Implementing a skin BSSRDF (or several...)

Christophe Hery
Industrial Light + Magic

Introduction

In the seminal paper from 2001: *A Practical Model for Subsurface Light Transport* ([Jensen '01]), Henrik Wann Jensen et al employed the concept of a BSSRDF (a bidirectional surface scattering distribution function) as a means to overcome the limitations of BRDFs.

The following year, in *A Rapid Hierarchical Rendering Technique for Translucent Materials* ([Jensen '02]), Jensen and Buhler presented a caching method to accelerate the computation of the BSSRDF.

We will try here to give pointers about how to implement such systems, as well as how to use them in production.

1 BSSRDF 101

Skin is a multi-layered medium, in which photons tend to penetrate and scatter around many many times before bouncing back out. Certain wavelengths are attenuated differently according to the thickness. The path the photons take is highly complex and depends on the nature of the substrates as well as their energy. But one thing is almost certain: they rarely leave through the location where they entered, which is why a BSSRDF - a model for transport of light through the surface - is needed.

The full BSSRDF system presented by Jensen et al. consists of two terms: a single scattering component, which through path tracing gives us an approximate solution for the cases where light bounced only once inside the skin, and a multiple scattering component, through a statistical dipole point source diffusion approximation. The beauty of this model is that it can be implemented in a simple ray-tracer, and with the use of some trickery, in a Z-buffer renderer.

We are now going to go over the papers and extract the relevant information for shader writers, so please have your Proceedings handy.

1.1 Single scattering

We are trying to consider the different paths photons could take such that they would exit towards the camera at the shaded point after one unique bounce inside the skin. Since we know the camera viewpoint $\underline{\mathsf{I}}$ at the shaded surface position $\underline{\mathsf{P}}$, we can compute the refracted outgoing direction T_o. So let's select some samples P_{samp} along T_o

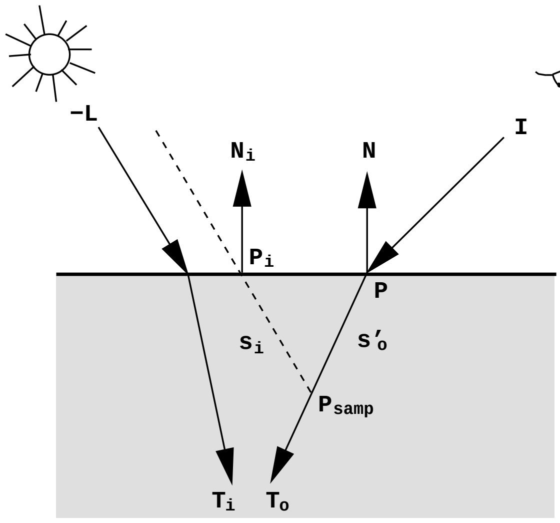

Figure 1: For single scattering, we march along the refracted outgoing ray T_o and project towards L.

(we'll figure out the density and the placement of those samples later), and let's try to find out how the light could have bounced back at those positions.

On the way in, the light should have refracted at the surface boundary. As mentioned in [Jensen '01], this is a hard problem for arbitrary geometry, and we make the approximation that the light did not refract. With this simplification, we can now find for each P_{samp} where the light first entered by simply tracing towards the light source.

In fact, we will not really trace towards the source position. Given the distances involved, we can consider the light to be uniform in direction and in color/intensity over the surface. So we simply trace along $\underline{\mathsf{L}}$ (the incident light direction at $\underline{\mathsf{P}}$) and multiply our result globally with $\underline{\mathsf{Cl}}$ (light energy received at $\underline{\mathsf{P}}$).

At this stage, we have: the distance the light traveled on the way in (s_i), and the distance the light traveled going out (s'_o). Prime signs here indicate, where appropriate, the refracted values. For instance, s'_o is the distance along the refracted outgoing ray, T_o, and it is a given, since we originally explicitly placed our sample P_{samp} along T_o from $\underline{\mathsf{P}}$ (in other words, $\mathsf{P}_{samp} = \underline{\mathsf{P}} + \mathsf{T}_o s'_o$).

The outscattered radiance for the single term, L_o, depends on the scattering coefficients, the phase function between the refracted directions, and some exponential falloff terms on s'_i and s'_o. But we do not know s'_i, the refracted distance before the bounce. The trick here is to use Snell's law to derive s'_i from s_i.

The other thing we are missing is the phase function. What is it? Simply stated, the phase function between two vectors $\vec{v}_1$ and $\vec{v}_2$ is a way to describe the amount of light scattered from the direction $\vec{v}_1$ into the direction $\vec{v}_2$. Knowing fully that we are not simulating, in CG, the light interactions one photon at a time, this phase becomes a probability density function (PDF), and for skin, it has been shown (for instance in [Pharr '01]) that the Henyey-Greenstein model is adequate.

Finally, we can try to make the solution converge faster. A naive implementation would have all P_{samp} uniformly selected along T_o. But this would require many many samples.

Let's look at L_0:

$$L_o(x_o,\vec{\omega}_o) = \frac{\sigma_s(x_o)F\,p(\vec{\omega}_i'\cdot\vec{\omega}_o')}{\sigma_{tc}} e^{-s_i'\sigma_t(x_i)} e^{-s_o'\sigma_t(x_o)} L_i(x_i,\vec{\omega}_i).$$

This expression can clearly be rewritten as the product of the exponential falloff in s_o' with another function:

$$L_o(x_o,\vec{\omega}_o) = [L_i(x_i,\vec{\omega}_i)\frac{\sigma_s(x_o)F\,p(\vec{\omega}_i'\cdot\vec{\omega}_o')}{\sigma_{tc}} e^{-s_i'\sigma_t(x_i)}] e^{-s_o'\sigma_t(x_o)}.$$

If we now pick the samples according to the PDF

$$x \sim \sigma_t e^{-\sigma_t x},$$

the integration in L_0 can be approximated by a simple summation. This is called importance sampling. By choosing

$$s_o' = \frac{-\log(random())}{\sigma_t},$$

we respect our PDF, and we can sum up all contributions without explicitly computing the falloff term in s_o'.

In pseudo shading language code, this becomes:

```
float singlescatter = 0;

vector To = normalize(refract(I,N,oneovereta));

for (i=0; i<nbrsamp; i+=1)
{
    float sp_o = -log(random())/sigma_t;
    point Psamp = P + To * sp_o;
    point (Pi,Ni) = trace(Psamp, L);
    float si = length(Psamp - Pi);
    float LdotNi = L.Ni;
    float sp_i = si * LdotNi
        / sqrt (1 - oneovereta*oneovereta * (1 - LdotNi*LdotNi));
    vector Ri, Ti; float Kri, Kti;
    fresnel (-L, Ni, oneovereta, Kri, Kti, Ri, Ti);
    Kti = 1-Kri;
    Ti  = normalize(Ti);
    float g2 = g*g;
    float phase = (1-g2) / pow(1+2*g*Ti.To+g2,1.5);
    singlescatter += exp(-sp_i*sigma_t) / sigma_tc * phase * Kti;
}

singlescatter *= Cl * PI * sigma_s / nbrsamp;
```

There is also a fresnel outgoing factor, but since it is to be applied for both the single and the diffusion terms, we are omitting it in this section.

It is that simple (in fact it is made simpler here, because we only deal with one wavelength)!

Lastly, note the trace call. It returns the position, the normal and potentially the textured diffuse color of the intersection of the ray going from P_{samp} in the $\underline{\mathsf{L}}$ direction. We will come back to it in section 3.

1.2 Diffusion scattering

We are trying to accumulate the observed profile of illumination over all positions on the surface. The dipole of point sources is a virtual construction to simulate the light attenuation on skin reported by medical scientists (the derivation of this model is beyond the scope of our discussion). The value we are summing up, at each sample, is the diffuse reflectance:

$$R_d(r) = \frac{\alpha'}{4\pi}[z_r(\sigma_{tr}d_r+1)\frac{e^{-\sigma_{tr}d_r}}{d_r^3} + z_v(\sigma_{tr}d_v+1)\frac{e^{-\sigma_{tr}d_v}}{d_v^3}]$$

Its expression mainly depends on the distance r from the shaded point to the sample, and also (directly and indirectly) on the scattering coefficients. The problem is once again the distribution of those samples. In fact, in [Jensen '02], a two-pass predistribution technique was shown (and we will come back to it in section 4). For now, let's assume that we want to place those samples at their appropriate locations during shading time. They need to lie on the surface and be located around $\underline{\mathsf{P}}$. The $\sigma_{tr}e^{-\sigma_{tr}d}$ terms are likely going to dominate the computation of R_d (both in times and in values). So we should try to put in a lot of effort in order to make them converge faster. Again we turn to the importance sampling theory, and we choose a PDF

$$x \sim \sigma_{tr}^2 e^{-\sigma_{tr}x}.$$

Why do we have a σ_{tr}^2 coefficient this time (where our PDF for single scattering had a simple σ_t multiplicator)? Well, we need to integrate in polar coordinates (whereas for single scattering we were marching along T_o in 1D), and a condition for a valid PDF is that its integral over the full domain comes to 1 (meaning that we have 100% chance to find a sample in the infinite range). This PDF corresponds to the cumulative density function (CDF)

$$P(R) = \int_0^R \sigma_{tr}^2 e^{-\sigma_{tr}r} r dr = 1 - e^{-\sigma_{tr}R}(1+\sigma_{tr}R)$$

By the change of variable $u = \sigma_{tr}R$, we get

$$P(u) = 1 - e^{-u}(1+u),$$

and through a numerical inversion of $P(u)$, we get a table of precomputed positions, that we can insert in the shader (we provide a C program in the Appendix that warps a uniform Poisson distribution into the one we want).

Again let's deal with the pseudo-code for one wavelength.

```
float diffusionscatter = 0;

uniform float scattdiffdist1[maxSamples] = { /* X table */ };
uniform float scattdiffdist2[maxSamples] = { /* Y table */ };

/* create a local frame of reference for the distribution */
vector  local = N;
vector  up    = vector "world"(0,1,0);
if (abs(local.up) > 0.9)
    up    = vector "world"(1,0,0);
vector  base1 = normalize(local^up);
vector  base2 = local^base1;

for (i=0; i<nbrsamp; i+=1)
{
    point Psamp = P + 1/sigma_tr *
        (base1 * scattdiffdist1[i] + base2 * scattdiffdist2[i]);
    point (Pi,Ni) = trace(Psamp, -local);
        /* make sure we are on the surface */

    float r = distance(P, Pi);
    float zr = sqrt(3.0*(1.0-alpha_prime)) / sigma_tr;
    float zv = A * zr;
    float dr = sqrt(r*r+zr*zr); /* distance to positive light */
    float dv = sqrt(r*r+zv*zv); /* distance to negative light */
    float sigma_tr_dr = sigma_tr*dr;
    float sigma_tr_dv = sigma_tr*dv;
    float Rd = (sigma_tr_dr+1.0) * exp(-sigma_tr_dr) * zr/(dr^3)
             + (sigma_tr_dv+1.0) * exp(-sigma_tr_dv) * zv/(dv^3);

    scattdiff += L.Ni * Rd / (sigma_tr*sigma_tr * exp(-sigma_tr*r));
        /* importance sampling weighting */
}

scattdiff *= Cl * (1-Fdr) * alpha_prime / nbrsamp;
```

Please be aware that, in this example, the distribution was done on the tangent plane at $\underline{P}$. Depending on the chosen tracing approach to reproject the samples on the actual geometry, this might not be the best solution.

2 Texture mapping and reparameterization

So far we have considered a shader that depends on some scattering coefficients. In reality, we need to texture them over our geometry. Painting them directly is non-trivial, and especially non-intuitive. In production, we tend to start from photographs of our live characters or maquettes. The question is: can we extract the variation of the relevant parameters from regular diffuse texture maps?

We are going to make some simplifications to make this conversion. First, we will assume that the influence of the single scattering is negligible compared to the diffusion term ([Jensen '02] provides a justification for skin materials). So let's assume that our diffuse texture map is the result of the diffusion scattering events under a uniform incident illumination, and try to solve for the scattering parameters which produce these colors.

In this case, we can approximate our BSSRDF as a BRDF:

$$R_d = \frac{\alpha'}{2}(1+e^{-\frac{4}{3}A\sqrt{3(1-\alpha')}})e^{-\sqrt{3(1-\alpha')}},$$

which only depends on α' and A (indirectly η). This is also equal, by construction, to our diffuse map $\mathsf{C}_{\mathsf{map}}$. So we now have one equation with two unknowns: α' and η. If we fix η (at 1.3 for skin), we can get α' as a function of $\mathsf{C}_{\mathsf{map}}(= R_d)$. In truth, we cannot do this analytically, but we can build yet another table numerically.

We used the following code in Matlab to produce the table:

```
clear;
eta = 1.3;
Fdr = -1.440/(eta.^2) + 0.710/eta + 0.668 + 0.0636*eta;
A = (1 + Fdr)/(1 - Fdr);
alpha = 0:.01:1;
c = alpha.*(1+exp(-4/3*A*sqrt(3*(1-alpha))))
         .* exp(-sqrt(3*(1-alpha)));
C = 0:.001:2;
ALPHA = interp1(c,alpha,C);
```

And, in the shader, we can lookup those values in the following manner:

```
uniform float alpha_1_3[2001] = { /* put matlab table here */ };
float alpha_prime = alpha_1_3[floor(diffcolor*2000.0)];
```

Because α' is the reduced transport albedo, namely the ratio:

$$\alpha' = \frac{\sigma_s'}{\sigma_a + \sigma_s'},$$

this does not resolve directly into the scattering coefficients.

Even though σ_s' is given for some specific materials in [Jensen '01], it is still a very non-intuitive parameter, hence we replace our tuning factor with the diffuse mean free path $ld = 1/\sigma_{tr}$. It turns out that σ_{tr} and α' are all that is needed to control the diffusion scattering. For the single scattering, we can get:

$$\sigma_t' = \frac{1}{ld\sqrt{3\,(1-\alpha')}}$$

and

$$\sigma_s' = \alpha'\,\sigma_t'.$$

Lastly, it is worth mentioning that, ideally, the color inversion should be done at every sample for diffusion scattering, so that we get the correct R_d on those positions. For single scattering, the dependence on the inversion during the ray marching is embedded in the σ_{tc} denominator and σ_t term in the exponential.

3 BSSRDF through Z-buffers

If we can manage to *trace* only along the light direction, we can achieve a projection towards $\underline{\mathsf{L}}$ by some simple operations on the shadow buffers, of the kind:

```
uniform matrix shadCamSpace, shadNdcSpace;
textureinfo(shadMap, "viewingmatrix", shadCamSpace);
textureinfo(shadMap, "projectionmatrix", shadNdcSpace);
uniform matrix shadInvSpace = 1/shadCamSpace;
point Pi       = transform(shadCamSpace, Psamp);
point tmpPsamp = transform(shadNdcSpace, Psamp);
float shadNdcS = (1.0+xcomp(tmpPsamp)) * 0.5;
float shadNdcT = (1.0-ycomp(tmpPsamp)) * 0.5;
float zmap = texture(shadMap,
                     shadNdcS, shadNdcT, shadNdcS, shadNdcT,
                     shadNdcS, shadNdcT, shadNdcS, shadNdcT,
                     "samples", 1);
if (comp(shadNdcSpace, 3, 3) == 1) /* orthographic */
    setzcomp(Pi, zmap);
else
    Pi *= zmap/zcomp(Pi);
Pi = transform(shadInvSpace, Pi);
```

To access any other values at P_i (like N_i), if we produced an image of those values from the point of view of the light (for instance through a secondary display output), we simply lookup the texture, as in:

```
color CNi = texture(shadCNMap,
                    shadNdcS, shadNdcT, shadNdcS, shadNdcT,
                    shadNdcS, shadNdcT, shadNdcS, shadNdcT,
                    "samples", 1);
normal Ni;
setxcomp(Ni, comp(CNi,0));
setycomp(Ni, comp(CNi,1));
setzcomp(Ni, comp(CNi,2));
Ni = ntransform(shadInvSpace, Ni);
```

Of course, if we want to emulate the soft shadows cast from a non-point light source, we can super-sample our shadNdcS and shadNdcT lookup coordinates or any relevant tracing value in the shader.

This overall tracing method drops right in for single scattering (the trace is indeed along $\underline{\mathsf{L}}$), but it is more complex to make it work for diffusion scattering.

Let's think about it for a minute. We need to place some samples on the surface according to a specific distribution. Our assumption was that we would first position them on the tangent plane at $\underline{\mathsf{P}}$, then project them (along $-\underline{\mathsf{N}}$) to the surface. With our Z-buffers, all we have is light projections. So what if we distributed our samples in shadow space, then projected along $-\underline{\mathsf{L}}$? The surface distribution would definitely be skewed. Our importance sampling mechanism would not converge as fast as expected (it would still be faster than a simple uniform distribution though). In fact, by doing it this way, it is almost as if we would be building implicitly the $\underline{\mathsf{L}} \cdot \underline{\mathsf{N}}$ product. We should then just "correct" the final result by omitting this factor.

In our pseudo-code from section 3, we just change:

- `vector local = N` into `vector local = L`,
- the trace with the method exposed above,
- the last line of the loop into `scattdiff += Rd / (sigma_tr*sigma_tr * exp(-sigma_tr*r));`

In practice, this twist on the theory works really well.

Ideally, we need 3 distributions (one per wavelength), ie 3 sets of fake trace Z-buffer projections. Not only is it a slow process (after all, `texture` is one of the most expensive operators in the shading language), but doing it that way creates a separation of the color channels, hence a very distracting color grain. The remedy is to do only one distribution with the minimum σ_{tr} (the wavelength that scatters the most).

Finally, since we use the shadow buffers as a way to represent the surface of our skin, we should not include in them any other geometry, like clothing or anything casting shadows from an environment (more on that later at paragraph 5.1). For the same reason, the resolution of the Z-buffers matters a great deal.

4 BSSRDF through a point cloud cache

As we said before (in section 1.2), the diffusion scattering term largely dominates the BSSRDF in the case of human skin rendering, to the point where we should think about dropping single scatter altogether from our model of skin. Single scatter can be made to produce good results though. One can balance it, tweak it, blur it, contain it with some area maps, so that its appearance is pleasing. However, from its very nature of being a directional term, the transitions of the illumination (especially the transitions from back lighting to front lighting) are very hard. For other materials with more isotropic scattering properties (scattering eccentricity closer to 0), such as marble, it becomes more critical to keep this term. In any case, when we speak about a skin BSSRDF from this section on, we will only consider the diffusion scattering.

Since the BSSRDF computation is fairly expensive, it is worth looking at acceleration methods. One observation we can make is that, no matter the pattern of distributions of the dipole samples, they are likely going to be extremely close from one shaded point to the next, yet we do indeed recompute their positions for each $\underline{\mathsf{P}}$. So one idea here would be to pre-distribute the samples on the surface, this time uniformly, then ”gather” them during the scattering loop. This is what [Jensen ’02] demonstrates.

There are several approaches one can take for this pre-distribution phase. One can build fancy Xsi scripts or mel commands that could achieve, through particle repulsion algorithms, a stabilized semi-Poisson set on the surface. One can implement a stand-alone Turk method ([Turk 1992]) as suggested in [Jensen ’02]. Or one can take advantage of a specific rendering engine. For example, in the case of Pixar’s Photorealistic Renderman, one can derive the positions directly from the micro-polygon grids through a dso (a la Matt Pharr’s `http://graphics.stanford.edu/~mmp/code/dumpgrids.c`), or more directly with the native bake3d() shadeop. All in all, there are certainly many more ways to obtain a similar point cloud representation of the geometry.

So let’s assume that we managed to pre-generate this distribution, and that it is stored in a file for which we know the format. The relevant information for each sample should be its location, the normal, the coverage area it represents - in other words its local sampling density (this is necessary in the cases for which our distribution is not exactly uniform) - and, if possible, the diffuse texture at that position.

The first thing to do will be to compute the diffuse irradiance at each point, and store it back in the data structure. Again there are several ways to do this (in fact, if we opted to do the distribution through Matt Pharr’s method or with bake3d(), the value can already be present in the file, and we can step over this stage).

One hacky approach is to read the file one sample at a time through a shader DSO, then do an illuminance loop on the position and the normal just obtained, compute the $\underline{\mathsf{L}} \cdot \mathsf{N}_{cache}$ factor, and write the whole thing back. This is the code for such a scheme:

```
surface lightbake_srf
    (
    string inputScattCache  = "";
    string outputScattCache = "";
    float doFlipNormals = 0.0;
    )
{
    if (opencache(inputScattCache) == 1) {
        float index = indexcache();
        while(index >= 0.0) {
            point  cacheP = getPcache(index);
            normal cacheN = getNcache(index);
            cacheP = transform("world", "current", cacheP);
            if (doFlipNormals == 1)
                cacheN = ntransform("world", "current", -cacheN);
            else
                cacheN = ntransform("world", "current", cacheN);
            illuminance(cacheP, cacheN, PI/2) {
                float LdotN = L.cacheN;
                if (LdotN > 0.0) {
                    accumcache(index, Cl * LdotN / length(L));
                }
            }
            index = indexcache();
        }
        writecache(outputScattCache);
    }
}
```

This is a pretty weird surface shader indeed: apart from the illuminance statement, it uses only DSO calls to access our file format. The cache is opened and kept in as a static member of the DSO at the opencache stage, so that we can look up any specific position quickly. We are using an index as a kind of a pointer to the current sample, so all information about it can be read and written correctly.

A word of caution here: the light shaders need to be modified to do their own point sampling on the shadows; otherwise, the derivatives, as far as Renderman is concerned, will come from the successive cacheP values (which can jump all over the place, depending on how they are stored). Here is another trick. Instead of assigning the lightbake shader to a random visible piece of geometry, we use an RiPoint primitive, which has no derivatives by construction. The shadow calls are thus forced to point sample the buffers, without any editing of the light code. Another side benefit is that the overall computation is much faster this way.

To obtain the final beauty BSSRDF result, in yet another DSO, we read our cache file and store the data in memory as an octree for quick lookup. Given a search radius R, we can sum up the R_d contributions from each sample found in the proximity region around $\underline{\mathrm{P}}$. This is the same R_d computation as presented in section 1-b:

```
float Rd = (sigma_tr_dr+1.0) * exp(-sigma_tr_dr) * zr/(dr^3)
         + (sigma_tr_dv+1.0) * exp(-sigma_tr_dv) * zv/(dv^3);
```

But we want to pre-multiply it by the correct precomputed radiance at each sample and its coverage area. Also, if we have the color information in the file, we can re-derive σ_{tr}, zr and zv through the alpha inversion as mentioned in paragraph 2. Note that the surface shader does not have to do any queries of the lights or any *tracing* anymore; it just needs to call the DSO.

This scatter gathering stage can still be fairly slow, as the cache might contain several thousands samples (to accurately represent the geometry and local details in the illumination and textures), so we use a level-of-detail approach. We generate some cached distribution at several densities. We search locally in the finer cache, and then, as the exponential falloffs in R_d tend towards 0, we look up in the more sparse files. This way, we can minimize the number of returned sample positions. [Jensen '02] describes a more formal data structure for this kind of optimization.

Alternatively, as recommended in Pixar's documentation, the dipoles can be summed on the point clouds themselves, through a utility like *ptfilter -ssdiffusion*. This is a highly efficient solution, as one does not incur the cost of integration at shading time, but its accuracy is as good as the density of the samples: as outlined in section 2, the alpha inversion can only be resolved on those points, thus potentially producing lower resolution texturing.

5 Extensions

5.1 Ambient occlusion, global illumination and cast shadows

Obviously the Z-buffer approach requires some sort of direct illumination, with some actual light sources. At first this makes it seem impossible to get an ambient occlusion (see last year's notes) contribution to participate in the scattering. Even though one can still do the ambient as a separate term in the shader, there is a sneaky way to merge the two together. The idea is to compute the occlusion, not through a shadeop or a ray-tracing pre-pass, but with a dome of lights. This method has recently been described on the web at: `http://www.andrew-whitehurst.net/amb_occlude.html`, so we are not going to cover it here.

Similarly, the skin position is assumed to be at the Z-buffer locations, so it does not seem easy to get correct cast shadows on our subject (we would not want to make the algorithm think that our surface is feet away from where it really is, right?). The solution to that is to generate a separate "outer" shadow buffer for each light, and scale the final result based on it.

On the other hand, there is no such limitation on the point cloud method of computing the scattering. Because we light the samples as a pre-pass, we can use any type of illumination algorithms on them, even any type of rendering engine, ahead of accumulating the dipoles in the final render.

For instance, we discovered that indirect diffuse computations have an impact on the look of the skin, and that they enhance the effects of the subsurface scattering. Ears tend to be a tiny bit too dark if not backlit, unless one first simulates at least one bounce of diffuse illumination.

5.2 Concept of blocking geometries

Up to this point the shader hasn't solved a small structural problem. It doesn't take into account the fact that inside the skin there are bones that are basically blocking some of the scattering effect. This extension is critical for small creatures and/or highly translucent materials, where one wants to give the impression of an obscuring entity inside the medium.

With the Z-buffer approaches, this is pretty trivial. We just need to generate yet another shadow buffer from the point of view of the scattering lights, a buffer of those inner blocking geometries. Then we can point sample it during the loops to get a z_{inner} value. By comparing our projected z_{map} and original sample point zcomp(P_i) with this z_{inner}, we can decide if and how much the sample participates in the illumination. This method is valid for both single and diffusion scattering.

If our BSSRDF is based on the point cloud approach, we can somehow simulate this effect by distributing samples on the inner geometries, summing the diffusion scattering they generate as seen from our skin shading point P, then subtracting this influence (or part of it) from our real surface diffusion scattering computation. This trick is certainly non-physical, but it seems to produce a pleasing effect.

6 Summary

We presented a set of techniques for implementing a production ready sub-surface scattering model in the shading language.

Acknowledgments

We would like to thank the following people who all contributed one way or another in turning the theory into a practical solution. Dan B Goldman, Andrea Maiolo and Hilmar Koch for their shading savviness, Tony Pelle for his ”pointscatter” tool and its viewer, Pixar and the Renderman team for integrating some of these methods natively, and the entire ILM R&D department for supporting this effort and letting us publish it. These notes copyright (c) 2003-2005, Industrial Light + Magic, a division of Lucasfilm. All rights reserved.

makeScatAlphaTables.C:

```
//   utility to generate a table corresponding to the distribution:
//   1 - exp(-r) * (1+r)

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

static const   int numSampleTables = 1;
static const   int maxSamples = 128;
static float     sampleX[numSampleTables][maxSamples];
static float     sampleY[numSampleTables][maxSamples];

static void randomPoint (float &u, float &v)
{
   do {
         u = drand48() - 0.5;
         v = drand48() - 0.5;
   } while (u * u + v * v > 0.25);
}

// First use the best candidate algorithm from [Mitchell 1991]:
// Spectrally Optimal Sampling for Distribution Ray Tracing
// Don P. Mitchell, Siggraph Proceedings 1991

void initSampleTables()
{
   const int q = 10;

   for (int table = 0; table < numSampleTables; table++) {

         randomPoint (sampleX[table][0], sampleY[table][0]);

         for (int i = 1; i < maxSamples; i++) {

            float dmax = -1.0;

            for (int c = 0; c < i * q; c++) {

                  float u, v;
                  randomPoint (u, v);

                  float dc = 2.0;

                  for (int j = 0; j < i; j++) {

                     float dj =
                           (sampleX[table][j] - u) * (sampleX[table][j] - u) +
                           (sampleY[table][j] - v) * (sampleY[table][j] - v);
                     if (dc > dj)
                           dc = dj;
                  }

                  if (dc > dmax) {
                     sampleX[table][i] = u;
                     sampleY[table][i] = v;
```

```
                    dmax = dc;
                }
            }
        }
    }
    return;
}

// Now take the uniform distribution and warp it into ours

inline float distFunc(float r)
{
    return(1.0-exp(-r)*(1.0+r));
}

float invDistFunc(float y, float tolerance)
{
    float test, diff;
    float x = y;
    while(1) {
        test = distFunc(x);
          diff = y - test;
          if (fabsf(diff) < tolerance) break;
          x = x + diff;
    }
    return(x);
}

void adaptSampleTables()
{
    float PI = fasin(1.0)*2.0;
    for (int i = 0; i < numSampleTables; i++) {
          for (int j = 0; j < maxSamples; j++) {
             float X = 2.0*sampleX[i][j]; // between -1 and 1
             float Y = 2.0*sampleY[i][j];
             float R = fsqrt(X*X+Y*Y);    // between 0 and 1
             float r = invDistFunc(R,.00001);
             float theta = fatan2(Y,X);   // between -PI and PI
             sampleX[i][j] = fcos(theta)*r;
             sampleY[i][j] = fsin(theta)*r;
          }
    }
}

// Finaly print the resulting tables

void printSampleTables()
{
    int i, j;

    printf ("uniform float scattdiffdist1[maxSamples] = {\n");
    for (i = 0; i < numSampleTables; i++) {
          for (j = 0; j < maxSamples; j++) {
             printf ("%f", sampleX[i][j]);
             if (j < maxSamples - 1)
                  printf (",");
          }
```

```
            if (i < numSampleTables - 1)
                printf (",\n");
    }
    printf ("\n};\n");

    printf ("uniform float scattdiffdist2[maxSamples] = {\n");
    for (i = 0; i < numSampleTables; i++) {
            for (j = 0; j < maxSamples; j++) {
                printf ("%f", sampleY[i][j]);
                if (j < maxSamples - 1)
                        printf (",");
            }
            if (i < numSampleTables - 1)
                printf (",\n");
    }
    printf ("\n};\n");
}

void main()
{
    initSampleTables();
    adaptSampleTables();
    printSampleTables();
    exit (0);
}
```