ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Photo-Realistic Real-time Face Rendering

Semester project
LGG Laboratory, EPFL

Daniel Chappuis

Supervisors : Dr. Thibaut Weise, Sofien Bouaziz
Professor : Dr. Mark Pauly
January 7, 2011

Contents

Chapter 1

Introduction

Skin rendering is a really important topic in Computer Graphics. A lot of virtual simulations or video games contain virtual humans. In order to obtain a realistic realism of their faces, we need to take care of skin rendering particularly because we are very sensitive to the appearance of skin. Nowadays, we can use modern 3D scanning technology to obtain very detailed meshes and textures for the face. But the difficulty with skin rendering is that we need to model subsurface scattering effects. Subsurface scattering is the fact that light goes under the skin surface, then scatters, gets partially absorbed, and at the end exits the skin somewhere else. It is very important to correctly handle this effect in order to render photo-realistic faces. There already exists offline techniques that simulate skin subsurface scattering and give very realistic looking skin. But this has a cost, it can take second or minutes to render.

With their work, Eugene d'Eon, David Luebke, and Eric Enderton [7] introduced a technique to approximate subsurface scattering of skin in real-time on the GPU. They have obtained very realistic results.

The goal of this project is to implement their algorithm in order to simulate subsurface scattering of skin. I will also compare the results with different parameters of the algorithm. I have also implemented diffuse environment lighting with occlusions.

1.1 Previous work

The first really important work about rendering translucent material was the work of Jensen et al. [16] in 2001. Then, in [10], Gosselin et al. approximated subsurface scattering to render skin. But this technique used an approximation of subsurface scattering but not based on single or multi-layered translucent material. For instance, they used a Gaussian smoothing but this is not physically-based and therefore the result was not very realistic. Then in 2005, with their work [3], Donner and Jensen show that realistic rendering of skin requires to correctly model multi-layered subsurface scattering. In their work, they used a three layer skin model. They have obtained very realistic results using a Monte Carlo renderer but the computation took 5 minutes. Their result is shown in figure 1.1.

Then, in 2007 with their work [7], Eugene d'Eon, David Luebke and Erir Enderton used an approximation of the scattering diffusion profiles of Donner and Jensen with a weighted sum of six Gaussian functions. Because of the nice properties of Gaussian filters, they have been able to render very realistic human skin in real-time on the GPU. You can see the result in figure 1.2.

Environment maps are usually used in Computer Graphcis to render a reflective surface in order that it reflects the surrounding environment. But this is all about specular reflection and it is much more difficult to use environment map for diffuse reflection because at each point on the surface, we need to compute diffuse reflection for every possible directions from the environment map. Therefore, we need to compute an integral for each point on the surface which we cannot do

Figure 1.1: *Skin rendering obtained by Donner and Jensen*

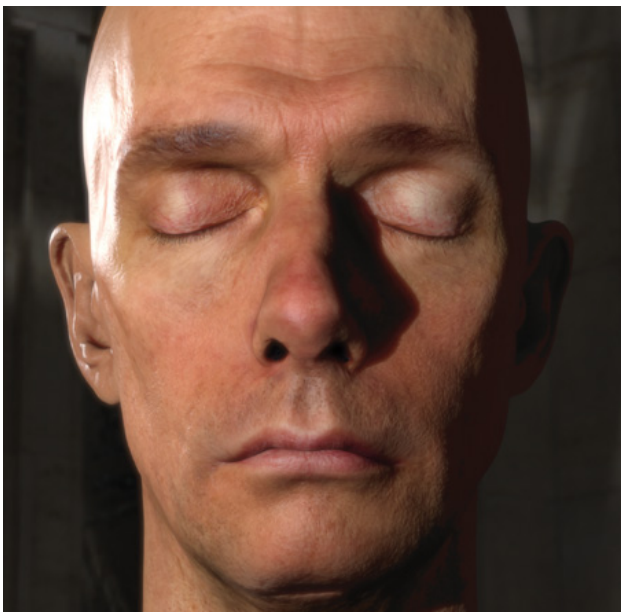

Figure 1.2: *Real-time skin rendering obtained by d'Eon, Luebke and Enderton*

efficiently in real-time. But in 2001, with their work [17], Ravi Ramamoorthi and Pat Hanrahan introduced a way to approximate diffuse lighting from an environment map with spherical harmonics. After a precomputation of the environment map, it is possible to compute diffuse lighting from the environment map in real-time.

Chapter 2

Real-time Skin Rendering

This chapter is about the main part of the project which is the algorithm for rendering skin subsurface scattering.

2.1 Theory of subsurface scattering

As we have seen, skin rendering is quite difficult because we need to take care of subsurface scattering. If we don't, the skin will look very hard and dry. To see this, compare the figure 2.1 where subsurface scattering is not used with the figure 2.2 which is a result of the skin rendering I have obtained with the subsurface scattering algorithm.

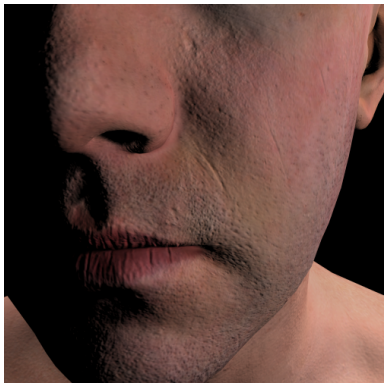

Figure 2.1: *Skin rendering without subsurface scattering*

As you can see, taking subsurface scattering into account is very important in order to have a realistic skin appearance. Subsurface scattering is the process where light goes beneath the surface of the skin, scatters and gets partially absorbed, and then exits the skin surface somewhere else. This results in a translucent appearance of the skin.

Now, we will try to understand how light interacts with skin. I will first discuss the reflectance of light at the surface of skin and then the subsurface reflectance.

2.1.1 Skin Surface Reflectance

The figure 2.3 shows a model of the skin used by Donner and Jensen in [8]. The skin is mainly made of three parts. The first one is the surface which is quite thin and also oily. Just under the surface, we find the epidermis layer and the dermis layer. The epidermis and the dermis are mainly responsible for the subsurface scattering. For the moment, we will focus on the reflection at the

Figure 2.2: *Skin with subsurface scattering*

surface of the skin.

Approximately 6 percent of the incident light reflects directly at the surface of skin without being colored. This is the result from a Fresnel effect with the topmost layer of the skin which is quite oily. This reflection is not a perfect mirror reflection because the surface of skin is also rough (as you can see on the right of figure 2.3). Therefore, an incident ray is not reflected in only one single direction. The reflectance at the surface can be modeled using a specular bidirectional reflectance distribution function (BRDF) quite often used in Computer Graphics. But we cannot use a simple model like Blinn-Phong because is it not physically-based and does not accurately approximate the specular reflection of skin. For this reason, we will use a physically-based specular BRDF. This function will be explained in section 2.3.

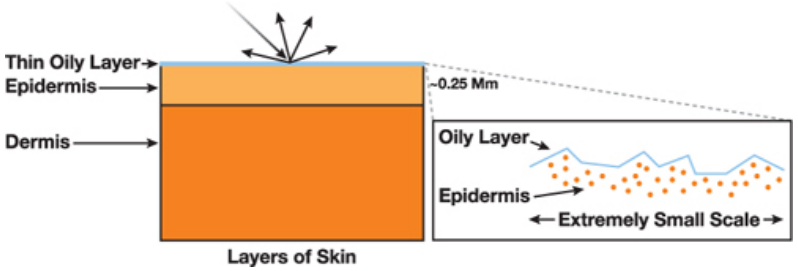

Figure 2.3: *Surface reflectance*

2.1.2 Skin Subsurface Reflectance

The light that is not directly reflected at the skin surface will enter the subsurface layers. The light enters those layers and is partially absorbed which will give its color and is also scattered quite often, returning and exiting the surface in a neighborhood of the initial entry point. This process is illustrated in the figure 2.4. Note that it is possible that light goes completely through thin regions like ears.

It is important to understand that light is not absorbed and scattered the same way in the different layers (epidermis and dermis) of the skin (see figure 2.4). The model that we are using here is composed of three layers (oily surface layer, epidermis and dermis). Actually, real skin is much more complex. Indeed, as explained in [14], the epidermis layer contains five different layers by itself. But Donner and Jensen in [3] have shown that a single-layer model is not sufficient for skin rendering and using a three-layer model seems to be a good choice. Note that the algorithm for subsurface scattering that we will use, doesn't handle single scattering effects. Single scattering

is the fact that each ray scatters beneath the surface but only once. This is a quite correct approximation for skin but if we would render marble or smoke for instance, it wouldn't be a good choice because for those materials, single scattering is really important.

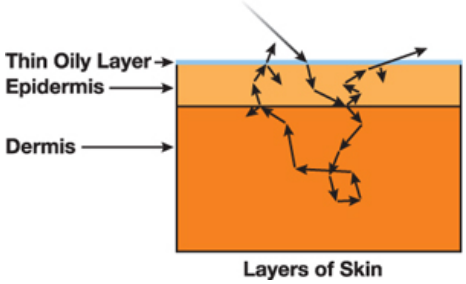

Figure 2.4: *Subsurface scattering in epidermis and dermis*

2.1.3 Diffusion Profiles

To better understand how to simulate subsurface scattering, we need to indroduce the notion of *diffusion profile*. A diffusion profile is an approximation of how the light scatters under the surface of a highly scattering translucent material. Consider a flat surface in a dark room. Now imagine that we illuminate that surface with a very thin white laser beam. The result is that we will see a glow around the center point where the laser touches the surface. This is because some light is going beneath the surface and returning nearby. This is illustrated on the left of figure 2.5. A *diffusion profile* is exactly the mathematical representation of this experiment. It's a function $R(r)$ (which depends on the distance r from the glow center) and that tells us how much light emerges as a function of the angle and distance from the center of the incident light ray. For uniform materials, the shape of the diffusion profile is the same in each direction. Note that we can have a different diffusion profile for each color channel (the diffusion profile depends on the wavelength of the incident ray). The image on the right of figure 2.5 illustrates the curves of three different diffusion profiles (one for each color). For instance in this example, we can see that the red scatters much more farther than the blue or the green.

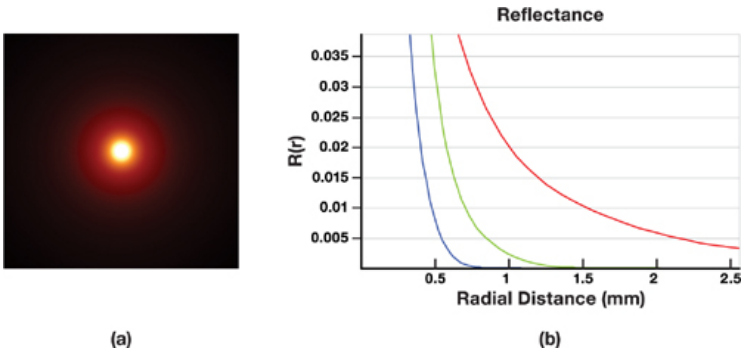

Figure 2.5: *Diffusion profile*

If we know the exact diffusion profiles of scattering of light in the light, we could simulate scattering in the following way. For each point on the skin surface, we collect all the incoming light and spread it around the surface point according to the shape of the diffusion profilles. By doing this for each surface point, we will obtain a translucent aspect of the skin and is exactly what we want. Notice that we don't really need to keep the direction information of incident rays for simulating surface scattering because in skin the light is diffused so quickly that only the amount of incident light is important.

In 2005, Donner and Jensen presented their three-layer skin model and created diffusion profiles predicted by their model.

2.1.4 Approximating Diffusion Profiles

A diffusion profile can be a quite complicated function but Eugene d'Eon, David Luebke, and Eric Enderton have shown in [7] that it can be approximated by a weighted sum of Gaussian functions. Consider $G(v, r)$ to be a Gaussian function of variance v :

$$G(v, r) = \frac{1}{2\pi v} e^{\frac{-r^2}{2v}}$$

The constant $\frac{1}{2\pi v}$ is chosen such that $G(v, r)$ doesn't darken or brighten the input image when used for a radial 2D blur (it has unit impulse response). Therefore, the diffusion profile $R(r)$ can be approximated with :

$$R(r) \approx \sum_{i=1}^{k} w_i G(v_i, r)$$

For instance, the figure 2.6 shows the diffusion profile of the figure 2.5 approximated by a sum of two or four Gaussian functions. But why choosing a sum of Gaussian functions ? Obviously, it is because of the very nice properties of Gaussian functions. Gaussian kernel is separable and radially symmetric and moreover convolving Gaussian functions with each other produces new Gaussian functions. Those properties are very useful mainly for efficiency reasons.

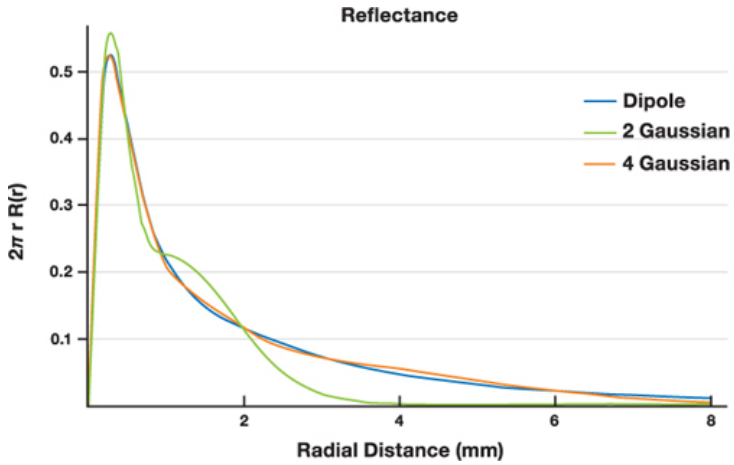

Figure 2.6: *Diffusion profile (dipole) approximated by a sum of Gaussian functions*

In [7], they have found that six Gaussian functions were needed to correctly approximate the three-layer skin model given in [3]. The figure 2.7 shows the parameters of those six Gaussian functions with the corresponding weights and the figure 2.8 plots the corresponding approximation of the diffusion profiles.

Note that the Gaussian weights of each profile sum up to 1. Therefore by normalizing these profiles to have a white diffuse color, we make sure that the result after scattering will be white on average. Then we can use an albedo texture color map to define the color of skin.

2.2 Skin Rendering Algorithm

Now that we have seen what is subsurface scattering and how it can be approximated, I will present the implementation of the skin rendering algorithm used in [7].

	Variance (mm^2)	Blur Weights Red	Green	Blue
	0.0064	0.233	0.455	0.649
	0.0484	0.100	0.336	0.344
	0.187	0.118	0.198	0
	0.567	0.113	0.007	0.007
	1.99	0.358	0.004	0
	7.41	0.078	0	0

Figure 2.7: *Six Gaussian functions to approximate a three-layer skin model*

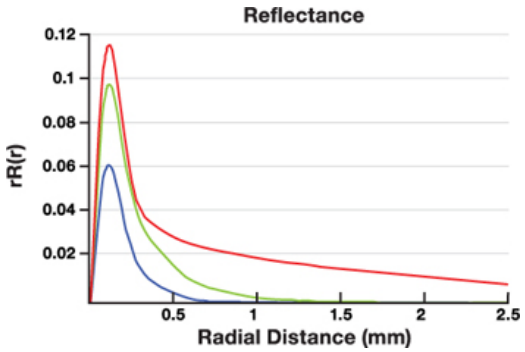

Figure 2.8: *Diffusion profiles of the sum of six Gaussian functions*

2.2.1 Texture-Space Diffusion

In 2003, Borshukov and Lewis introduced the *texture-space diffusion* technique in [9] for *The Matrix* movie. The goal of this technique is to simulate subsurface scattering efficiently. The idea is first to unwrap the 3D mesh of the head using texture coordinates as render coordinates in a 2D texture. Then, the unwrapped 2D texture of the mesh is blurred using a simple convolution that can be done very efficiently. Any kind of diffusion profiles can be used for the convolution. In 2004, Green shows in [12] that texture-space diffusion can be implemented in real-time using GPUs.

In [7], they also used *texture-space diffusion* to implement the six Gaussian convolutions. They have also incorporated transmission through thin regions like ears and have used stretch correction to obtain a more precise texture-space diffusion.

2.2.2 Overview of the algorithm

Here is the basic algorithm of the skin rendering. The following pseudo-code is executed every frame :

1. Render a shadow map for each light source.
2. Render the stretch correction map.
3. Render the irradiance into an off-screen texture (in texture-space).
4. For each one of the six Gaussian kernel in the diffusion profile approximation: we first perform a separable blur pass in U direction of the off-screen irradiance texture and render the result in a temporary buffer and then we perform a separable blur pass in V direction of the texture

in the temporary buffer and keep the resulting texture. After this step, we have six different blurred versions of the off-screen irradiance texture.

5. Render the mesh in 3D. We compute here the weighted sum of the six irradiance textures to approximate for subsurface scattering at each pixel. We also add the specular reflectance at this point.

2.2.3 Rendering Irradiance in Texture-Space

The first step of the algorithm is to render a shadow map for each light source of the scene. This process is described in section 2.4. Then we need to compute the off-screen irradiance texture (in texture-space). Therefore, we render (in an off-screen texture) the incident irradiance at each pixel of the mesh. Notice that we render the irradiance in texture-space by unwrapping the mesh as described in section 2.2.1 using the mesh texture coordinates as render coordinates. At each pixel, we compute the sum of the incident diffuse irradiance for each light source and store it in the irradiance texture. Note that for each light source, we need to take care of its shadow by looking up in the corresponding shadow map if the current pixel is on shadow or not (see section 2.4 for more details about shadow mapping).

The code for the irradiance texture rendering is given in the `irradianceShader` vertex and fragment shaders. This shader also implement energy conservation which will be descibed in section 2.6. The figure 2.9 illustrates the off-screen irradiance texture computed for our mesh.

Figure 2.9: *Off-screen irradiance texture (computed in texture-space)*

2.2.4 Blurring the Irradiance Texture

Now that we have computed the irradiance texture, we know at each point of the surface of the mesh the amount of incoming diffuse light. Then, the next step is to simulate subsurface scattering. As we have seen, subsurface scattering is represented by the diffusion profiles. We have also seen that the diffusion profiles of skin can be approximated with a weighted sum of six Gaussian functions with the parameters and weights given in figure 2.7. Thus, what we have to do is to convolve the irradiance with the weighted sum of Gaussian functions. Notice that we will use a very nice property of convolution: the convolution of an image I by a kernel that is a weighted sum of functions $G(v_i, r)$ is the same as a weighted sum of images I, each of which is the original image convolved by each of the the functions :

$$I * \left(\sum_{i=1}^{k} w_i G(v_i, r) \right) = \sum_{i=1}^{k} w_i I * G(v_i, r)$$

Therefore, what we have to do is to convolve the irradiance texture I independently with each of those six Gaussian kernels $G(v_i, r)$ and at the end compute the weighted sum of the resulting textures to simulate subsurface scattering according to the skin diffusion profiles. To correctly match the diffusion profiles of skin we have to use the correct weights given in figure 2.7 when computing the weighted sum of convolded irradiance textures. Also note that the smallest Gaussian is so narrow that using it to convolve the irradiance texture will not make any difference. Therefore we use the initial irradiance texture as the convoled version with the smallest Gaussian kernel. Thus, we only need to compute five other convolved irradiance textures. The figure 2.10 shows the irradiance texture of figure 2.9 convolved with the largest Gaussian kernel.

Figure 2.10: *Irradiance texture convolved with the largest Gaussian kernel*

A second nice property of Gaussian kernels is that they are separable. Therefore when computing the convolution of an image with a Gaussian kernel, we don't have to compute a 2D convolution which is quite expensive if the kernel and the image are large. We only need to compute a 1D convolution in the U direction and store the result into a temporary image and then compute another 1D convolution in the V direction of that temporary image. This reduces a lot the number of operations needed for convolving a 2D image.

The third property of Gaussians that we will use is that the convolution of two Gaussian functions is also a Gaussian. Therefore, we can produce each convolved irradiance texture by convolving the result of the previous one, allowing us to convolve the irradiance texture by wider and wider Gaussian kernels without increasing the number of taps at each step.

The result of convolving two Gaussian functions $G(v_1, r)$ and $G(v_2, r)$ with variances v_1 and v_2 is the following :

$$G(v_1, r) * G(v_2, r) = \int_0^\infty \int_0^\infty G\left(v_1, \sqrt{x'^2 + y'^2}\right) G\left(v_2, \sqrt{(x - x')^2 + (y - y')^2}\right) dx'dy' = G(v_1 + v_2, r)$$

with $r = \sqrt{x^2 + y^2}$. Therefore, if the previous convolved irradiance texture contains the convolved version $I_1 = I * G(v_1, r)$ (note that I is the irradiance texture) and we want to compute

$I_2 = I * G(v_2, r)$, we only need to convolve I_1 with $G(v_2 - v_1, r)$.

We use a separable seven-tap convolution in each U and V direction. The blurring is done in the `blurShader` vertex and fragment shaders. The same shader is used for the blurring in both U and V direction and take as a parameter the blurring direction. We begin to convolve in the U direction and store the result in a temporary texture and then convolve in the V direction. The seven Gaussian tap weights are the following :

$$\{0.006, 0.061, 0.242, 0.383, 0.242, 0.061, 0.006\}$$

Those weights represent a Gaussian kernel with a standard deviation of 1, considering that the coordinates of the taps are :

$$\{-3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0\}$$

We use this seven-tap kernel for all the convolutions. The taps are linearly scaled about the center tap to convolve by any desired Gaussian function. The spacing is scaled by the standard deviation of the current Gaussian. The listing 2.2 shows the `blurShader` fragment shader that performs the Gaussian convolution in one dimension.

Correction of texture-space distortion

Because we are convolving in texture-space, we need to take care of UV distortion. What is UV distortion ? If we have a curved surface, the distance between two locations in the texture doesn't correspond to the distance on the mesh due to UV distortion. Therefore, if we don't take care of texture-space distortion, the subsurface scattering on curved surfaces would not be accurate. A simple solution to correct this problem is to compute at every frame and for each pixel a stretch value in both U and V direction (in texture-space) and store it into a stretch-map texture. Then when convolving, we modulate the spacing of the convolution taps at each point on the surface in texture-space according to the value in the stretch-map texture.

The listing 2.1 shows how the stretch-map texture is computed. First, we compute the texture-space derivatives of world-space coordinates `derivu` and `derivv` in both U and V direction. Then we invert those values and multiply by a `stretchscale` value in order that the inverted value is in the range $[0, 1]$ to be stored in a RGB texture.

```
// Compute the world coordinate changes
vec3 derivu = dFdx(worldCoord);
vec3 derivv = dFdy(worldCoord);

// Compute to pixel changes
float stretchU = 1.0 / length(derivu);
float stretchV = 1.0 / length(derivv);

// convert stretch to [0,1] range
stretchU = stretchU * stretchscale;
stretchV = stretchV * stretchscale;

// Output the stretch in U and V directions
gl_FragColor = vec4(stretchU, stretchV, 0, 1);
```

Listing 2.1: Code to compute the stretch-map

Then in the `blurShader` fragment shader, we need to lookup the corresponding stretch value in the stretch-map texture to correctly scale the Gaussian taps. The listing 2.2 shows the code for the `blurShader` fragment shader that compute the convolution in 1D using the stretch-map texture to correct for UV distortion.

```
// Get the stretch value of the texture from the stretch map
vec2 stretch = texture2D(stretchMap, texCoord).xy;
vec2 scaleConv = 1.0 / imageDimension;
```

```
// Compute the step distance
vec2 step = scaleConv * blurDirection * stretch * gaussianStd / stretchScale;

// Gaussian weights of the 7 taps
float weights[7];
weights[0] = 0.006; weights[1] = 0.061; weights[2] = 0.242;
weights[3] = 0.383; weights[4] = 0.242; weights[5] = 0.061;
weights[6] = 0.006;

// Compute the coordinate of the first tap
vec2 coords = texCoord - step * 3.0;
vec4 sum = vec4(0.0);

// Tap 0
vec4 tap0 = texture2D(inputTexture, coords);
sum += tap0 * weights[0];
coords += step;

// Tap 1
vec4 tap1 = texture2D( inputTexture, coords);
sum += tap1 * weights[1];
coords += step;

// Tap 2
vec4 tap2 = texture2D(inputTexture, coords);
sum += tap2 * weights[2];
coords += step;

// Tap 3
vec4 tap3 = texture2D(inputTexture, coords);
sum += tap3 * weights[3];
coords += step;

// Tap 4
vec4 tap4 = texture2D(inputTexture, coords);
sum += tap4 * weights[4];
coords += step;

// Tap 5
vec4 tap5 = texture2D(inputTexture, coords);
sum += tap5 * weights[5];
coords += step;

// Tap 6
vec4 tap6 = texture2D(inputTexture, coords);
sum += tap6 * weights[6];

// Store the sum in the texture
gl_FragColor = sum;
```

Listing 2.2: Code to convolve an irradiance texture in one direction

Pre-Scatter or Post-Scatter Diffuse Texturing

As we have said before, the color of the skin comes from a diffuse albedo texture. This texture has been created by several photographs of a human subject under diffuse illumination. The diffuse albedo color has to be multiply by the irradiance from the lights to obtain the correct skin color. But now we have a choice to make. One possibility is to multiply by the diffuse albedo color before the subsurface scattering convolution when we compute the irradiance texture. This is called *Pre-Scatter texturing*. According to [7], a possible drawback of this technique is that it may lose too much of the high-frequency details. The figure 2.11 shows an image of our human head with only pre-scatter texturing. As we can observe, the result seems a little bit too smooth.

The other possibility is to multiply by the diffuse albedo color in the final pass after the subsurface scattering convolutions and not when computing the irradiance texture. This is called

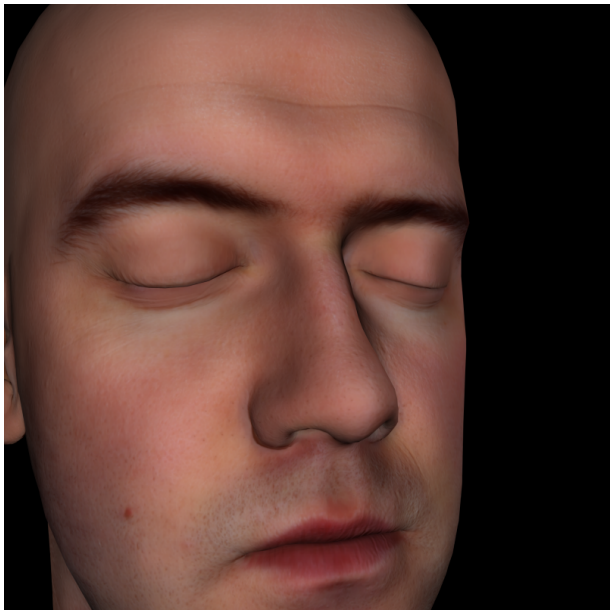

Figure 2.11: *Subsurface scattering with only pre-scatter texturing (without specular reflectance)*

Post-Scatter texturing. With this technique, the subsurface scattering convolutions is done without any colors. Now, high-frequency details are kept because their are not blurred by the subsurface scattering convolutions. Again, according to [7], the problem with this method is that there is no color bleeding of the skin tones. The figure 2.12 shows an image of our human head with only post-scatter texturing. We can observe that the high-frequency details are kept.

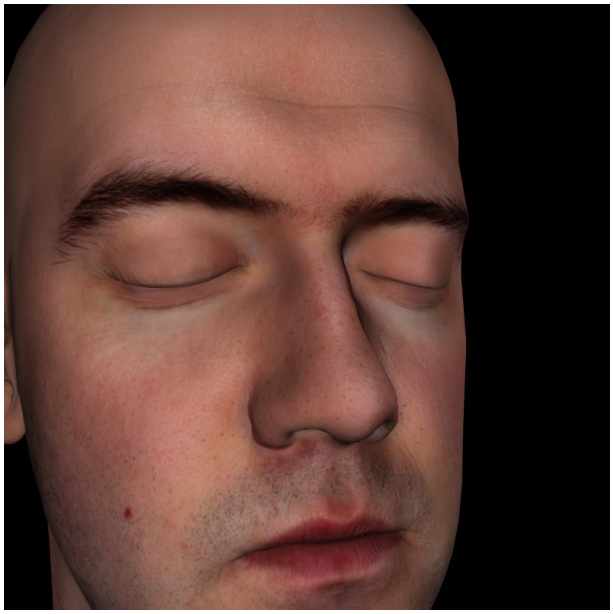

Figure 2.12: *Subsurface scattering with only post-scatter texturing (without specular reflectance)*

A good solution is to combine pre-scatter and post-scatter texturing. With this technique, a part of the diffuse albedo color is applied in the irradiance texture computation (pre-scatter texturing) and the remaining part is applied at the end after the subsurface scattering computation (post-

scatter texturing). To implement this, we can multiply the diffuse irradiance `diffuseIrradiance` by a portion of the diffuse albedo `diffuseAlbedo` color in the irradiance texture computation as follows :

$$\texttt{diffuseIrradiance} *= \texttt{pow(diffuseAlbedo, mixRatio)}$$

where `mixRatio` is a value in the range $[0, 1]$ representing the amount of pre-scatter or post-scatter texturing. The previous multiplication is implemented in the `irradianceShader` fragment shader. Then, we also have to apply the remaining part of the diffuse albedo color at the end in the final pass (after the subsurface scattering computation). We implement this as follows :

$$\texttt{finalDiffuseColor} *= \texttt{pow(diffuseAlbedo, 1.0 - mixRatio)}$$

This computation is implemented in the `finalSkinShader` fragment shader. For instance, if `mixRatio` $= 1.0$, we only apply the diffuse albedo color in the irradiance texture computation and therefore this corresponds to pre-scatter texturing only. According to [7], a good value for `mixRatio` is 0.5. All the images in this document (except figures 2.11 and 2.12) have been rendered with a value of 0.5 for `mixRatio`.

2.3 Specular lighting

As we have already said, the Phong model is used a lot in Computer Graphics for simulating specular reflectance. But this model is not physically based and therefore doesn't give very realistic results. For instance, the Phong model doesn't capture increased specularity at grazing angles. It can also output more energy than it receives. Therefore we need to use a physically-based Bidirectional Reflectance Distribution Function (BRDF) for the specular reflectance. We will use the Kelemen/Szirmay-Kalos model and see how to implement it.

In general, a BRDF function $f_{BRDF}(\omega_i, \omega_o)$, where ω_i and ω_o are respectively the input and output directions of the light, is defined as follows at a given surface point :

$$f_{BRDF}(\omega_i, \omega_o) = \frac{dL_r(\omega_o)}{dE_i(\omega_i)} = \frac{dL_r(\omega_o)}{L_i(\omega_i)\cos\theta_i d\omega_i} \tag{2.1}$$

where L is the output radiance, E is the irradiance and θ_i is the angle between the normal at the given surface point and ω_i. Therefore, we have :

$$dL_r(\omega_o) = f_{BRDF}(\omega_i, \omega_o) \cdot L_i(\omega_i)\cos\theta_i d\omega_i \tag{2.2}$$

Usually, a specular BRDF is a product of several terms like a constant `rho_s` which is the specular intensity, a surface normal vector N, a viewing vector V, a light direction vector L, an index of refraction `eta` (used for the Fresnel term) and a roughness parameter m. Then for each light source, the specular reflectance is computed as follows :

```
specularReflect  += lightColor * lightShadow * rho_s
                  * specularBRDF(N, V, L, eta, m) * dot(N, L)
```

Note that the term `dot(N, L)` comes from the definition of the BRDF (see the cosine term in equation 2.2).

The Fresnel effect is the fact that the specular reflectance increases at grazing angles. It is important to take this effect into account if we want to obtain a physically plausible result. As in [6], we use the Schlick's Fresnel approximation (see [4]) that seems to works quite well for skin. The listing 2.3 shows the code to compute the Fresnel term. This code can be found in the `finalSkinShader` fragment shader. Note that H is the half-viewing vector, V is the viewing vector and $F0$ is the reflectance at normal incidence. As explained in [6], the value 0.028 should be used for skin.

```
// Compute the Fresnel reflectance for specular lighting
float fresnelReflectance(vec3 H, vec3 V, float F0) {
    float base = 1.0 - dot(V, H);
    float exponential = pow(base, 5.0);
    return exponential + F0 * (1.0 - exponential);
}
```

Listing 2.3: Code to compute the Fresnel term

A specularly reflecting surface doesn't have a specular highlight as the perfectly sharp reflected image of a light source. Usually, the objects show a blurred specular highlights. This can be explained by the existence of microfacets [1]. The idea is to assume that a surface that is not perfectly smooth is made of several very small facets, each of which is a perfect specular reflector. These microfacets have normals that are distributed about the normal of the approximating smooth surface. The degree to which microfacet normals differ from the smooth surface normal is determined by the roughness of the surface. There exists several ways to predict the distribution of the microfacets. For instance, in the Phong model, the specular highlight k_{spec} is computed as follows :

$$k_{spec} = \cos^n(R, V)$$

where R is the reflection vector of the light vector L at a given point on the surface and V is the viewing direction. But a more physically based model is the *Beckmann distribution*. With this model, the specular intensity k_{spec} is computed as follows :

$$k_{spec} = \frac{\exp(\frac{-\tan^2(\alpha)}{m^2})}{\pi m^2 \cos^4(\alpha)}, \ \alpha = \arccos(N \cdot H)$$

where m is the roughness of the material and H is the half-vector bewteen L and V.

Heidrich and Seidel described in 1999 a precomputation strategy to efficiently evaluate a BRDF model (see [18]). The idea was to factor the BRDF into several precomputed 2D textures. In [6], the used a similar method but precomputed a unique texture corresponding to the Beckmann distribution function that we have just seen and used the Schlick Fresnel approximation that we have introduced above. Their precomputation of the Beckmann distribution is done by rendering a texture that will be accessed by the dot product of N and H and the roughness m. The listing 2.4 shows how the Beckmann texture is precomputed. This code can be found in the `beckmannShader` fragment shader.

```
void main() {
    // Compute and map the PH value in the range [0, 1] to be stored in the
        texture
    float PH = 0.5 * pow(PHBeckmann(texCoord.x, texCoord.y), 0.1);
    gl_FragColor = vec4(PH, PH, PH, 1.0);
}

// Compute the Beckmann PH value
float PHBeckmann(float ndoth, float m) {
    float alpha = acos(ndoth);
    float ta = tan(alpha);
    float val = 1.0/(m*m*pow(ndoth,4.0))*exp(-(ta*ta)/(m*m));
    return val;
}
```

Listing 2.4: Code to precompute the Beckmann texture

The listing 2.5 shows the code to compute the specular BRDF using the Beckmann texture. This code can be found in the `finalSkinShader` fragment shader.

```
// Kelemen/Szirmay-Kalos Specular BRDF
float KS_Skin_Specular(vec3 N, vec3 L, vec3 V, float m, float rho_s) {
    float specular = 0.0;
```

```
        float ndotl = dot(N, L);
        if (ndotl > 0.0) {
            vec3 h = L + V;
            vec3 H = normalize(h);
            float ndoth = dot(N, H);
            float PH = pow(2.0 * texture2D(beckmann_texture, vec2(ndoth, m)).r,
                10.0);
            float F = fresnelReflectance( H, V, 0.028);
            float frSpec = max( PH * F / dot( h, h ), 0 );
            specular = ndotl * rho_s * frSpec;
        }
        return specular;
    }
```

Listing 2.5: Code to compute the specular BRDF

The figures 2.13, 2.14 and 2.15 show the specular reflectance for different values of the roughness m. In [2], they say that a roughness of $m = 0.3$ is a good average value for the face. It is also possible to paint the roughness value in a map in order that the value can change over the face. All the other images in the document use a value of 0.3 for the roughness.

Figure 2.13: *Rendering with roughness* $m = 0.1$. *Right image is only specular*

2.4 Shadows

Obviously, simulating shadows is very important for realistic rendering. As we have already said before, we use *shadow mapping* to render shadows. Shadow mapping is a classical but widely-used technique. A surface point of our object is not in shadow if there is no other object between the point and a certain light source. The idea of shadow mapping is to place the camera at the light source position and render the scene from this point but instead of rendering the pixel color of the objects of the scene, we render the distance d_{shadow} of each pixel to the light source. This rendering is stored into a texture which is called a *shadow map*. Therefore, the shadow map contains the distances to the light source of all objects visible from that light source. Then when rendering a given pixel p of our object from the camera view, we compute the distance d_{pixel} from that pixel p to the light source. Then we lookup in the shadow map the distance d_{shadow} in the shadow map corresponding to the direction of the pixel p from the light source. Then we compare both distances and if $d_{pixel} > d_{shadow}$, it means that the pixel p is in shadow and cannot receive light from that light source. The figure 2.16 shows the difference when rendering without and with shadows.

Figure 2.14: *Rendering with roughness $m = 0.3$. Right image is only specular*

Figure 2.15: *Rendering with roughness $m = 0.5$. Right image is only specular*

We need to generate a shadow map for each light source of the scene. This is done at the beginning of each frame. Then when computing the irradiance texture, we use the shadow maps to compute a shadow factor for each pixel and light source. The shadow factor for a given pixel and a given light source tells us if the light source contributes to the irradiance of that pixel.

Shadow mapping is not so easy because it can generate serious artifacts. Firstly, we need to take care of self-shadowing. Self-shadowing occurs because the comparison between d_{pixel} and d_{shadow} is a floating number comparison and therefore because of the floating number precision, it cannot always get the right answer if two values are very close. This can cause Moiré artifacts as shown in the left image of figure 2.17. A possible solution to this problem is to add a small bias for instance using the `glPolygonOffset()` function of OpenGL.

A second problem with shadow mapping is the generation of hard shadows and aliased shadow edges (as you can see in the right image of figure 2.17) To correct this problem we implement a

Figure 2.16: *Rendering without shadows (left) and with shadows (right)*

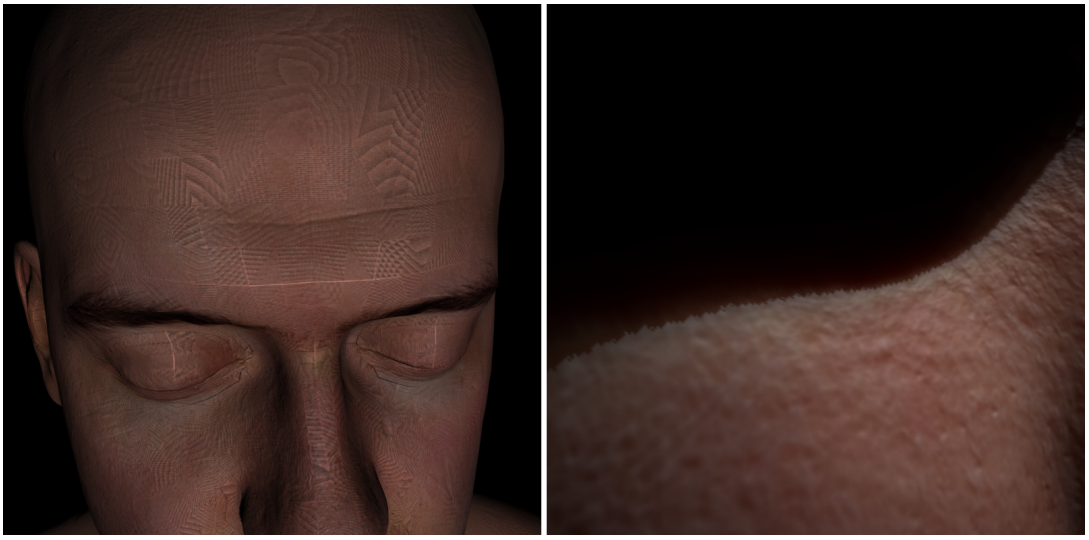

Figure 2.17: *Shadow mapping artifacts (Moiré patterns) due to self-shadowing (left) and aliased shadow edges (right)*

technique called *Percentage Closer Filtering* (PCF). The idea is to smooth the aliased edges by simply sampling several surrounding texels in the shadow map along with the center texel and take the average of all the shadow factors. I use a 4x4 PCF kernel which means I sample the 16 surrounding texels of the texel in the shadow map I want to compute the shadow factor. The listing 2.6 shows the code that implement the PCF technique. Notice that we need to compute a shadow factor in the `irradianceShader` fragment shader to take care of shadows for diffuse lighting but also in the `finalSkinShader` fragment shader to obtain shadows for the specular reflectance term.

```
// Lookup in a shadow map at a given offset
float lookupShadowMap(vec4 shadowCoord,vec2 offSet,sampler2DShadow shadowMap)
    {
    return shadow2DProj(shadowMap, shadowCoord + vec4(offSet.x * xPixelOffset
        * shadowCoord.w, offSet.y * yPixelOffset * shadowCoord.w, 0.0, 0.0) ).
        x;
```

```
}

// Compute a 4x4 PCF average from the shadow map to obtain smooth shadows
float shadowPCF(vec4 shadowCoord, sampler2DShadow shadowMap) {
    float shadow = 0.0;
    float x,y;;

    if (shadowCoord.w > 1.0) {
        // Compute an 4x4 average from the shadow map
        for (y = -1.5 ; y <=1.5 ; y+=1.0) {
            for (x = -1.5 ; x <=1.5 ; x+=1.0) {
                shadow += lookupShadowMap(shadowCoord, vec2(x,y), shadowMap);
            }
        }
        shadow /= 16.0 ;
    }

    return shadow;
}
```

Listing 2.6: Code for the PCF shadow smoothing

2.5 Modified Translucent Shadow Map

Texture-space diffusion cannot capture light transmitting completely through thin regions such as ears because two surface locations can be very close in 3D space but quite distant in texture space. In [7], they have modified the *Translucent Shadow Map* technique introduced by Dachsbacher and Stamminger in [5] that allows an efficient estimate of diffusion through thin regions. In the original Translucent Shadow Map method, what is rendered in the shadow map is the depth z, the normal and the irradiance of each point on the surface nearest the light source (the light-facing surface). The modified version that we use here is to render the depth z and the (u,v) texture coordinate of the light-facing surface. Then at each point in shadow, we can compute the thickness through the mesh (using the depth z) and we can access the irradiance texture on the opposite side of the surface (using the texture coordinate (u,v)). This process is illustrated in the figure 2.18.

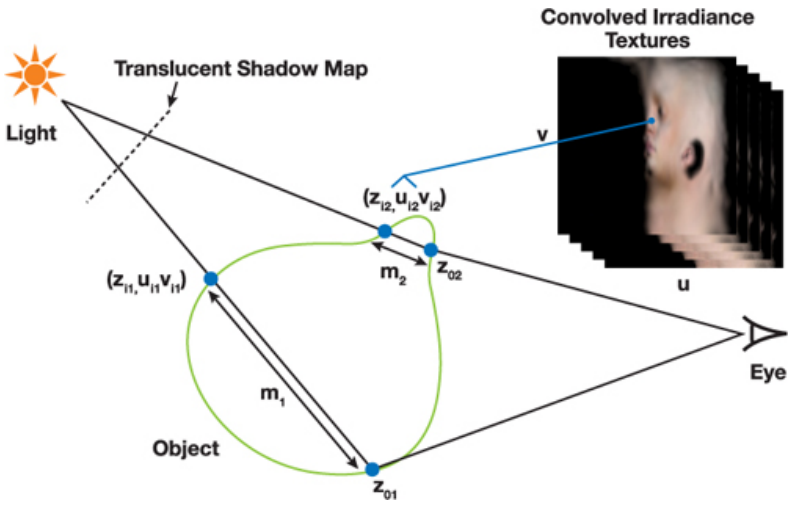

Figure 2.18: *Modified Translucent Shadow Map*

Now, we will see how to compute global scattering given that we have the modified translucent shadow map corresponding to a light source. If you look at the figure 2.19, the goal is to compute at any shadowed surface point C global subsurface scattering using the part of the convolved irradiance texture around the point A that is in the light-facing surface. For each point C, the

Translucent Shadow Map contains the distance m and the UV texture coordinate of point A. The scattered light that exits point C is the convolution of the light-facing points by the diffusion profile R, where distances from point C to each sample are computed individually. But instead, we compute this for the point B because it is easier and for small angles q, B is close to C. If q is large, the Fresnel and $N \cdot L$ factor will make the contribution to be quite small and hide the error.

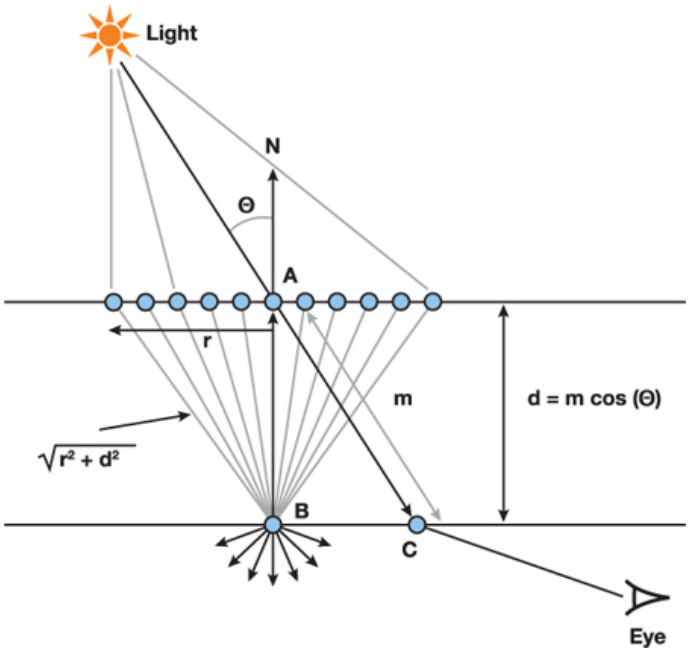

Figure 2.19: *Distance correction*

In order to compute scattering at point B, any sample at distance r away from point A on the light-facing surface has to be convolved by the following :

$$R(\sqrt{r^2+d^2}) = \sum_{i=1}^{k} w_i G\left(v_i, \sqrt{r^2+d^2}\right) = \sum_{i=1}^{k} w_i e^{\frac{-d^2}{v_i}} G(v_i, r)$$

This formula is very useful because we know that the points on the light-facing surface have already been convolved with $G(v_i, r)$ (it is what is stored in the convolved irradiance textures). Therefore the total diffuse light exiting at point C is a sum of k texture lookups (using the texture coordinates (u,v) of point A in the Translucent Shadow Map), each weighted by the weights w_i and the exponential term $e^{\frac{-d^2}{v_i}}$. Note that the depth m computed from the Translucent Shadow Map is corrected by the factor $\cos(q)$ because we use a diffusion approximation and the most direct thickness is more applicable. We also compare surface normals at points A and C and this correction is only applied if the normal are in opposite directions. We use linear interpolation to blend this correction. To avoid high-frequency artifacts in the Translucent Shadow Map, we store the depth through the surface in the alpha channel of the irradiance texture. Then when blurring the irradiance texture during convolution, the alpha channel is also blurred resulting in a blurred version of depth at the end. Then, we use the correct version of depth (the correct convolved irradiance texture) when we compute global scattering. For instance, we use the alpha channel of convolved irradiance texture $i-1$ for the depth when we compute the Gaussian i. Note that we store $e^{(-const \cdot d)}$ in the alpha channel in order that the thickness d can be stored in a 8-bit channel.

When the texture coordinates (u,v) of the point beeing rendered approaches the texture coordinates of the point in the Translucent Shadow Map, double contribution to subsurface scattering can occur. We use linear interpolation such that the Translucent Shadow Map contribution is zero

for each Gaussian term when the two texture coordinates approach each other. The listing 2.7 shows the code of the `irradianceShader` fragment shader that compute the thickness through the skin and store it into the alpha channel of the irradiance texture.

```
    // Compute the thickness through skin and store it in texture
    float distanceToLight0 = length(light0PositionWorld.xyz - worldPosition);
    vec4 TSMTap = texture2D(translucentShadowMap,shadowCoord0.xy / shadowCoord0.w);
    vec3 normalBackFace = 2.0 * texture2D(normal_texture, TSMTap.yz).xyz - vec3
        (1.0);
    float backFacingEst = clamp(-dot(normalBackFace, normal), 0.0, 1.0);
    float thicknessToLight = distanceToLight0 - TSMTap.x;

    float val = thicknessToLight;

    if (NdotL[0] > 0.0) {   // Set a large distance for surface points facing the
        light
        thicknessToLight = 50.0;
    }
    if (thicknessToLight < 2.0) { // To remove artifacts of the shadow map
        thicknessToLight = 50.0;
    }
    float correctedThickness = clamp(-NdotL[0], 0.0, 1.0) * thicknessToLight;
    float finalThickness = mix(thicknessToLight,correctedThickness,backFacingEst);
    float alpha = exp(finalThickness * -0.1); // Exponentiate thickness for storage

    // Store the irradiance and the thickness in the texture
    gl_FragColor = vec4(diffuse, alpha);
```

Listing 2.7: Code for computing the thickness through skin and storing it into the texture

The code that computes the part of the global scattering using the thickness through the skin is in the `finalSkinShader` fragment shader and you can see that code in the listing 2.11. The figure 2.20 shows the result of rendering with global scattering using Modified Translucent Shadow Map. Note that in our application there are three positional light sources but I have only applied Translucent Shadow Map for the first light source. The Translucent Shadow Map can contains some artifacts due to the borders of the 3D mesh (faces that are at a 90 degrees angle with the light direction) where the thickness through the skin is very close to zero. Such artifacts can be seen sometimes depending on the positon of the light source and the orientation of the mesh.

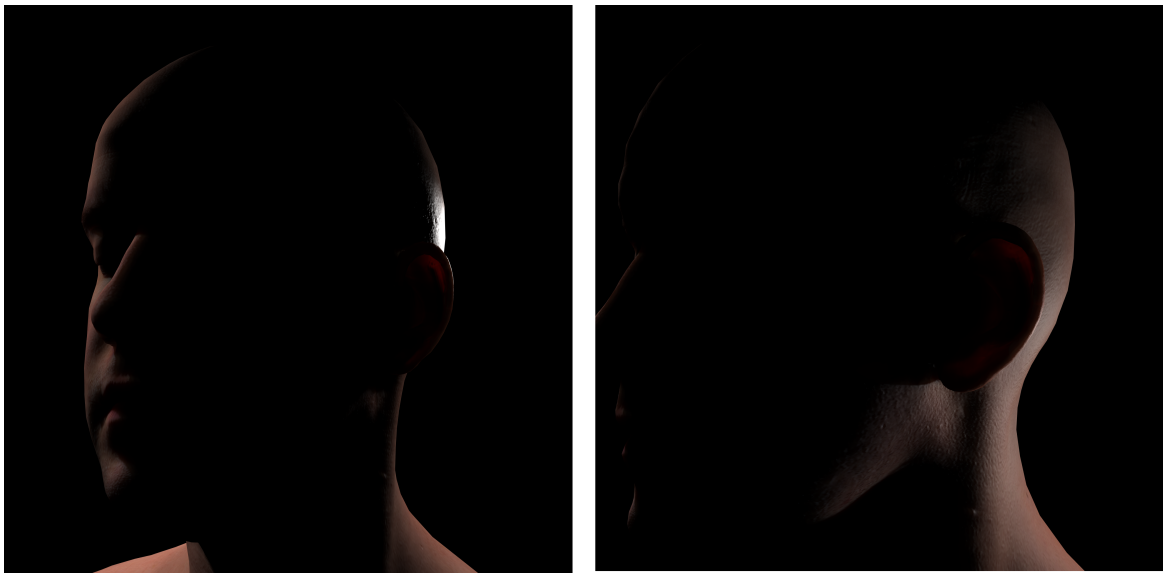

Figure 2.20: *Rendering with Modified Translucent Shadow Map*

2.6 Energy conservation

In the skin rendering algorithm, specular and diffuse lighting are treated completely independently. But this could be a problem because the total light energy leaving the surface at a given point can be larger than the incoming light energy. This would not be physically realistic. Therefore we need to take care of energy conservation in the algorithm.

The energy available for the subsurface scattering is exactly all the energy not reflected by the specular BRDF. Therefore, before computing the diffuse lighting and storing it in the irradiance texture, we need to compute the total specular reflectance at this surface point and multiply the diffuse light by the fraction of energy that remains. Note that we need to integrate the specular reflectance over all the hemisphere to take care of all the viewing directions V. Therefore, if we consider that $f_r(x, \omega_o, L)$ is the specular BRDF, L is the light vector at surface point x and ω_o is a viewing direction in the hemisphere about the normal N, then the diffuse light will be attenuated by the following factor for each light :

$$\rho_{dt}(x, L) = 1 - \int_{2\pi} f_r(x, \omega_o, L)(\omega_o \cdot N)\mathrm{d}\omega_o$$

If we use spherical coordinates, we have :

$$\rho_{dt}(x, L) = 1 - \int_0^{2\pi} \int_o^{2\pi} f_r(x, \omega_o, L)(\omega_o \cdot N) sin\theta \mathrm{d}\theta \mathrm{d}\phi$$

Note that this value will change depending on the roughness m at surface point x and on the dot product $N \cdot L$. The previous integral is precomputed for all combinations of roughness values and angles and is stored in a 2D texture to be accessed in the `irradianceShader` fragment shader based on m and $N \cdot L$. Note that this precomputation is an approximation of the integral. This precomputation is peformed in the `energyConservation` fragment shader and the code is shown in the listing 2.8. Note that the ρ_s from the specular BRDF is not taken into account in the precomputation because it will be applied later on.

```
// Integrate the specular BRDF component over the hemisphere
float costheta = texCoord.x;
float pi = 3.14159265358979324;
float m = texCoord.y;
float sum = 0.0;

int numterms = 80;
vec3 N = vec3(0.0, 0.0, 1.0);
vec3 V = vec3(0.0, sqrt(1.0 - costheta * costheta), costheta);

for (int i = 0; i < numterms; i++ ) {
    float phip = float(i) / float(numterms - 1) * 2.0 * pi;
    float localsum = 0.0f;
    float cosp = cos(phip);
    float sinp = sin(phip);

    for (int j = 0; j < numterms; j++) {
        float thetap = float(j) / float(numterms - 1) * pi / 2.0;
        float sint = sin(thetap);
        float cost = cos(thetap);
        vec3 L = vec3(sinp * sint, cosp * sint, cost);
        localsum += KS_Skin_Specular(N, L, V, m, 0.0277778) * sint;
    }

    sum += localsum * (pi / 2.0) / float(numterms);
}

float result = sum * (2.0 * pi) / (float(numterms));
gl_FragColor = vec4(result, result, result, 1.0);
```

Listing 2.8: Code to precompute the energy conservation texture

Now that we have a texture that gives us the term ρ_{dt}, we can use it in the `irradianceShader` fragment shader to attenuate the irradiance that will be used for subsurface scattering by using only the energy that remains. We have to do this for each light source. The corresponding code is in the `irradianceShader` fragment shader and is shown in the listing 2.9. Note that the `specIntensity` corresponds to the ρ_s factor from the specular BRDF.

```
float attenuation = specIntensity * texture2D(energyAttenuationTexture, vec2(
    NdotL, roughness)).r;
energyFactor = 1.0 - attenuation;
diffuse += shadow * energyFactor * max(NdotL, 0.0) * gl_LightSource[0].diffuse
    .rgb;
```

Listing 2.9: Attenuation of irradiance energy

Note that after light has scattered beneath the surface, it must pass through the same rough interface that is modeled with the specular BRDF. According to the direction from which we are looking at the surface (based on $N \cdot V$), a different quantity of diffuse light will exit. Here we use a diffusion approximation and therefore, we consider that the exiting light will be flowing equally in all directions as it reaches the surface. Therefore, we need to compute another integral term with an hemisphere of incoming directions from below the surface and a single outgoing direction V. But we don't really have to compute this new integral because the BRDF function is reciprocal (light and camera can be swapped and the reflectance is the same). Thus, we only need to reuse the same previous integral to evaluate ρ_{dt} but this time, we index it with direction V instead of L. This is applied in the `finalSkinShader` fragment shader and the corresponding code in shown in the listing 2.10.

```
float attenuation = specIntensity * texture2D(energyAttenuationTexture, vec2(
    dot(n, v), roughness)).r;
float finalScale = 1.0 - attenuation;
diffuse *= finalScale;
```

Listing 2.10: Attenuation of energy of outgoing light

I didn't really observed a change after adding energy conservation to the rendering algorithm. According to [7], the effect is very subtle. Moreover, this model is not perfect because for instance, it doesn't take interreflections into account.

2.7 Gamma correction

When we want to render realistic images, we need to take care of the non-linear behavior of a CRT screen. This non-linear behavior comes from the fact that the conversion from voltages into light intensities is not linear. The behavior of a CRT screen is represented by the bottom curve in figure 2.21. The monitor's response is an exponential curve. The exponent is usually called *gamma* (γ) and we typically have $\gamma = 2.2$ (this value can be different from device to device). If we consider that x are R,G,B values, the output intensities $y_{crt}(x)$ of a screen are such that :

$$y_{crt}(x) = x^{\gamma}$$

Usually, this behavior is called *Gamma behavior*. Note that LCD screens don't inherently have this property but they are constructed to mimic the behavior of a CRT screen.

To avoid that this gamma behavior changes the constrast of our rendered images, we need to correct for it. The basic idea is to apply the inverse of this gamma behavior to our final images just before that they are sent to the screen. Therefore, we have to apply the following function to the intensities x of our image before sending it to the screen :

$$y_{correct}(x) = x^{\frac{1}{\gamma}}$$

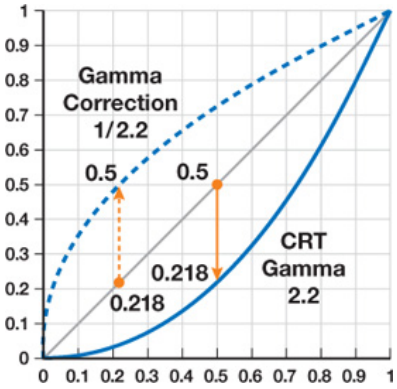

Figure 2.21: *Non-linear behavior of a CRT screen and Gamma correction*

This is called *Gamma correction*. If we apply this correction to our final image before sendind it to the screen, the resulting output will be linear and therefore, the contrasts in our image will be preserved. I have applied this gamma correction at the end of the `finalSkinShader` (see listing 2.11).

But this is not all, we have to know that a lot of image file (like JPEG for instance) are precorrected (with a gamma of 2.2) in order that a user doesn't have to take care of gamma correction by himself if he wants to look at the image on a screen. Therefore, intensity values in such an image file are not linear. Therefore, we cannot directly use those value from an input texture in our subsurface scattering algorithm because we would perform subsurface scattering in a non-linear space. This would cause some problems as described in [13]. A common problem with skin rendering, is the appearance of blue-green glow around the shadow edges. To avoid this effect, we have to apply the inverse of gamma correction to our texture images before using them. For instance, you can see in the `irradianceShader` and `finalSkinShader` fragment shaders, that we apply the following transform $y(x)$ before using the albedo texture :

$$y(x) = x^{2.2}$$

2.8 The Final Skin Shader

The listing 2.11 shows the code of the `finalSkinShader` fragment shader. The shader computes the weighted sum of Gaussian functions using the 6 convolved irradiance textures. The code contains also a part for the computation of global scattering using the thickness through the skin available in the alpha channel of the irradiance textures. The specular reflectance is computed using the specular BRDF we have seen previously. Then, after the gamma correction, the final color is written for the final rendering.

```
// Compute the diffuse lighting (sum of the weighted 6 irradiance textures)
vec3 diffuse = vec3(0.0);
if (isBlurringActive) {
    vec4 irrad1tap = texture2D(diffuse_texture0, texCoord);
    diffuse += irrad1tap.xyz * gauss1w;
    vec4 irrad2tap = texture2D(diffuse_texture1, texCoord);
    diffuse += irrad2tap.xyz * gauss2w;
    vec4 irrad3tap = texture2D(diffuse_texture2, texCoord);
    diffuse += irrad3tap.xyz * gauss3w;
    vec4 irrad4tap = texture2D(diffuse_texture3, texCoord);
    diffuse += irrad4tap.xyz * gauss4w;
    vec4 irrad5tap = texture2D(diffuse_texture4, texCoord);
    diffuse += irrad5tap.xyz * gauss5w;
    vec4 irrad6tap = texture2D(diffuse_texture5, texCoord);
    diffuse += irrad6tap.xyz * gauss6w;
```

```
    // Renormalize diffusion profile to white
    vec3 normConst = gauss1w + gauss2w + gauss3w + gauss4w + gauss5w + gauss6w;
    diffuse /= normConst;

    // If the translucent shadow mapping is active
    if (isTranslucentShadowMapActive) {
        // Compute global scatter from modified TSM
        // TSMtap = (distance to light, u, v)
        vec4 TSMtap = texture2D(translucentShadowMap, shadowCoord0.xy /
            shadowCoord0.w);
        // Four average thicknesses through the object (in mm)
        vec4 thickness_mm = 1.0 * -(1.0 / 0.2) * log(vec4(irrad1tap.w,
            irrad2tap.w, irrad3tap.w, irrad4tap.w));
        vec4 stretchTap = texture2D(stretchMap, texCoord);
        float stretchval = 0.5 * (stretchTap.x + stretchTap.y);
        vec4 a_values = vec4(0.433, 0.753, 1.412, 2.722);
        vec4 inv_a = -1.0 / (2.0 * a_values * a_values);
        vec4 fades = exp(thickness_mm * thickness_mm * inv_a);
        float textureScale = 1024 * 0.1 / stretchval;
        float blendFactor4 = clamp(textureScale * length(texCoord.xy - TSMtap.
            yz) / (a_values.y * 6.0), 0.0, 1.0);
        float blendFactor5 = clamp(textureScale * length(texCoord.xy - TSMtap.
            yz) / (a_values.z * 6.0), 0.0, 1.0);
        float blendFactor6 = clamp(textureScale * length(texCoord.xy - TSMtap.
            yz) / (a_values.w * 6.0), 0.0, 1.0);
        diffuse += gauss4w / normConst * fades.y * blendFactor4 * texture2D(
            diffuse_texture3, TSMtap.yz).xyz;
        diffuse += gauss5w / normConst * fades.z * blendFactor5 * texture2D(
            diffuse_texture4, TSMtap.yz).xyz;
        diffuse += gauss6w / normConst * fades.w * blendFactor6 * texture2D(
            diffuse_texture5, TSMtap.yz).xyz;
    }
}
else {
  diffuse += texture2D(diffuse_texture0, texCoord).xyz;
}

// Compute diffuse albedo color from the albedo texture
vec3 diffuseAlbedo = pow(texture2D(albedo_texture, texCoord).rgb, vec3(2.2));
diffuseAlbedo = pow(diffuseAlbedo, vec3(1.0-mixRatio));

// Post-Scattering according to the mix ratio value
if (isAlbedoActive) {
    diffuse *= diffuseAlbedo;
}

// Energy conservation
if (isEnergyConservationActive) {
    float attenuation = specIntensity * texture2D(energyAttenuationTexture,
        vec2(dot(n, v), roughness)).r;
    float finalScale = 1.0 - attenuation;
    diffuse *= finalScale;
}

// Compute the shadow factor for each light source
float shadow[3];
shadow[0] = 1.0;
shadow[1] = 1.0;
shadow[2] = 1.0;
if (isShadowActive) {
    shadow[0] = shadowPCF(shadowCoord0, shadowMap0);
    shadow[1] = shadowPCF(shadowCoord1, shadowMap1);
    shadow[2] = shadowPCF(shadowCoord2, shadowMap2);
}

// Compute specular reflectance
vec3 specular =  vec3(0.0);
if (isSpecularActive) {
    if (isLight0Active) specular += shadow[0] * gl_LightSource[0].specular.rgb
```

```
            * KS_Skin_Specular(n, L[0], v, roughness, specIntensity);
        if (isLight1Active) specular += shadow[1] * gl_LightSource[1].specular.rgb
            * KS_Skin_Specular(n, L[1], v, roughness, specIntensity);
        if (isLight2Active) specular += shadow[2] * gl_LightSource[2].specular.rgb
            * KS_Skin_Specular(n, L[2], v, roughness, specIntensity);
    }

    // Apply gamma correction
    vec3 finalColor = pow(diffuse + specular, vec3(1.0/gamma));

    // Render the final pixel color
    gl_FragColor = vec4(finalColor, 1.0);
```

Listing 2.11: Code for the final skin fragment shader

Chapter 3

Environment lighting

Instead of using a certain number of light sources for illuminating a scene, it can be much more realistic to use environment lighting. The idea is to use the light comming from the whole environment of the scene. Usually we can do this using environment mapping where we have a texture (a cube or a sphere) that contains the light comming from each direction around the object. Environment map are very often used for specular lighting for rendering objects that reflect the environment. The problem is that it is quite expensive to compute the diffuse irradiance at a given point of the surface of an object if the environment map is large.

Consider that we have an environment map with k texels. Each texel can be thought of beeing a single light source. Therefore, for each surface point of the object the diffuse component can be computed as follows :

$$\text{diffuse} = \text{surfaceAlbedo} \cdot \sum_{i=1,\ldots,k} \text{lightColor}_i \cdot \max(0, L_i \cdot N)$$

where L_i is the light direction of texel i and N is the surface normal. It is obvious that if the number of texels k is large computing this sum for each point of the surface of the object is really expensive.

More generally, this is the same as computing the irradiance E with the integral over a upper hemisphere $\Omega(N)$ of light directions ω as follows :

$$E(N) = \int_{\Omega(N)} L(\omega)(N \cdot \omega) d\omega \tag{3.1}$$

where N is the surface normal and $L(\omega)$ is the amount of light comming from the direction ω. It is not possible to compute such an integral for surface point of our object in real-time.

We will use a technique using *Spherical Harmonics* to approximate the diffuse lighting comming from an environment map and to efficiently compute the diffuse irradiance at each surface point very efficiently. This technique has been introduced in 2001 by Ravi Ramamoorthi and Pat Hanrahan with their work [17]. First, we will introduce the concept of spherical harmonics.

3.1 Spherical harmonics

First, consider that we have a 1D function $f(x)$ and we want compute an approximation $\tilde{f}$ of that function with a weighted sum of basis functions $b_i(x)$. Therefore, we want to have :

$$\tilde{f}(x) = \sum_i c_i b_i(x) \tag{3.2}$$

where c_i is the weight of the basis function b_i. To find those weights, we need to *project* the original function $f(x)$ onto each basis function $b_i(x)$. We can do this by computing the integrals :

$$c_i = \int f(x) b_i(x) dx$$

Then by using equation 3.2, we can compute an approximation of the original function $f(x)$. But now, take a look at the equation 3.1 that computes the irradiance over a hemisphere. We can consider that the function $L(\omega)$ is a function over the surface of the sphere. How can we use the technique we have just seen with a 1D function to approximate a lighting function $f(s)$ over a 2D surface of a sphere S. To do this, we can use *Spherical Harmonics*. The spherical harmonics are the basis functions over the surface of the sphere. Robin Green has written a very nice document [11] that explains the theory of spherical harmonics.

Consider the standard parameterization of points on the surface of a unit sphere into spherical coordinates :

$$\begin{cases} x = \sin\theta\cos\phi \\ y = \sin\theta\sin\phi \\ z = \cos\theta \end{cases}$$

The spherical harmonics basis functions $y_l^m(\theta, \phi)$ are defined by :

$$y_l^m(\theta,\phi) = \begin{cases} \sqrt{2} K_l^m \cos(m\phi) P_l^m(\cos\theta) & \text{if } m > 0 \\ \sqrt{2} K_l^m \sin(-m\phi) P_l^{-m}(\cos\theta) & \text{if } m < 0 \\ \sqrt{2} K_l^0 P_l^0(\cos\theta) & \text{if } m = 0 \end{cases}$$

where P_l^m are the Legendre polynomials and K_l^m is a scaling factor given by :

$$K_l^m = \sqrt{\left(\frac{(2l+1)(l-|m|)!}{4\pi(l+|m|)!}\right)}$$

For the following, we will consider the spherical harmonics basis functions $y_i(\theta, \phi)$ defined by :

$$y_i(\theta,\phi) = y_l^m(\theta,\phi) \quad \text{where } i = l(l+1)+m \quad \text{with } l \in \mathbb{R}^+ \quad \text{and } -l \leq m \leq l$$

Now, the idea is to approximate the diffuse lighting function $f(s)$ over the surface of the sphere S with a weighted sum of the spherical harmonics basis functions $y_l^m(\theta, \phi)$. Therefore, we need to find the corresponding weights (or spherical harmonics coefficients) c_l^m. As we have seen before for a 1D function is quite simple to find the coefficients c_l^m, we just need to integrate the product of the function f and the spherical harmonic basis function y_l^m. Therefore, we have :

$$c_l^m = \int_S f(s) y_l^m ds \tag{3.3}$$

But we need to compute this integral numerically and therefore we can use *Monte Carlo Integration*. Monte Carlo integration allows us to compute the integral of a function $f(x)$ as follows :

$$\int f(x)dx \approx \frac{1}{N}\sum_{j=1}^{N} f(x_j) w(x_j)$$

where N is the number of samples $f(x_j)$ of the function f that we have and $w(x_j)$ is given by :

$$w(x_j) = \frac{1}{p(x_j)}$$

where $p(x)$ is the probability distribution function of the samples.

Therefore, if we consider again the equation 3.3, we have, using spherical coordinates :

$$c_i = \int_0^{2\pi}\int_0^{\pi} f(\theta,\phi) y_i(\theta,\phi)\sin\theta d\theta d\phi$$

If we choose the samples for Monte Carlo Integration such that they are unbiased w.r.t. surface on the sphere, each sample has equal probability of appearing anywhere on the surface of the sphere which gives us a probability function :

$$p(x_j) = \frac{1}{4\pi}$$

Therefore, using Monte Carlo Integration, if we have the samples $f(x_j)$, we can compute the spherical harmonics coefficients as follows :

$$c_i = \frac{4\pi}{N} \sum_{j=1}^{N} f(x_j) y_i(x_j)$$

Then we can compute an n-th order approximation $\tilde{f}$ of the function f with :

$$\tilde{f}(s) = \sum_{l=0}^{n-1} \sum_{m=-l}^{l} c_l^m y_l^m(s) = \sum_{i=0}^{n} c_i y_i(s) \tag{3.4}$$

Note that for a n-th order approximation we need n^2 spherical harmonic coefficients.

3.1.1 Properties of Spherical Harmonics

Spherical harmonics have some very nice properties. For instance, the spherical harmonics functions are rotationally invariant, meaning that if a function g is a rotated copy of a function f, then after the spherical harmonic projection, we have :

$$\tilde{g}(s) = \tilde{f}(R(s))$$

This is useful because it means that by using spherical harmonic functions we can guarantee that when we animate scenes, move lights or rotate the objects, the intensity of lighting will not fluctuate or have any artifacts.

The other very nice property is that integrating the product of two spherical harmonic functions is the same as evaluating the dot product of their coefficients. Consider two spherical harmonic functions $\tilde{f}(s)$ and $\tilde{g}(s)$, we have :

$$\int_S \tilde{f}(s)\tilde{g}(s)ds = \sum_{i=0}^{n} f_i g_i$$

where f_i and g_i are the spherical harmonics coefficients of the functions f and g.

3.2 Spherical harmonics for environment lighting

Now that we have seen the theory of spherical harmonics, we need to see how to use them for environment lighting. As we have said, the environment light irradiance (represented by an environment map) can be seen as a 2D function $L(s)$ over the surface of the sphere S. The previous section tells us how to project such a function onto the spherical harmonics basis functions y_i to get the corresponding spherical harmonics coefficients c_i. Then for a given environment map, we can precompute this projection offline to get the coefficients c_i and then at rendering time, we can compute equation 3.4 to approximate the irradiance at a given point of the surface of our object. The more spherical harmonics coefficients we use the better is the approximation. But that doesn't help a lot so far. What is very interesting is that in [17], they have shown that only a third order approximation is enough to approximate the diffuse irradiance because the diffuse irradiance is a quite low frequency function. Remember that for a third order approximation, we only need 9 spherical harmonics coefficients.

This is really useful because we can precompute only 9 spherical harmonics coefficients offline for a given environment map and at rendering time we just have to compute the equation 3.4 with $n = 3$ which is much more efficient than evaluating an integral such as in equation 3.1 for each surface point of the object.

In [17], they have computed the spherical harmonics coefficients for a certain number of light probes environment maps that can be found on `http://ict.debevec.org/~debevec/Probes/`. Notice that each of the spherical harmonic coefficient is separated in the three color channels resulting in a total of 27 coefficients for a given environment map.

3.3 Spherical harmonics and occlusions

The main issue of the environment lighting technique we have just seen is that we didn't take occlusions into account. Rendering with occlusions is realy important to obtain a realistic result. The good point is that we can extend what we have seen before about spherical harmonics to take care of occlusions. Consider that at each vertex we have a visibility function $V(\omega)$ that is 1 if the point on the surface of the sphere S in the direction ω is visible from that vertex and 0 if it is occluded. Therefore, at each point on the surface of our object we want to compute the diffuse irradiance E which is given by :

$$E = \int_S L(\omega)V(\omega)(N \cdot \omega)d\omega$$

We can rewrite this as :

$$E = \int_S L(\omega)g(\omega)d\omega$$

where $g(\omega) = V(\omega)(N \cdot \omega)$. Now, we can see that L is a function that depends only on the environment map and that g is a function that depends only on the geometry of the mesh. Thus, at each point of the surface of our object, we need to compute this integral of the product of the functions L and g. But we have seen in section 3.1 that integrating the product of two spherical harmonics functions is equivalent to evaluating the dot product of their spherical harmonics coefficients. Therefore, we can compute (by projection) the spherical harmonics coefficients L_i and g_i and then at rendering time, for each point of the surface of the object, we only have to compute :

$$E = \int_S L(\omega)g(\omega)d\omega = \sum_{i=0}^{n} L_i g_i \tag{3.5}$$

The coefficients L_i comes from the projection of the environment lighting function onto spherical harnomics basis functions. They are the same 9 coefficients of the previous section that can be precomputed given an environment map.

We can compute the others spherical harmonics coefficients g_i by projecting the function g onto the spherical harmonics basis function. Notice that this function is different at each point on the surface of the object because it depends on the normal vector N and the visibility function $V(\omega)$. Projecting the function g will also give us 9 spherical harmonics coefficients g_i. Therefore, we need to compute those 9 coefficients g_i at each point of the surface of our object. What we can do instead is to approximate this by computing those coefficients only at each vertex of the mesh (which seems to be a good approximation if our head model contains a large number of faces). This projection is precomputed offline for our head object.

How can we precompute the 9 coefficients g_i for a given vertex of the mesh. Remember that $g(\omega)$ is a function defined on the surface of a sphere for each vertex. Because we want to project this function onto the spherical harmonics basis using Monte Carlo integration, we have to generate some samples on the surface of the sphere. Then for each sample direction ω_j, we need to evaluate $g(\omega_j)$, which means we need to compute the dot product $N \cdot \omega_j$ where N is the normal of the vertex and also we have to evaluate the visibility term $V(\omega_j)$ at the current vertex. To do this,

I have used the idea found in [15] that consist of placing the camera at the vertex position and rendering the mesh into a cubemap (the cubemap is cleared in white and the mesh is rendered is black). Thus, at each vertex, we have a way to evaluate the visibility term $V(\omega_j)$. We only have to lookup in the cubemap in the direction ω_j and if the cubemap texel is this direction is white it means the vertex is not occluded in this direction and if it is black, the vertex is occluded by another part of the mesh. For each vertex, and for each sample direction ω_j we send the values N, ω_j and the cubemap to a fragment shader that computes the evaluation of the sample $g(\omega_j)$. The listing 3.1 shows the corresponding code that can be found in the `shOcclusionShader` fragment shader.

```
    // Get the sample light direction from the texture
    vec3 sampleDir = texture2D(texSampleDir ,gl_TexCoord[0].st).rgb;

    // Get the texture coordinate for the occlusion information
    vec2 indirection = textureCube(texCBOIndirection, sampleDir).ra;

    // Get the information if the vertex is occluded or not
    float isOccluded = texture2D(texCBO, indirection).r;

    // Store the occlusion term and dot product of light direction and normal
    gl_FragColor = vec4(isOccluded * dot(sampleDir, normalDir));
```

Listing 3.1: Code to evaluate the funtion $g(\omega_j)$ at a given sample direction ω_j

Now, we can evaluate the function g at each vertex in a certain number of sample directions ω_j which allows us to compute by Monte Carlo Integration the projection of the function g onto spherical harmonics basis functions to get the 9 coefficients g_i at each vertex of the mesh.

Because the coefficients L_i and g_i are precomputed offline, at rendering time, we only have to compute, at each point of the surface of the object, the dot product of equation 3.5 to obtain the diffuse irradiance comming from the environment lighting. This is done when computing the irradiance in the `irradianceShader` vertex shader. The listing 3.2 shows to corresponding code. Notice that the lighting coefficients L_i contains color information and therefore we have one spherical harmonic coefficient L_i per color channel which gives us in total 27 coefficients. The spherical harmonics coefficients g_i don't contain color information and therefore we have nine coefficients per vertex.

```
    // Compute the environment lighting color
    envColor = vec3(0.0);
    envColor += vertexSHCoeff1_2_3.r * envmapSHCoeffL00;
    envColor += vertexSHCoeff1_2_3.g * envmapSHCoeffL1m1;
    envColor += vertexSHCoeff1_2_3.b * envmapSHCoeffL10;
    envColor += vertexSHCoeff4_5_6.r * envmapSHCoeffL11;
    envColor += vertexSHCoeff4_5_6.g * envmapSHCoeffL2m2;
    envColor += vertexSHCoeff4_5_6.b * envmapSHCoeffL2m1;
    envColor += vertexSHCoeff7_8_9.r * envmapSHCoeffL20;
    envColor += vertexSHCoeff7_8_9.g * envmapSHCoeffL21;
    envColor += vertexSHCoeff7_8_9.b * envmapSHCoeffL22;
```

Listing 3.2: Code to compute the environment lighting

The figure 3.1 shows the environment lighting in our application. The environment lighting is done using the spherical harmonics coefficients obtained in [17] by precomputing the environment light probe of the Grace's Cathedral from `http://ict.debevec.org/~debevec/Probes/`. The left image is the environment lighting without occlusions and on the right with occlusions. As you can see, the occlusions allow to obtain a much better result near the ears, the eyes and around the nose. We can also observe the shadow behind the ear. As you can see, the approximation of computing the coefficients g_i only at the vertex is quite good for our mesh.

Figure 3.1: *Environment lighting without occlusions (left) and with occlusions (right)*

3.4 Rotation of Spherical Harmonics Coefficients

As we have said before, the spherical harmonics functions are rotationally invariant. Notice that if we rotate the mesh, we also need to rotate the spherical harmonics coefficients of the light L_i in order that the environment lighting remains correct. How can we rotate the spherical harmonics coefficients ?

We can use the technique from [11], the idea is to represent our 9 spherical harmonics coefficients by a 9×1 vector C and to multiply it by a 9×9 rotation matrix $R_{SH}(\alpha, \beta, \gamma)$ that depends on the rotation Euler angles α, β and γ. We can decompose the R_{SH} matrix using the ZYZ formulation. It means that first we rotate about the Z axis, then around the rotated Y axis and finally around the rotated Z axis. With this formulation, we can generate rotations around any possible orientation. The rotation of an angle β around the Y axis can be decomposed as a rotation X_{+90} of 90 degrees around the X axis, a rotation Z_β of an angle β around the Z axis a a rotation X_{-90} of -90 degrees around the X axis. Therefore, we have :

$$R_{SH}(\alpha, \beta, \gamma) = Z_\gamma Y_\beta Z_\alpha = Z_\gamma X_{-90} Z_\beta X_{+90} Z_\alpha$$

Therefore, we only need the three rotation matrices Z, X_{+90} and X_{-90}. The Z matrix can be computed as follows given an angle θ :

$$Z_\theta = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \cos(\theta) & 0 & \sin(\theta) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -\sin(\theta) & 0 & \cos(\theta) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cos(2\theta) & 0 & 0 & 0 & \sin(2\theta) \\ 0 & 0 & 0 & 0 & 0 & \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 0 & -\sin(2\theta) & 0 & 0 & 0 & \cos(2\theta) \end{pmatrix}$$

and the matrix X_{+90} is defined by :

$$X_{+90} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 0 & -\frac{\sqrt{3}}{2} \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \end{pmatrix}$$

And the last matrix X_{-90} is the transposed matrix of X_{+90}. Therefore, given the three Euler angles of rotation of our mesh α, β and γ, we can compute the rotation matrix R_{SH} and multiply it by the spherical harmonics coefficients vector C to obtain a new 9×1 vector that contains the rotated spherical harmonics coefficients of the environment lighting. Notice that we should use the fact that the matrices Z, X_{-90} and X_{+90} are quite sparse to make the rotation computation more efficient.

Chapter 4

Results

4.1 Conclusion

I have implemented the skin rendering algorithm introduced in [7] on the GPU. It seems to work quite well. There is still some artifacts with the global scattering using Modified Translucent Shadow Map. The shadow mapping also wasn't so easy to implement because of the several artifacts that arise when using shadow maps. Finally, I think that the rendering images that I have obtained are quite realistic.

During this project, I learned a lot about physically-based rendering. Especially about subsurface scattering, shadow mapping, translucent shadow maps and environment lighting using spherical harmonics. I've also improved my skills with GPU programming.

4.2 Future work

Now, we will see what can be done in the future to improve or extend the application.

As I have already said earlier, for the current application we use only a constant roughness parameter m over the entire face. Instead, we can use a texture map in which the roughness parameter is stored and is allowed to vary over the face. This would give a more realistic result because it would be more physically plausible.

Texture seams can generate problems for texture-space diffusion, because connected regions on the 3D mesh are disconnected in the texture space and cannot easily diffuse light between them. Indeed, the empty regions on the texture will blur onto the mesh along each seam edge which causes artifacts in the final rendering. A solution to this problem is described in [6]. The idea is to use a map or alpha value to detect when the irradiance textures are being accessed in a region near a seam (or an empty space). When such a place is detected, the subsurface scattering computation is turned off and the diffuse reflectance is replaced with an equivalent local computation. The amount of artifacts depends on the texture of the mesh. It can be important to take care of the seam artifacts because if they are strongly visible, the rendering will not look very realistic.

Another way to improve a lot the rendering would be to use High Dynamic Range (HDR) rendering. With HDR rendering, the realism of the scene can be improved especially when the scene contains for instance very glossy specular highlights. The current application is using the GLUT library for managing the OpenGL window. Using another windows manager could allow us to use more sample buffers for multisampling. The more sample buffers we use the better we can deal with antialiasing of the scene. Therefore, by using more sample buffers, we could increase the quality of the scene of the application.

4.3 Rendered images

Here is some rendering that I have obtained with the application.

Figure 4.1: *Rendering with three light sources and the environment lighting from the Uffizi Gallery light probe*

Figure 4.2: *Rendering with only one positional light source*

Figure 4.3: *Rendering with two light sources and the environment lighting from the Grace Cathedral light probe*

Figure 4.4: *Rendering with two light sources and the environment lighting from the Uffizi Gallery light probe*

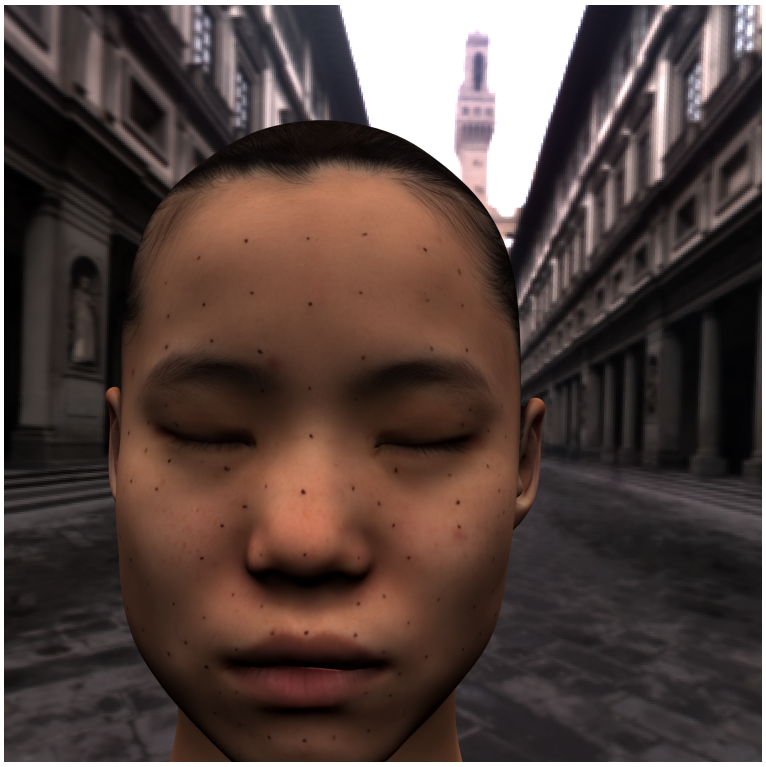

Figure 4.5: *Rendering with environment lighting from the Uffizi Gallery light probe with a girl mesh*

Figure 4.6: *Rendering with two light sources and environment lighting from the Uffizi Gallery light probe*

Appendix A

The Skin Rendering Application

A.1 About the Application

With the application, it is possible to load the mesh that I used during this project and to render it using the subsurface scattering algorithm. The application contains three positional light sources. Is is also possible to activate the environment lighting with the spherical harmonics coefficients computed in [17] for the Uffizi Gallery light probe from `http://ict.debevec.org/~debevec/Probes/`. The spherical harmonics coefficients for the occlusions at each vertex of the mesh have already been precomputed and are available in the file `meshRoyaltyShCoeff.txt`. But it is also possible to precompute them and put them into a file using the application.

The parameters of the skin rendering algorithm like the roughness parameter `m` or the `mixRatio` for pre-scatter texturing can be changed easily and can be found in the `PhotoRealistic.hhp` file.

The code for precomputing the spherical harmonics coefficients for the occlusions at each vertex can be found in the `Mesh3D.cpp` file.

A.2 How to use the application

To run the application, use the following command :

```
./photorealistic dataDirectory/ [computeVertexSHCoeffs]
```

where `dataDirectory` is the directory where the data needed by the application are stored like the mesh, the textures, ... The second parameter is optional and can be either y (yes) or n (no). If this parameter is `y`, the spherical harmonics coefficients for the occlusions for each vertex are precomputed when the application starts and will be used later to render the scene while the application is running. By default, those coefficients are not computed at the beginning of the application but are loaded from a file.

For instance, we can run the application with the command :

```
./photorealistic data/
```

While the application is running, it is possible to use the following commands :

key 0 : enable/disable the light source 0.

key 1 : enable/disable the light source 1.

key 2 : enable/disable the light source 2.

key a : move the light source 0 to the left around the vertical axis.

key d : move the light source 0 to the right around the vertical axis.

key w : move the light source 0 up around the horizontal axis.

key s : move the light source 0 down around the horizontal axis.

key e : enable/disable the environment lighting.

key z : enable/disable the specular lighting.

key o : enable/disable the shadows.

key b : enable/disable subsurface scattering.

key y : enable/disbale the Translucent Shadow Map for global scattering.

key h : rotate the mesh around the vertical axis.

key j : rotate the mesh around the horizontal axis.

mouse : use the mouse to rotate the camera around the mesh.

Bibliography

[1] Beckmann distribution. `http://en.wikipedia.org/wiki/Specular_highlight#Beckmann_distribution`.

[2] Nvidia demo team secrets: Advanced skin rendering - gdc 2007 demo. `http://developer.download.nvidia.com/presentations/2007/gdc/Advanced_Skin.pdf`.

[3] Donner C. and Jensen H. W. Light diffusion in multi-layered translucent materials. *ACM Trans. Graph*, 2005.

[4] Schlick Christophe. A customizable reflectance model for everyday rendering. 1993.

[5] Carsten Dachsbacher and Marc Stamminger. Translucent shadow maps. 2004.

[6] Eugene d'Eon and David Luebke. *Advanced Techniques for Realistic Real-Time Skin Rendering, GPU Gems 3.* 2008.

[7] Eugene d'Eon, David Luebke, and Eric Enderton. Efficient rendering of human skin. 2007.

[8] Donner, Craig, and Henrik Wann Jensen. A spectral bssrdf for shading human skin. 2006.

[9] Borshuko George and J.P. Lewis. Realistic human face rendering for the matrix reloaded. 2003.

[10] Mitchell J. L Gosselin D., Sander P. V. *Real-time texture-space skin rendering.* 2004.

[11] Robin Green. Spherical harmonic lighting: The gritty details. 2003.

[12] Simon Green. Real-time approximations to subsurface scattering. *GPU Gems*, 2004.

[13] Larry Gritz and Eugene d'Eon. *GPU Gems 3, The Importance of Being Linear.* 2007.

[14] Poirer Guillaume. Human skin modeling and rendering. 2004.

[15] Sebastien Hillaire. Spherical harmonics lighting. http://sebastien.hillaire.free.fr/demos/sh/sh.htm.

[16] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. 2001.

[17] Ravi Ramamoorthi and Pat Hanrahan. An efficient representation for irradiance environment maps. 2001.

[18] Heidrich Wolfgang and Hans-Peter Seidel. Realistic, hardware-accelerated shading and lighting. 1999.