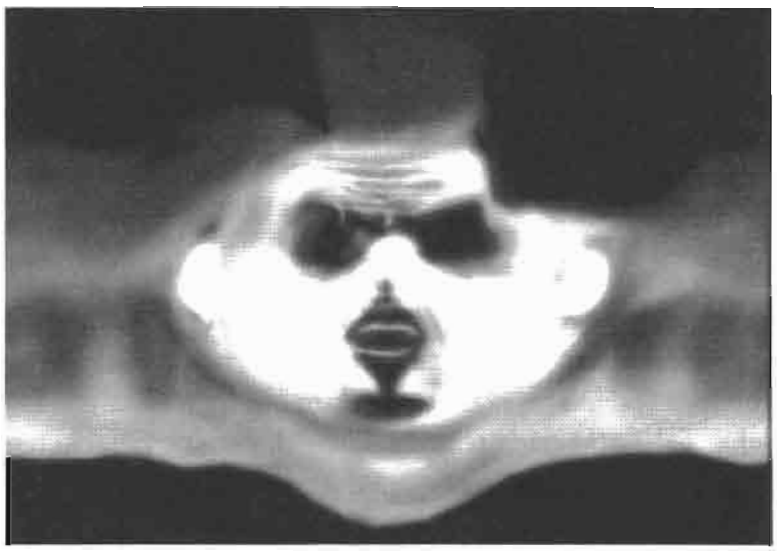


FIGURE 2.8.5 *The blur kernel for the model in Figures 2.8.4, 2.8.6, and 2.8.7.*

Texture Boundary Dilation

In order to prevent boundary artifacts when fetching from the light map, the texture needs to be dilated prior to blurring. We needed an efficient real-time solution to this problem. We accomplished this by modifying the Poisson disc filter shader to check whether a given sample is just outside the boundary of useful data, and if so, copy from an interior neighboring sample instead. We only need to use this modified, more expensive filter in the first blurring pass.

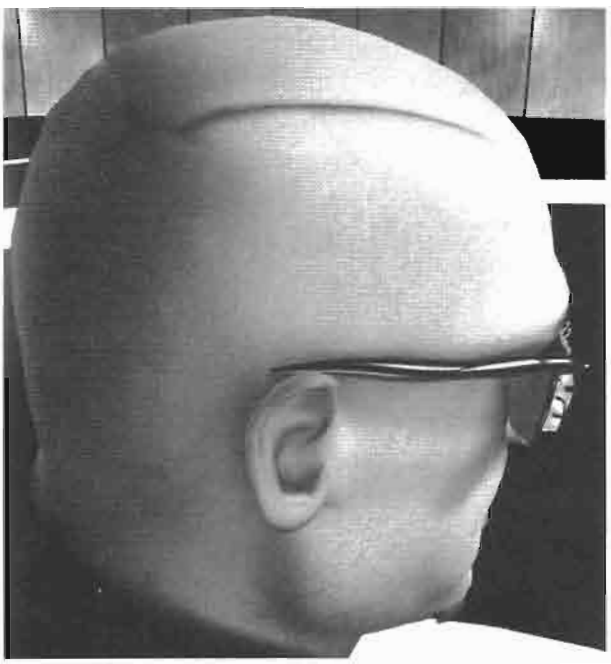
The pixel shader modifications to the blur shader needed to accomplish this dilation are relatively small. The first step is to determine if a particular texel in the off-screen light map has been written. Since the alpha of the off-screen light map is cleared to 1.0 before any processing and the light map rendering pass writes the blur kernel size into the alpha channel, as long as the blur kernel texture is never 1.0, we can use the alpha channel to determine which texels were written. The first step in the actual blur shader is to store off the center pixel alpha value (which represents blur kernel size). This happens before the main filter loop.

```
float flag = ss.a;
```

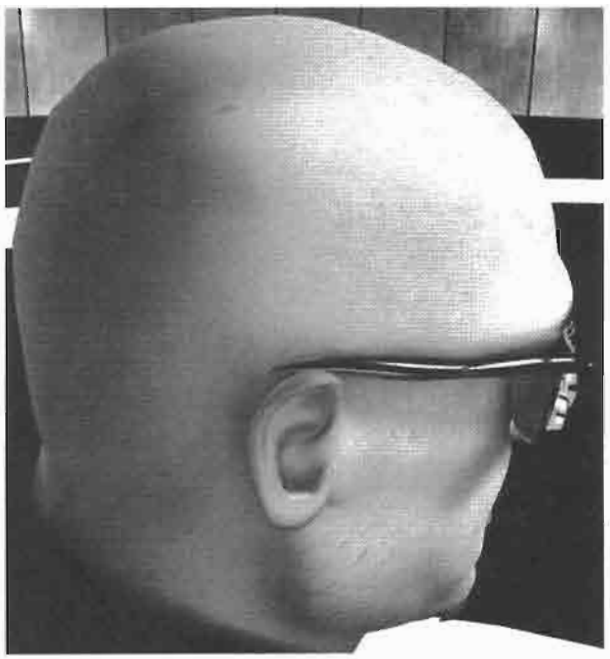
Within the filter loop this flag is updated by taking the maximum value of the current sample and the current flag. Additionally, the variable ss is updated if the current filter sample has a lower alpha value.

```
flag = max(s.a, flag);
if (s.a < ss.a)
{
    ss = s;
}
```

Finally, after the filter loop is finished, the flag is checked, and if a texel was found which was not written by the lighting shader, the sample with the lowest alpha (blur kernel size) is used instead. This gives the desired dilation effect (Figure 2.8.6).



(a) Without dilation



(b) With dilation

FIGURE 2.8.6 *Renderings with and without texture dilation. Images from ATI's demo* Ruby: The Double Cross. *© ATI Technologies 2004.*

```
if (flag == 1.0f)
    return float4(ss.rgb, 1.0f);
else
    return float4(cOut / 13.0f, ss.a);
```

It is possible to consider more complicated schemes for doing the dilation. For example, only samples that were written could be included in the filter average. In our case, however, this proved to be unnecessary to achieve the results desired.

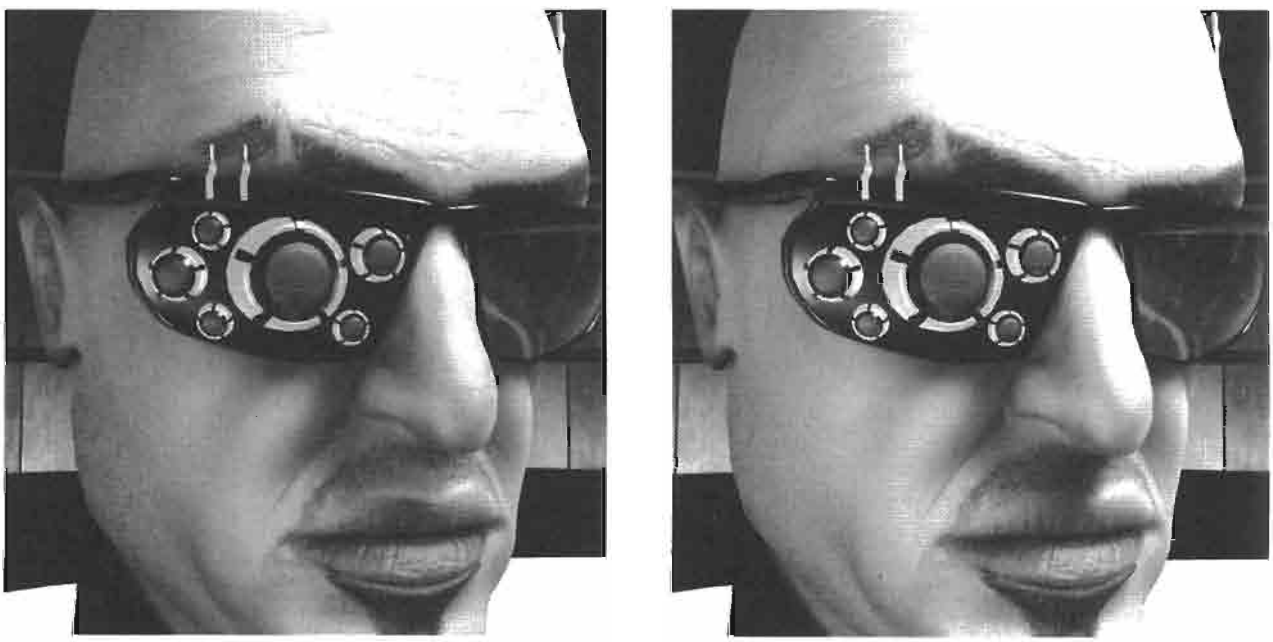
Render Final Mesh

After the light map has been blurred, we are ready to render the object. In this final rendering pass, we render the model with diffuse and specular lighting. When computing the diffuse illumination, we simply set it to be the value stored in the light map multiplied by the value in the color map for that surface point:

```
float4 cBase = tex2D (tBase, i.texCoord.xy);
float4 cLightMap = tex2D (tLightMap, i.texCoord);
float3 diffuse = cLightMap*cBase;
```

Specular Lighting with Shadows

Since it can be expensive to perform a separate blur pass for the shadow component, we cannot directly apply shadows to the specular illumination. We have found, however, that the luminance of the blurred light map can be used to attenuate the specular term of the shadow casting light to obtain a natural look. In Figure 2.8.7, note the



(a) Without shadowing the specular term (b) Shadowing the specular term

FIGURE 2.8.7 *Renderings with and without shadowing the specular term. Images from ATI's demo* Ruby: The Double Cross. © *ATI Technologies 2004.*

improvement on the shadows above the mouth of the model. In order to do this, the specular component is computed as follows:

```
//compute luminance of light map sample
float lum = dot(float3(0.2125, 0.7154, 0.0721), cLightMap.rgb);

//possibly scale and bias lum here depending on the light setup

//multiply specular by lum
specular *= lum;
```

Acceleration Techniques

In order to further optimize this technique, we employ hardware early-z culling to avoid processing regions of our light map that are outside of the view frustum, facing away from the viewer, or simply too far away to need updating. These optimizations significantly improve performance, since many or all pixels in the light map rendering and blurring passes can be skipped. We perform three inexpensive "z pre-passes" in texture space in order to set pixels in the texture-space z buffer to zero or one. These zeroes and ones in the z buffer don't represent distances but, rather, a logical value that indicates whether the pixel needs to be processed on subsequent passes. On those subsequent passes, these z values will drive the early-z culling hardware in order to avoid processing certain pixels [Krüger03].

Frustum Culling

Before rendering to the light map, we clear the z buffer to 1 and perform a very simple and cheap texture space rendering pass in which we just set the z value to 0 for all rendered samples. If the bounding box of the model's head lies outside the view frustum and is culled by the graphics engine, the z value is not modified. On all further texture-space passes, we set the z value to 0 in the vertex shader and the z test to "equal". This ensures that if the model lies outside the view frustum, hardware early-z culling will prevent all pixels from being processed.

Backface Culling

Similarly, in yet another cheap rendering pass, backface culling can be performed by setting the z value accordingly. This time, a dot product of the view vector and normal is computed in the vertex shader and passed on to the pixel shader. If the sample is frontfacing, a 0 is written to the z buffer, otherwise the pixel is clipped, leaving the z buffer untouched. We bias the result of the dot product by 0.3 so that samples that are "slightly" backfacing are not culled. This is because some of these samples may be within the blur radius of frontfacing samples. Figure 2.8.8 shows a texture with backface culled regions rendered in black. Here is the vertex shader code for the backface culling pre-pass:

```
//compute+bias the dot product of the view vector and the normal
//output result to a color interpolator
float3 viewVec = normalize ((float3)(worldCamPos-pos));
viewVec = normalize(viewVec);
o.dotp = dot(viewVec, i.normal);
o.dotp += 0.3;
```

and the pixel shader code for the z pre-pass:

```
float4 main (PsInput i) : COLOR
{
    //clip backfacing regions, otherwise return background color
    clip(i.dotp);
    return cBackColor;
}
```

We noticed that if we do backface culling on the ears of the model, sometimes a culled sample bleeds into a visible sample after blurring. That is because of the curvature on the ear is significantly higher than in other regions of the face. To address this problem, we added a texture coordinate check on the vertex shader which prevents culling the region around the ears.

(a) Blurred texture with backface culling

(b) Rendered model

FIGURE 2.8.8 *The texture space computation is culled on backfacing regions. Images from ATI's demo* Ruby: The Double Cross. © *ATI Technologies 2004.*

Distance Culling

If the model lies very far from the camera, the z value is set to 1, and the light map from the previously rendered frame is used (the light map buffer never needs to be