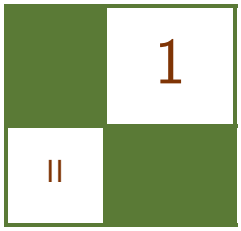

Pre-Integrated Skin Shading

Eric Penner and George Borshukov

1.1 Introduction

Rendering realistic skin has always been a challenge in computer graphics. Human observers are particularly sensitive to the appearance of faces and skin, and skin exhibits several complex visual characteristics that are difficult to capture with simple shading models. One of the defining characteristics of skin is the way light bounces around in the dermis and epidermis layers. When rendering using a simple diffuse model, the light is assumed to immediately bounce equally in all directions after striking the surface. While this is very fast to compute, it gives surfaces a very "thin" and "hard" appearance. In order to make skin look more "soft" it is necessary to take into account the way light bounces around inside a surface. This phenomenon is known as *subsurface scattering*, and substantial recent effort has been spent on the problem of realistic, real-time rendering with accurate subsurface scattering.

Current skin-shading techniques usually simulate subsurface scattering during rendering by either simulating light as it travels through skin, or by gathering incident light from neighboring locations. In this chapter we discuss a different approach to skin shading: rather than gathering neighboring light, we pre-integrate the effects of scattered light. Pre-integrating allows us to achieve the nonlocal effects of subsurface scattering using only locally stored information and a custom shading model. What this means is that our skin shader becomes just that: a simple pixel shader. No extra passes are required and no blurring is required, in texture space nor screen space. Therefore, the cost of our algorithm scales directly with the number of pixels shaded, just like simple shading models such as Blinn-Phong, and it can be implemented on any hardware, with minimal programmable shading support.

1.2 Background and Previous Work

Several offline and real-time approaches have been based on an approach, taken from film, called *texture-space diffusion* (TSD). TSD stores incoming light in texture space and uses a blurring step to simulate diffusion. The first use of this technique was by [Borshukov and Lewis 03,Borshukov and Lewis 05] in the *Matrix* sequels. They rendered light into a texture-space map and then used a custom blur kernel to gather scattered light from all directions. Based on extensive reference to real skin, they used different blur kernels for the red, green, and blue color channels, since different wavelengths of light scatter differently through skin. Since the texture-space diffusion approach used texture-blur operations, it was a very good fit for graphics hardware and was adopted for use in real-time rendering [Green 04, Gosselin et al. 04]. While TSD approaches achieved much more realistic results, the simple blurring operations performed in real time couldn't initially achieve the same level of quality of the expensive, original, nonseparable blurs used in film.

A concept that accurately describes how light diffuses in skin and other translucent materials is known as the *diffusion profile*. For a highly scattering translucent material it is assumed that light scatters equally in all directions as soon as it hits the surface. A diffusion profile can be thought of as a simple plot of how much of this diffused light exits the surface as a function of the distance from the point of entry. Diffusion profiles can be calculated using measured scattering parameters via mathematical models known as *dipole* [Jensen et al. 01] or *multipole* [Donner and Jensen 05] diffusion models. The dipole model works for simpler materials, while the multipole model can simulate the effect of several layers, each with different scattering parameters.

The work by [d'Eon and Luebke 07] sets the current high bar in real-time skin rendering, combining the concept of fast Gaussian texture-space diffusion with the rigor of physically based diffusion profiles. Their approach uses a sum of Gaussians to approximate a multipole diffusion profile for skin, allowing a very large diffusion profile to be simulated using several separable Gaussian blurs. More recent approaches have achieved marked performance improvements. For example, [Hable et al. 09] have presented an optimized texture-space blur kernel, while [Jimenez et al. 09] have applied the technique in screen space.

1.3 Pre-Integrating the Effects of Scattering

We have taken a different approach to the problem of subsurface scattering in skin and have departed from texture-space diffusion (see Figure 1.1). Instead, we wished to see how far we could push realistic skin rendering while maintaining the benefits of a local shading model. Local shading models have the advantage of not requiring additional rendering passes for each object, and scale linearly with the number of pixels shaded. Therefore, rather than trying to achieve subsur-

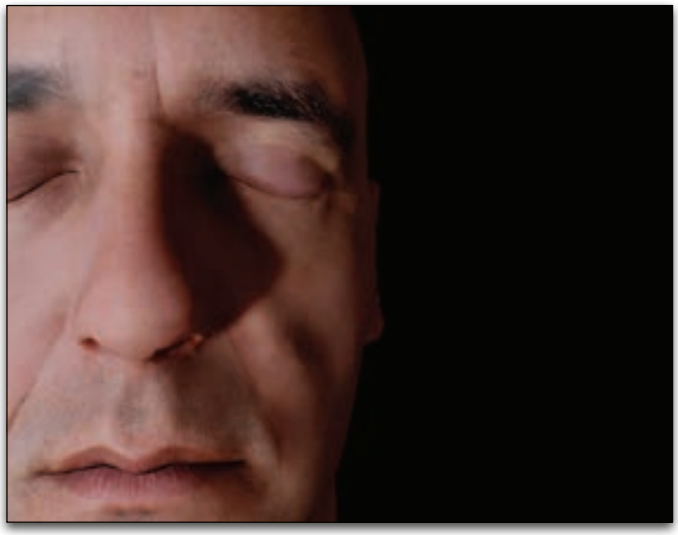

Figure 1.1. Our pre-integrated skin-shading approach uses the same diffusion profiles as texture-space diffusion, but uses a local shading model. Note how light bleeds over lighting boundaries and into shadows. (Mesh and textures courtesy of XYZRGB.)

face scattering by gathering incoming light from nearby locations (performing an integration during runtime), we instead seek to pre-integrate the effects of subsurface scattering in skin. Pre-integration is used in many domains and simply refers to integrating a function in advance, such that calculations that rely on the function's integral can be accelerated later. Image convolution and blurring are just a form of numerical integration.

The obvious caveat of pre-integration is that in order to pre-integrate a function, we need to know that it won't change in the future. Since the incident light on skin can conceivably be almost arbitrary, it seems as though precomputing this effect will prove difficult, especially for changing surfaces. However, by focusing only on skin rather than arbitrary materials, and choosing specifically where and what to pre-integrate, we found what we believe is a happy medium. In total, we pre-integrate the effect of scattering in three special steps: on the lighting model, on small surface details, and on occluded light (shadows). By applying all of these in tandem, we achieve similar results to texture-space diffusion approaches in a completely local pixel shader, with few additional constraints.

To understand the reasoning behind our approach, it first helps to picture a completely flat piece of skin under uniform directional light. In this particular case, no visible scattering will occur because the incident light is the same everywhere. The only three things that introduce visible scattering are changes in

the surrounding mesh curvature, bumps in the normal map, and occluded light (shadows). We deal with each of these phenomena separately.

1.4 Scattering and Diffuse Light

If we take our previously flat surface with no visible scattering and start to make it a smooth and curvy surface, like skin (we will keep it smooth for now), scattering will start to become visible. This occurs due to the changes in incident light across the surface. The Lambert diffuse-lighting model assumes that diffuse light scatters equally in all directions, and the amount of incident light is proportional to the cosine of the angle between the surface normal and light direction $(N \cdot L)$.

Since $N \cdot L$ falloff is a primary cause of changing incident light, and thus visible scattering, there have been several rendering tricks that attempt to add the look of scattering by altering the $N \cdot L$ fall-off itself. This involves making the falloff wrap around the back of objects, or by letting each wavelength of light (r, g, and b) fall off differently as $N \cdot L$ approaches zero. What we found to be the big problem with such approaches is that they aren't based on physical measurements of real skin-like diffusion profiles; and if you tune them to look nice for highly curved surfaces, then there will be a massive falloff for almost-flat surfaces (and vice versa).

To address both issues, we precompute the effect of diffuse light scattering. We do this in a fashion similar to measured *bidirectional reflectance distribution functions* (BRDFs). Measured BRDFs use physically measured data from real surfaces to map incoming to outgoing light. This is as opposed to analytical BRDFs such as Blinn-Phong that are analytical approximations for an assumed micro-facet structure. Typical measured BRDFs don't incorporate $N \cdot L$ since $N \cdot L$ just represents the amount of incoming light and isn't part of the surface's reflectance. We are concerned *only* with $N \cdot L$ (diffuse light), as this is the light that contributes to subsurface scattering.

One approach we considered to precompute the effect of scattered light at any point on a surface, was to simulate lighting from all directions and compress that data using spherical harmonics. Unfortunately, spherical harmonics can efficiently represent only very low frequency changes or would require too many coefficients. Thus, instead of precomputing the effect of scattering at all locations, we chose to precompute the scattering falloff for a subset of surface shapes and determine the best falloff during forward rendering. As discussed above, one characteristic that we can calculate in a shader is surface curvature, which largely determines the effect of scattering on smooth surfaces.

To measure the effect of surface curvature on scattering, we add curvature as the second parameter to our measured diffuse falloff. The skin diffusion profiles from [d'Eon and Luebke 07] on flat skin can be used to simulate the effect of scattering on different curvatures. We simply light a spherical surface of a

given curvature from one direction and measure the accumulated light at each angle with respect to the light (see Figures 1.2 and 1.3). This results in a two-dimensional lookup texture that we can use at runtime. More formally, for each skin curvature and for all angles θ between N and L, we perform the integration in Equation (1.1):

$$D(\theta, r) = \frac{\int_{-\pi}^{\pi} \cos(\theta + x) \cdot R(2r \sin(x/2))dx}{\int_{-\pi}^{\pi} R(2 \sin(x/2))dx} \tag{1.1}$$

The first thing to note is that we have approximated a spherical integration with integration on a ring. We found the difference was negligible and the ring integration fits nicely into a shader that is used to compute the lookup texture. The variable $R()$ refers to the diffusion profile, for which we used the sum of Gaussians from [d'Eon and Luebke 07] (see Table 1.1). Rather than performing an expensive arccos() operation in our shader to calculate the angle, we push this into the lookup, so our lookup is indexed by $N \cdot L$ directly. This is fine in our case, as the area where scattering occurs has plenty of space in the lookup. Figures 1.2 and 1.3 illustrate how to compute and use the diffuse lookup texture.

While this measured skin model can provide some interesting results on its own, it still has a few major flaws. Primarily, it assumes that all skin resembles a sphere, when, in fact, skin can have fairly arbitrary topology. Stated another way, it assumes that scattered light arriving at a given point depends on the curvature of that point itself. In actuality it depends on the curvature of all of the surrounding points on the surface. Thus, this approximation will work very well on smooth surfaces without fast changes in curvature, but breaks down when curvature changes too quickly. Thankfully, most models of skin are broken up into two detail levels: smooth surfaces represented using geometry, and surface

Variance	Red	Green	Blue
0.0064	0.233	0.455	0.649
0.0484	0.100	0.366	0.344
0.187	0.118	0.198	0.0
0.567	0.113	0.007	0.007
1.99	0.358	0.004	0.0
7.41	0.078	0	0.0

Table 1.1. The weights used by [d'Eon and Luebke 07] for texture-space diffusion. Although we aren't limited to the sum of Gaussians approximations, we use the same profile for comparison.

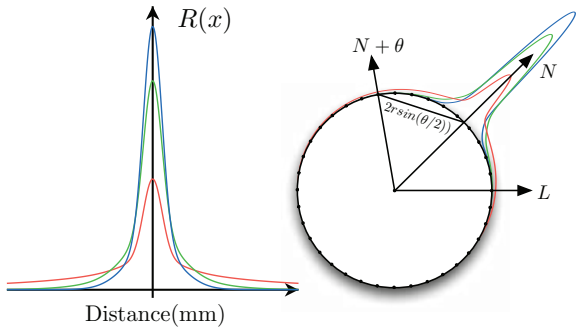

Figure 1.2. The graph (left) illustrates the diffusion profile of red, green, and blue light in skin, using the sum of Gaussians from Table 1.1. The diagram (right) illustrates how we pre-integrate the effect of scattering into a diffuse BRDF lookup. The diffusion profile for skin (overlaid radially for one angle) is used to blur a simple diffuse BRDF for all curvatures of skin.

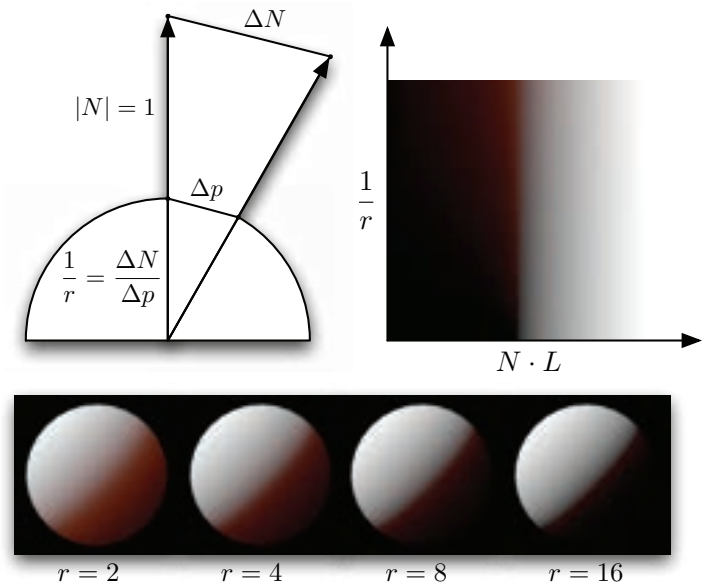

Figure 1.3. The diagram (top left) illustrates how we calculate curvature while rendering using two derivatives. The diffuse BRDF lookup, indexed by curvature (sphere radius) and $N \cdot L$ (top right). Spheres of different sized renderings using the new BRDF lookup (bottom).

details represented in a normal map. We take advantage of this and let the measured diffuse falloff be chosen at the smooth geometry level, while adding another approach to deal with creases and small bumps in normal maps, which are responsible for quick changes in curvature.

1.5 Scattering and Normal Maps

We now turn to the effect of scattering on small wrinkles and pores that are usually represented with a normal map. Since the normal from a small crease always returns to the dominant surface normal, the reflected scattered light coming from that crease will look very similar to light reflected from a nonscattering surface with a physically broader (or blurred-out) crease. Coincidentally, one way of approximating the look of scattered-over small creases and bumps is to simply blur the creases and bumps themselves! Most important however, this effect will be different for each wavelength of light, because of their different diffusion profiles.

Interestingly, the inverse of this phenomenon was noted when capturing normals from a real subject, using image-based lighting. Ma et al. [Ma et al. 07] noted that, when captured using spherical gradient lighting, normals become bent toward the dominant-surface normal, depending on the wavelength of light used to capture them (red was more bent than green, etc.). They also noted that a local skin-shading model was improved by using all the normals they captured instead of only one. In this case the image-based normal capture was physically integrating all the scattered light when determining the best normal to fit the data. Since we have only one set of normal maps to begin with, we essentially work in reverse. We assume that our original normal map is the accurate surface normal map and blur it several times for each wavelength, resulting in a separate normal map for each color, and for specular reflection. As mentioned by previous authors [d'Eon and Luebke 07, Hable et al. 09, Jimenez et al. 09], care should be taken to make sure the original normal map has not been blurred already in an attempt to get a smoother look.

While it might seem to make sense to simply blur the normal map using the diffusion profile of skin, this approach is not completely valid since lighting is not a linear process with regard to the surface normal. What we really want to have is a representation of the normal which can be linearly filtered, much in the same way that shadow maps can be filtered linearly using a technique like variance shadow mapping. Interestingly, there has been some very recent work in linear normal map filtering. *Linear efficient antialiased normal* (LEAN) mapping [Olano and Baker 10] represents the first and second moments of bumps in surface-tangent space. Olano and Baker focused primarily on representing specular light but also suggest a diffuse-filtering approximation from [Kilgard 00] which simply uses the linearly filtered unnormalized normal and standard diffuse lighting. It is noteworthy that the unnormalized normal is actually a valid approximation when

a bump self-shadowing and incident/scattered lighting term is constant over the normal-map filtering region. In that case,

$$\frac{1}{n}\Sigma_{i=1}^{n}(K_{\text{diffuse}}L \cdot N_i) = K_{\text{diffuse}}L \cdot (\frac{1}{n}\Sigma_{i=1}^{n}N_i).$$

The reason this isn't always the case is that diffuse lighting incorporates a self-shadowing term $\max(0, N \cdot L)$ instead of simply $N \cdot L$. This means back-facing bumps will actually contribute negative light when linearly filtered. Nonetheless, using the unnormalized normal will still be valid when all bumps are unshadowed or completely shadowed, and provides a better approximation than the normalized normal in all situations, according to [Kilgard 00].

Although we would prefer a completely robust method of pre-integrating normal maps that supports even changes in incident/scattered light over the filtering region, we found that blurring, using diffusion profiles, provided surprisingly good results (whether or not we renormalize). In addition, since using four normals would require four transformations into tangent space and four times the memory, we investigated an approximation using only one mipmapped normal map. When using this optimization, we sample the specular normal as usual, but also sample a red normal clamped below a tunable miplevel in another sampler. We then transform those two normals into tangent space and blend between them to get green and blue normals. The resulting diffuse-lighting calculations must then be performed three times instead of once. The geometry normal can even be used in place of the second normal map sample, if the normal map contains small details exclusively. If larger curves are present, blue/green artifacts will appear where the normal map and geometry normal deviate, thus the second mipmapped sample is required.

We found that this approach to handling normal maps complements our custom diffuse falloff very well. Since the red normal becomes more heavily blurred, the surface represented by the blurred normal becomes much more smooth, which is the primary assumption made in our custom diffuse falloff. Unfortunately, there is one caveat to using these two approaches together. Since we have separate normals for each color, we need to perform three diffuse lookups resulting in three texture fetches per light. We discuss a few approaches to optimizing this in Section 1.7.

1.6 Shadow Scattering

Although we can now represent scattering due to small- and large-scale features, we are still missing scattering over occluded light boundaries (shadows). The effect of light scattering into shadows is one of the most noticeable features of realistically rendered skin. One would think that scattering from shadows is much more difficult since they are inherently nonlocal to the surface. However, by using

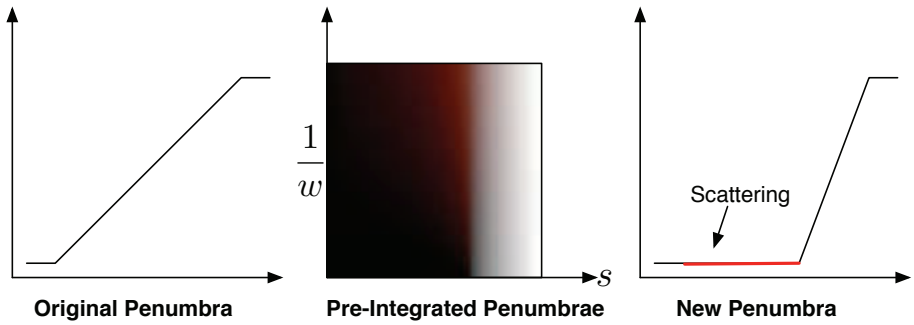

Figure 1.4. Illustration of pre-integrated scattering falloff from shadows. A typical shadow falloff from a box-filtered shadow map (left). A smaller penumbra that we pre-integrate against the diffusion profile of skin (right). The lookup maps the first penumbra into the second but also stores additional scattered light. The lookup is parameterized by the original shadow value and the width of the penumbra in world space (center).

a small trick, we found we could pre-integrate the effect of scattering over shadow boundaries in the same way we represent scattering in our lighting model.

The trick we use for shadows is to think of the results of our shadow mapping algorithm as a falloff function rather than directly as a penumbra. When the falloff is completely black or white, we know we are completely occluded or unoccluded, respectively. However, we can choose to reinterpret what happens between those two values. Specifically, if we ensure the penumbra size created by our shadow map filter is of adequate width to contain most of the diffusion profile, we can choose a different (smaller) size for the penumbra and use the rest of the falloff to represent scattering according to the diffusion profile (see Figure 1.4).

To calculate an accurate falloff, we begin by using the knowledge of the shape of our shadow mapping blur kernel to pre-integrate a representative shadow penumbra against the diffusion profile for skin. We define the representative shadow penumbra $P()$ as a one-dimensional falloff from filtering a straight shadow edge (a step function) against the shadow mapping blur kernel. Assuming a monotonically decreasing shadow mapping kernel, the representative shadow falloff is also a monotonically decreasing function and is thus *invertible* within the penumbra. Thus, for a given shadow value we can find the position within the representative penumbra using the inverse $P^{-1}()$. As an example, for the simple case of a box filter, the shadow will be a linear ramp, for which the inverse is also a linear ramp. More complicated filters have more complicated inverses and need to be derived by hand or by using software like Mathematica. Using the inverse, we can create a lookup texture that maps the original falloff back to its location

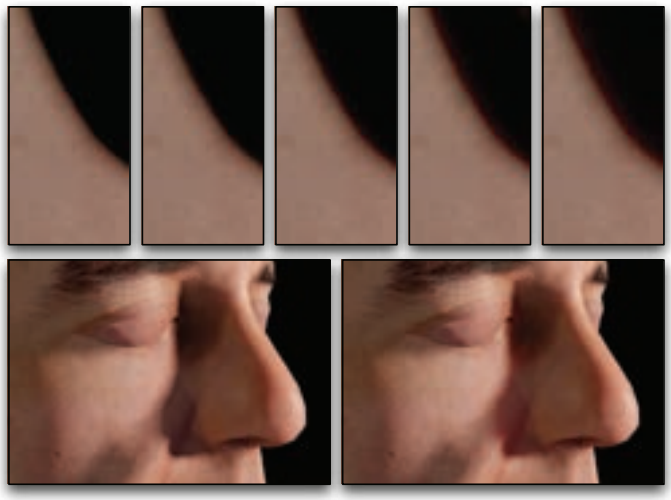

Figure 1.5. Illustration of pre-integrated scattering falloff from shadows. Controlled scattering based on increased penumbra width, such as a penumbra cast onto a highly slanted surface (top). Comparison with and without shadow scattering (bottom).

in the penumbra and then to a completely different falloff. Specifically, we can make the new shadow falloff smaller and use the remainder to represent subsurface scattering from the new penumbra. In the end, we are left with a simple integration to perform that we can use as a lookup during rendering, exactly like our diffuse falloff.

We should note at this point that we could run into problems if our assumptions from above are invalidated. We found that a two-dimensional shadow falloff was not noticeably different from a one-dimensional one, but we have also assumed that all shadow transitions are sharp. For example, if something like a screen door were to cast a shadow, it might result in a constant penumbra value between zero and one. In that case, we would assume there is scattering from a falloff that isn't there. Additionally, we have assumed projection onto a flat surface. If the surface is highly slanted, then the true penumbra size will be much larger than the one we used during pre-integration. For this reason we add a second dimension to our shadow falloff texture, which represents the size of the penumbra in world space. This is similar to the way we pre-integrate lighting against all sphere sizes. In the end, our two-dimensional shadow-lookup integration is a simple convolution:

$$P_S(s,w) = \frac{\int_{-\infty}^{\infty} P'(P^{-1}(s)+x)R(x/w)dx}{\int_{-\infty}^{\infty} R(x/w)dx},$$

where $P^{-1}()$ is the inverse of our representative falloff, $P'()$ is the new, smaller penumbra, $R()$ is the diffusion profile, and s and w are the shadow value and penumbra width in world space, respectively. Penumbra width can be detected using either the angle of the surface with respect to the light, or potentially the derivative of the shadow value itself. Since creating a large penumbra is expensive using conventional shadow filtering (although, see the *Pixel Quad Amortization* chapter), having the penumbra stretched over a slanted surface provides a larger space for scattering falloff and thus can actually be desirable if the initial shadow penumbra isn't wide enough. In this case the lookup can be clamped to insure that the falloff fits into the space provided.

1.7 Conclusion and Future Work

We have presented a new local skin-shading approach based on pre-integration that approximates the same effects found in more expensive TSD-based approaches. Our approach can be implemented by adding our custom diffuse- and shadow-falloff textures to a typical skin shader (see Figure 1.6).

Although we found that our approach worked on a large variety of models, there are still a few drawbacks that should be mentioned. When approximating curvature using pixel shader derivatives, triangle edges may become visible where curvature changes quickly. Depending on how the model was created we also found that curvature could change rapidly or unnaturally in some cases. We are looking into better approaches to approximating curvature in these cases. This is much more easily done with geometry shaders that can utilize surface topology.

Figure 1.6. Comparison of our approach with texture-space diffusion using an optimized blur kernel from [Hable et al. 09]. (Mesh and textures courtesy of XYZRGB.)

We would also like to look at the effect of using more than one principal axis of curvature. For models where curvature discontinuities occur, we generate a curvature map that can be blurred and further edited by hand, similar to a stretch map in TSD.

Another challenge we would like to meet is to efficiently combine our normal map and diffuse-lighting approaches. When using three diffuse normals, we currently need three diffuse-texture lookups. We found we could use fewer lookups depending on the number of lights and the importance of each light. We have also found it promising to approximate the diffuse and shadow falloffs using analytical approximations that can be evaluated without texture lookups.

We would also like to apply our technique to environment mapping. It should be straightforward to support diffuse-environment mapping via an array of diffuse-environment maps that are blurred based on curvature, in the same manner as our diffuse-falloff texture.

1.8 Appendix A: Lookup Textures

```
float  Gaussian (float  v, float r)
{
    return 1.0/sqrt(2.0*PI*v) * exp(-(r*r)/(2*v));
}

float3 Scatter(float r)
{
   //Coefficients from GPU Gems 3 - ''Advanced Skin Rendering
       .''
   return Gaussian(0.0064 * 1.414,r) * float3
       (0.233,0.455,0.649) +
           Gaussian(0.0484 * 1.414,r) * float3
               (0.100,0.336,0.344) +
           Gaussian(0.1870 * 1.414,r) * float3
               (0.118,0.198,0.000) +
           Gaussian(0.5670 * 1.414,r) * float3
               (0.113,0.007,0.007) +
           Gaussian(1.9900 * 1.414,r) * float3
               (0.358,0.004,0.000) +
           Gaussian(7.4100 * 1.414,r) * float3
               (0.078,0.000,0.000);
}

float3 integrateShadowScattering(float penumbraLocation,
                                 float penumbraWidth)
{
    float3 totalWeights = 0;
    float3 totalLight = 0;

    float a= -PROFILE_WIDTH;
```

```
    while( a<=PROFILE_WIDTH )
    while( a<=PROFILE_WIDTH )
    {
        float light = newPenumbra(penumbraLocation + a/
            penumbraWidth);
        float sampleDist = abs(a);
        float3 weights = Scatter(sampleDist);
        totalWeights += weights;
        totalLight += light * weights;
        a+=inc;
    }

    return totalLight / totalWeights;
}

float3 integrateDiffuseScatteringOnRing(float cosTheta,float
    skinRadius)
{
    // Angle from lighting direction.
    float  theta = acos(cosTheta);
    float3 totalWeights = 0;
    float3 totalLight = 0;

    float a= -(PI/2);
    while( a<=(PI/2) )
    while( a<=(PI/2) )
    {
        float sampleAngle = theta + a;
        float diffuse = saturate( cos(sampleAngle) );
        float sampleDist = abs(2.0*skinRadius*sin(a*0.5));
            // Distance.
        float3 weights = Scatter(sampleDist);
            // Profile Weight.
        totalWeights += weights;
        totalLight += diffuse * weights;
        a+=inc;
    }
    return totalLight / totalWeights;
}
```

Listing 1.1. Shader code to precompute skin falloff textures.

1.9 Appendix B: Simplified Skin Shader

```
float3 SkinDiffuse( float curv, float3 NdotL )
{
    float3 lookup = NdotL * 0.5 + 0.5;
    float3 diffuse;
```

```
    diffuse.r = tex2D(SkinDiffuseSampler, float2(lookup.r, curv
        )).r;
    diffuse.g = tex2D(SkinDiffuseSampler, float2(lookup.g, curv
        )).g;
    diffuse.b = tex2D(SkinDiffuseSampler, float2(lookup.b, curv
        )).b;
    return diffuse;
}

float3 SkinShadow( float shad, float width )
{
    return tex2D(SkinShadowSampler, float2(shad, width)).rgb;
}
...
//Simple curvature calculation.
float curvature = saturate(length(fwidth(Normal)) /
                  length(fwidth(WorldPos)) * tuneCurvature );
...
//Specular/Diffuse Normals.
float4 normMapHigh   = tex2D(NormalSamplerHigh, Uv) * 2.0 -
    1.0;
float4 normMapLow    = tex2D(NormalSamplerLow , Uv) * 2.0 -
    1.0;
float3 N_high = mul(normMapHigh.xyz,TangentToWorld);
float3 N_low  = mul(normMapLow.xyz,TangentToWorld);
float3 rS = N_high;
float3 rN = lerp(N_high,N_low,tuneNormalBlur.r);
float3 gN = lerp(N_high,N_low,tuneNormalBlur.g);
float3 bN = lerp(N_high,N_low,tuneNormalBlur.b);
...
//Diffuse lighting
float3 NdotL = float3( dot(rN,L), dot(gN,L), dot(bN,L) );
float3 diffuse = SkinDiffuse( curvature, NdotL ) * LightColor *
                 SkinShadow( SampleShadowMap(ShadowUV) );
```

Listing 1.2. Skin shader example.

Bibliography

[Borshukov and Lewis 03] George Borshukov and J.P. Lewis. "Realistic Human Face Rendering for The Matrix Reloaded." In *ACM Siggraph Sketches and Applications*. New York: ACM, 2003.

[Borshukov and Lewis 05] George Borshukov and J.P. Lewis. "Fast Subsurface Scattering." In *ACM Siggraph Course on Digital Face Cloning*. New York: ACM, 2005.

[d'Eon and Luebke 07] E. d'Eon and D. Luebke. "Advanced Techniques for Realistic Real-Time Skin Rendering." In *GPU Gems 3*, Chapter 14. Reading, MA: Addison Wesley, 2007.

[Donner and Jensen 05] Craig Donner and Henrik Wann Jensen. "Light Diffusion in Multi-Layered Translucent Materials." *ACM Trans. Graph.* 24 (2005), 1032–1039.

[Gosselin et al. 04] D. Gosselin, P.V. Sander, and J.L. Mitchell. "Real-Time Texture-Space Skin Rendering." In *ShaderX^3: Advanced Rendering with DirectX and OpenGl*. Hingham, MA: Charles River Media, 2004.

[Green 04] Simon Green. "Real-Time Approximations to Subsurface Scattering." In *GPU Gems*, pp. 263–278. Reading, MA: Addison-Wesley, 2004.

[Hable et al. 09] John Hable, George Borshukov, and Jim Hejl. "Fast Skin Shading." In *ShaderX^7: Advanced Rendering Techniques*, Chapter II.4. Hingham, MA: Charles River Media, 2009.

[Jensen et al. 01] Henrik Jensen, Stephen Marschner, Mark Levoy, and Pat Hanrahan. "A Practical Model for Subsurface Light Transport." In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, *SIGGRAPH '01*, pp. 511–518. New York: ACM, 2001.

[Jimenez et al. 09] Jorge Jimenez, Veronica Sundstedt, and Diego Gutierrez. "Screen-Space Perceptual Rendering of Human Skin." *ACM Transactions on Applied Perception* 6:4 (2009), 23:1–23:15.

[Kilgard 00] Mark J. Kilgard. "A Practical and Robust Bump-Mapping Technique for Today's GPUs." In *GDC 2000*, 2000.

[Ma et al. 07] W.C. Ma, T. Hawkins, P. Peers, C.F. Chabert, M. Weiss, and P. Debevec. "Rapid Acquisition of Specular and Diffuse Normal Maps from Polarized Spherical Gradient Illumination." In *Eurographics Symposium on Rendering*. Aire-la-Ville, Switzerland: Eurographics Association, 2007.

[Olano and Baker 10] Marc Olano and Dan Baker. "LEAN mapping." In *ACM Siggraph Symposium on Interactive 3D Graphics and Games*, pp. 181–188. New York: ACM, 2010.