Screen-Space Subsurface Scattering

Jorge Jimenez and Diego Gutierrez

7.1 Introduction

Many materials exhibit a certain degree of translucency, by which light falling onto an object enters its body at one point, scatters within it, then exits the object at some other point. This process is known as *subsurface scattering*. We observe many translucent objects in our daily lives, such as skin, marble, paper, tree leaves, soap, candles, and fruit. In order to render these materials in a realistic way, we must recreate the effects of subsurface scattering in the rendering pipeline.

Skin is one of the most important materials that demonstrates a significant degree of subsurface scattering. Many video games are very character and story driven. These kinds of games seek to create believable, realistic characters so that the player becomes fully immersed in the game. Adding accurate subsurface scattering to human skin, especially on faces, can dramatically improve the overall impression of realism.

There has been an emerging trend towards applying computationally expensive three-dimensional methods (such as ambient occlusion or global illumination) in screen space. We present an algorithm capable of simulating subsurface scattering in screen space as a post-process (see Figure 7.1), which takes as inputs the depth-stencil and color buffer of a rendered frame.

In this article we will describe a very efficient screen-space implementation of subsurface scattering. We will measure the cost of our technique against other common screen-space effects such as depth of field or bloom, in order to motivate its implementation in current game engines. As we will see, our method maintains the quality of the best texture-space algorithms but scales better as the number of objects increases (see Figure 7.2).

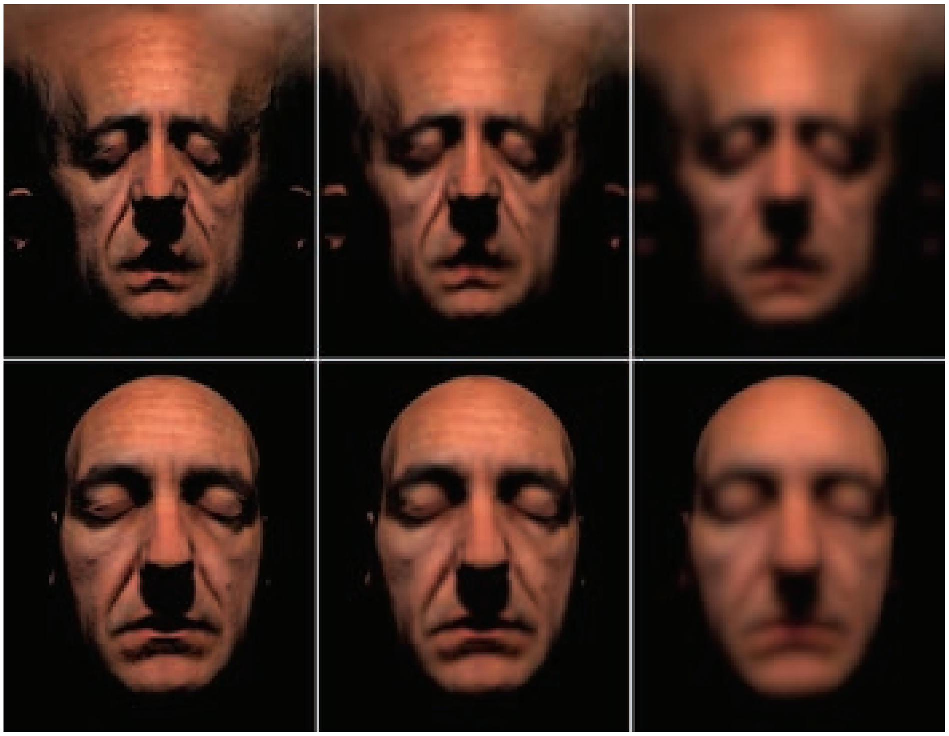

Figure 7.1. Blurring performed in texture space, as done by current real time subsurface scattering algorithms (top). Blurring done directly in screen space (bottom).

7.2 The Texture-Space Approach

7.2.1 Subsurface Scattering and Diffusion Profiles

Homogeneous subsurface scattering can be approximated by using one-dimensional functions called diffusion profiles. A diffusion profile defines how the light attenuates as it travels beneath the surface of an object, or in other words, it describes how light intensity ($R(r)$) decays as a function of the radial distance to the incidence point (r). As shown in Figure 7.3, a majority of the light intensity occurs close to the point of incidence, and quickly decays as it interacts with the inner structure of the object before escaping through the surface.

7.2.2 Irradiance Texture

Applying a diffusion profile implies calculating the irradiance at incident and adjacent points for each point of the surface, which leads to wasted calculations.

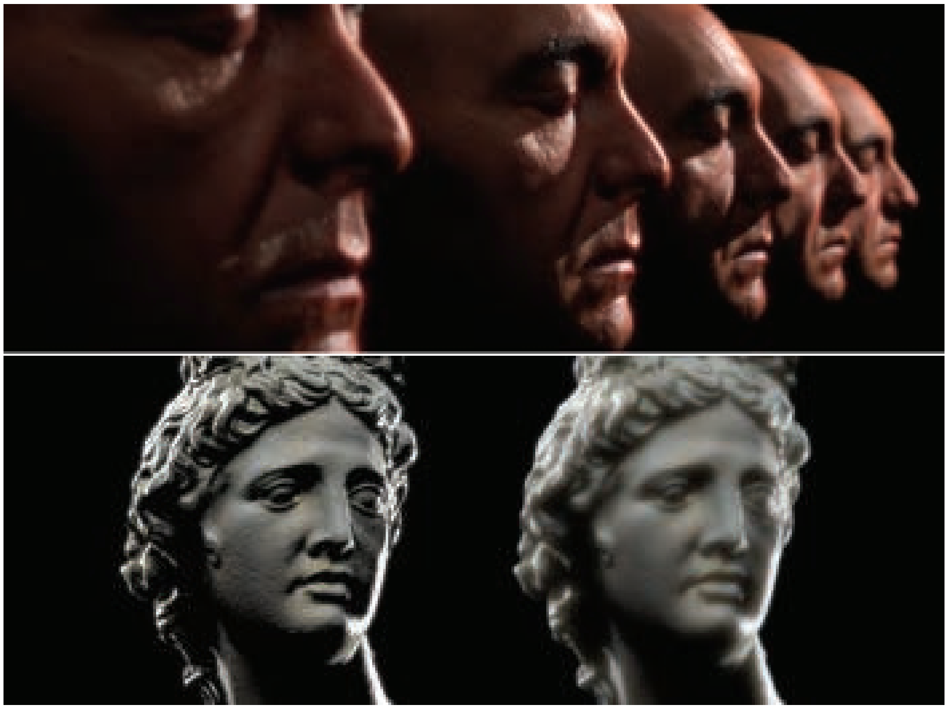

Figure 7.2. Examples of our screen-space approach. Unlike the texture-space approach, our method scales well with the number of objects in the scene (top). Rendering marble without taking subsurface scattering into account leads to a stone-like appearance (bottom left); our subsurface scattering technique is used to create a softer appearance, more indicative of subsurface scattering (bottom right).

In order to perform an efficient profile evaluation, an irradiance map—also known as light map—is created which stores the incoming light at each point of the surface, and serves as light *cache*, in the sense that it enables you to calculate the lighting at each point once but use that value many times. Figure 7.1 (top left) shows an example of an irradiance texture.

7.2.3 Gaussians and the Jittered Kernel

Once an irradiance map is calculated, applying a diffusion profile consists of no more than applying a two-dimensional convolution over this map. There are a few previous works that provide detailed insights into how to perform this convolution in an efficient way, all for the specific case of skin rendering.

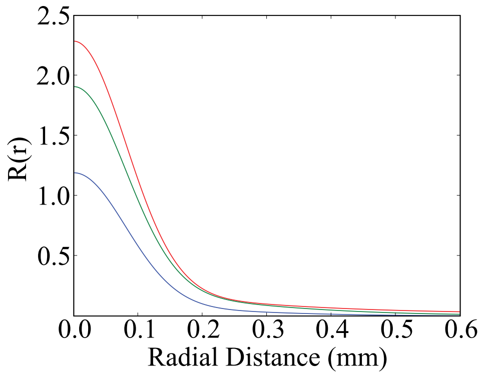

Figure 7.3. Diffusion profile of the three-layer skin model described in [d'Eon and Luebke 07].

The approach followed by d'Eon and Luebke [d'Eon and Luebke 07], consists of approximating this lengthy two-dimensional convolution by a sum of Gaussians, which can be separated into faster 1D convolutions. To summarize, rendering subsurface scattering using this sum-of-Gaussians consists of the following steps:

1. Render the irradiance map.
2. Blur the irradiance map with six Gaussians.
3. Render the scene, calculating a weighted sum of the six Gaussians, which will approximate the original diffusion profile.

While rendering an irradiance map (and in the following blurs), two optimizations that would otherwise be implicitly performed by the GPU are lost, namely backface culling and view frustum clipping. Jimenez and Gutierrez [Jimenez and Gutierrez 08] reintroduce those two optimizations in the rendering pipeline proposed in [d'Eon and Luebke 07]. They also perform an optimal, per-object modulation of the irradiance map size based on a simple, depth-based method.

Hable et al [Hable et al. 09], using a 13-sample jittered kernel, together with a small 512×512 irradiance map and a similar culling optimization, managed to improve the performance over the 6-Gaussian implementation [d'Eon and Luebke 07], at the cost of a loss of *fleshiness*, as the authors noticed.

7.2.4 Texture-Space Diffusion Problems

Current real-time subsurface scattering algorithms rely on texture-space diffusion, which has some intrinsic problems that can be easily solved by working in screen space instead. We outline the most important ones:

- It requires special measures to bring back typical GPU optimizations (back-face culling and viewport clipping), and to compute an irradiance map proportional to the size of the subject on the screen. In screen space, these optimizations are straightforward.
- Each subject to be rendered requires her own irradiance map (thus forcing as many render passes as subjects). In image space, *all subjects* of the same material are processed at the same time.
- Furthermore, rendering large amounts of objects with subsurface scattering is not efficient as either you have to use an irradiance map for each object, in order to be able to use instancing (which would be especially useful for rendering tree leaves for example), or you have to reuse the same irradiance map, and render each object sequentially.
- The irradiance map forces the transformation of the model vertices twice: during the irradiance map calculation, and to transform the final geometry at the end. In screen-space, only this second transformation is required.
- In texture space, for materials like skin that directly reflect a significant amount of light, lighting is usually calculated twice (once for the irradiance map and again for the sharpest of the Gaussian blurs since this Gaussian blur is usually replaced by the unblurred lighting in the final compositing pass). In screen-space, lighting must be calculated only once.
- Modern GPUs can perform an early-Z rejection operation to avoid overdraw and useless execution of pixel shaders on certain pixels. In texture space, it is unclear how to leverage this and optimize the convolution processes according to the final visibility in the image. In screen space, a depth pass can simply discard occluded fragments before sending them to the pixel shader.
- Adjacent points in three-dimensional world space may not be adjacent in texture space. Obviously this will introduce errors in texture-space diffusion that are naturally avoided in screen space. This implies that special care must be taken when preparing the UV map of the models, in order to reduce the number of seams as much as possible.

7.3 The Screen-Space Approach

7.3.1 The Big Picture

Our algorithm translates the evaluation of the diffusion approximation from texture- to screen-space. Instead of calculating an irradiance map and convolving

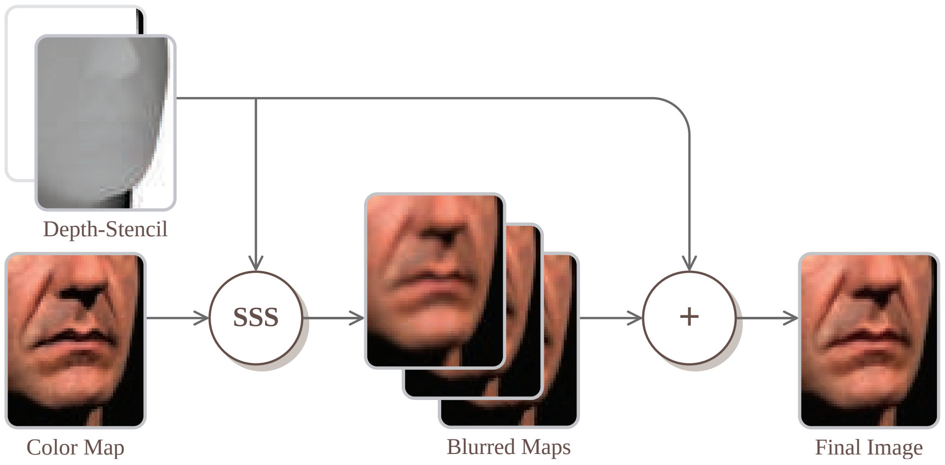

Figure 7.4. Conceptual interpretation of our algorithm (see text in the article for actual implementation details).

it with the diffusion profile, we apply the convolution directly to the final rendered image. Figure 7.4 shows an overview of our algorithm. As we have said, to perform this task in screen-space we need as input the depth-stencil and color buffer of a rendered frame.

As subsurface scattering should only be applied to the diffuse component of the lighting function, we could store the diffuse and specular components separately, for which we could use multiple render targets or the alpha channel of the main render target. However, we have found that applying subsurface scattering to both the diffuse and specular components of the lighting yields very appealing results, because a) the specular component is softened creating a more natural look, with less apparent aliasing artifacts, and b) it creates a nice bloom effect for the specular highlights.

In order to efficiently mask out pixels that do not need to be processed we use the stencil buffer. This vastly improves the performance when the object is far away from the camera, and thus occupies a small area in screen space. We also tried to use dynamic branching in our pixel shaders but we found the stencil method to be more efficient particularly when a large percentage of the screen must be rejected.

Our algorithm also requires linear depth information [Gillham 06] which we render into an additional render target. We use this depth together with the original depth-stencil map, which serves to perform the stenciling and to implement a depth-based Gaussians level of detail, as we will see in the next section. It is possible to access the depth buffer directly, but we are not able to simultaneously use the same depth buffer both as an input texture and as depth stencil buffer, so we must create a copy.

With the depth-stencil information as input, we apply the diffusion profile directly to the final rendered image, instead of applying it to a previously calculated irradiance map. For skin rendering, we use our own four-Gaussian fit of the three-layer skin model defined in [d'Eon and Luebke 07] (see Section 7.3.3 for details), and, as done in this work, perform each of them as two separated one-dimensional convolutions (horizontal and vertical blurs). The results obtained using our four-Gaussian fit are visually indistinguishable from the original six-Gaussian fit. For marble we calculated the profile using the parameters given by [Jensen et al. 01], for which we obtained a four-Gaussian fit (see [d'Eon and Luebke 07] for details on how to perform this fitting).

In screen space we need to take into account the following considerations:

- Pixels that are far from the camera need narrower kernel widths than pixels that are close to the camera.
- Pixels representing surfaces that are at steep angles with respect to the camera also need narrower kernels.

These considerations translate into the following *stretch factors*:

$$s_x = \frac{\text{ssslevel}}{d(x,y) + \text{correction} \cdot \min(abs(ddx(d(x,y), \text{maxdd})},$$
$$s_y = \frac{\text{ssslevel}}{d(x,y) + \text{correction} \cdot \min(abs(ddy(d(x,y), \text{maxdd})},$$

where $d(x,y)$ is the depth of the pixel in the depth map, "ssslevel" indicates the global subsurface scattering level in the image, "correction" modulates how this subsurface scattering varies with the depth gradient, and "maxdd" limits the effects of the derivative. This derivative limitation is required for very soft materials (like marble), as subtle artifacts may arise at depth discontinuities under very specific lighting conditions. These artifacts are caused by the huge derivatives found at those locations, that locally nullify the effect of subsurface scattering. The first and second terms in the denominators account for the first and second considerations listed above. This stretching is similar in spirit to the UV stretching performed in texture space algorithms.

The *stretch factors* are then multiplied by each Gaussian width in order to obtain the final kernel width:

$$\text{finalwidth}_x = s_x \cdot \text{width},$$
$$\text{finalwidth}_y = s_y \cdot \text{width}.$$

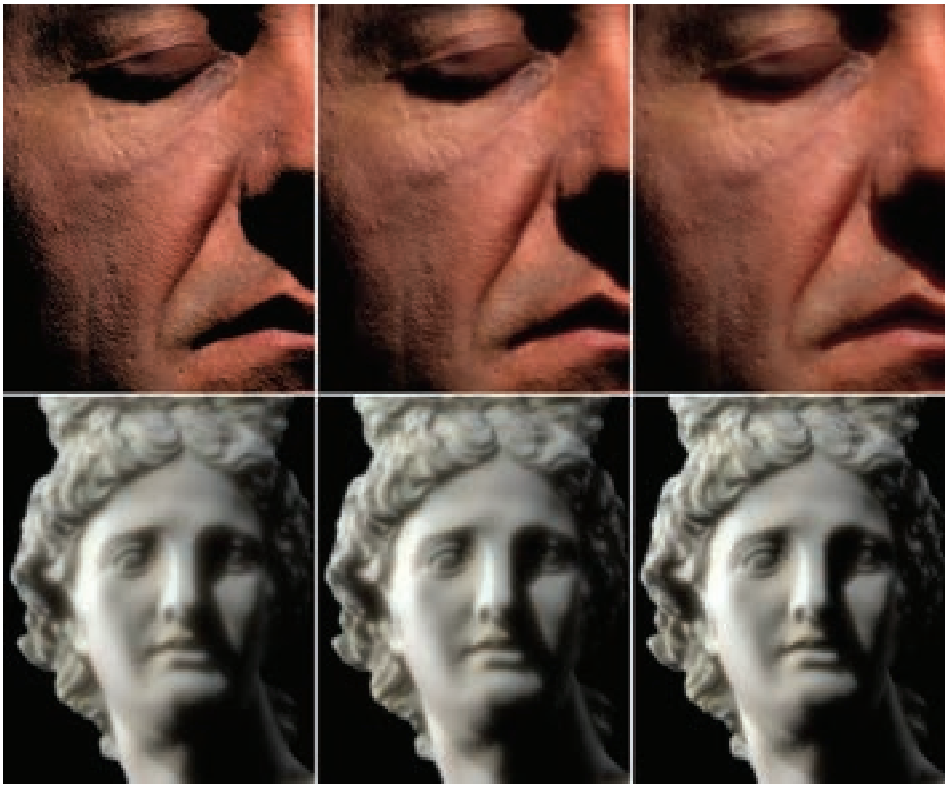

Figure 7.5. The influence of the ssslevel and correction parameters. Fixed correction = 800, maxdd = 0.001 and varying ssslevel of 0, 15.75 and 31.5, respectively (top). Note how the global level of subsurface scattering increases. Fixed ssslevel = 31.5, maxdd = 0.001 and varying correction of 0, 1200 and 4000, respectively (bottom). Note how the gray-to-black gradient on the nose gets readjusted according to the depth derivatives of the underlying geometry (the range of the correction parameter has been extended for visualization purposes; it usually is limited to [0..1200] because of the limited precision of the derivatives).

In the silhouette of the object, the derivatives will be very large, which means that the kernel will be very narrow, thus limiting the bleeding of background pixels into object's pixels. The value of ssslevel is influenced by the size of the object in three-dimensional space, the field-of-view used to render the scene and the viewport size (as these parameters determine the projected size of the object), whereas maxdd only depends on the size of the object. The images included in this article use fixed values of ssslevel = 31.5, correction = 800 and maxdd = 0.001;

```
float width;
float sssLevel, correction, maxdd;
float2 pixelSize;
Texture2D colorTex, depthTex;

float4 BlurPS(PassV2P input) : SV_TARGET {
  float w[7] = { 0.006, 0.061, 0.242, 0.382,
                 0.242, 0.061, 0.006 };

  float depth = depthTex.Sample(PointSampler,
                                input.texcoord).r;
  float2 s_x = sssLevel / (depth + correction *
                           min(abs(ddx(depth)), maxdd));
  float2 finalWidth = s_x * width * pixelSize *
                      float2(1.0, 0.0);

  float2 offset = input.texcoord - finalWidth;
  float4 color = float4(0.0, 0.0, 0.0, 1.0);
  for (int i = 0; i < 7; i++) {
    float3 tap = colorTex.Sample(LinearSampler, offset).rgb;
    color.rgb += w[i] * tap;
    offset += finalWidth / 3.0;
  }

  return color;
}
```

Listing 7.1. Pixel shader that performs the horizontal Gaussian blur.

these values were chosen empirically for a head 1.0 units tall, a field-of-view of 20°, and a viewport height of 720 pixels. Figure 7.5 illustrates the influence of ssslevel and correction in the final images.

Listing 7.1 shows the implementation of previous equation for the case of the horizontal blur. Our subsurface scattering approach just requires two short shaders, one for the horizontal blur and a similar one for the vertical blur.

Certain areas of the character, such as hair or beard, should be excluded from these calculations; we could therefore want to locally disable subsurface scattering in such places. For this purpose, we could use the alpha channel of the diffuse texture to modulate this local subsurface scattering level. We would need to store the alpha channel of this texture into the alpha channel of the main render target during the main render pass. Then, in our post-processing pass we would use the following modified stretch factors:

$$s'_x = s_x \cdot \text{diffuse}(x, y).a$$
$$s'_y = s_y \cdot \text{diffuse}(x, y).a$$

In the following, we describe the two key optimizations that our algorithm performs.

7.3.2 Depth-Based Gaussians Level of Detail

Performing the diffusion in screen space allows us to disable Gaussians on a per-pixel basis as we fly away from the model. Narrow Gaussians are going to have little effect on pixels far away from the camera, as most of the samples are going to land on the same pixel. We can exploit this to save computations.

For this purpose we use the following inequality, based on the final kernel width equation from the previous section, where we are making the assumption of camera facing polygons which have zero-valued derivatives:

$$\frac{\text{width} \cdot \text{ssslevel}}{d(x,y)} > 0.5.$$

If the width of the kernel for the current pixel is less than 0.5—that is, a half a pixel—all samples are going to land on the same pixel and thus we can skip blurring at this pixel (Figure 7.6, left).

We can use depth testing to efficiently implement this optimization. If we solve the previous inequality for $d(x, y)$:

$$2 \cdot \text{width} \cdot \text{ssslevel} > d(x,y),$$

then we just need to render the quad used to perform the convolution at a depth value of $2 \cdot \text{width} \cdot \text{ssslevel}$ and configure the depth testing function to *greater than*. However, we have found that instead using a value of $0.5 \cdot \text{width} \cdot \text{ssslevel}$ allows us to disable them much faster without any noticeable popping.

Though we have a copy of the depth-stencil buffer in linear space, the original depth-stencil buffer is kept in non-linear space for precision issues. This means that we have to transform the left side of previous inequality into the same non-linear space [Gillham 06], and clamp the resulting non-linear depth values to [0..1].

7.3.3 Alpha Blending Workflow

Using n Gaussians to approximate a diffusion profile means we need n render targets for irradiance storage. In order to keep the memory footprint as low as possible, we accumulate the sum of the Gaussians on the fly, eliminating the need to sum them in a final pass.

In order to accomplish this task, we require two additional render targets. The first render target is used to store the widest Gaussian calculated thus far (RT1) and the second for the usual render target ping ponging (RT2).

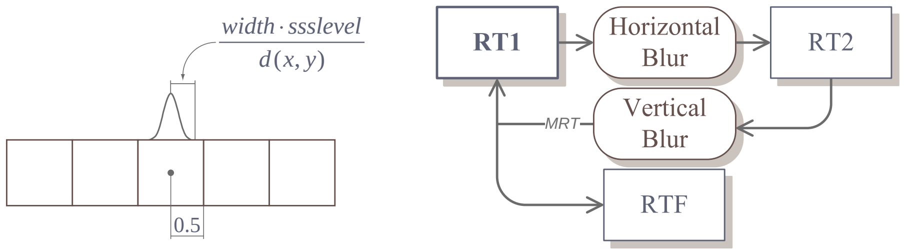

Figure 7.6. If a Gaussian is narrower than a pixel, the result of its application will be negligible (left). Alpha blending workflow used to enhance memory usage (right). Multiple Render Targets (MRT) is used to render to RT1 and RTF simultaneously.

Applying one of the Gaussians consists of two steps:

1. Perform the horizontal Gaussian blur into RT2.

2. Perform the vertical Gaussian blur by sampling from the horizontally blurred RT2 and outputting into both RT1 and RTF (the final render target) using multiple render targets. For RTF we use an alpha blending operation in order to mix the values accordingly. The exact weight required for the blending operation will be examined in the following paragraphs.

Figure 7.6 (right) shows the alpha blending workflow used by our algorithm. We need to sum the n Gaussians as follows:

$$R = \sum_{i=k}^{n} w_i G_i,$$

where w_i is the weight vector (different for each RGB channel) of each Gaussian G_i, and k is the first Gaussian that is wide enough to have a visible effect on the final image, as explained in the previous section.

As we may not calculate all Gaussians, we need to find a set of values w'_i that will produce a normalized result at each step i of the previous sum:

$$w'_i = \frac{w_i}{\sum_{j=1}^{i} w_i}.$$

Then we just need to configure alpha blending to use a blend factor as follows:

$$\text{color}_{\text{out}} = \text{color}_{\text{src}} \cdot \text{blendfactor} + \text{color}_{\text{dst}} \cdot (1 - \text{blendfactor}),$$

where blendfactor is assigned to w'_i. Listing 7.2 shows the FX syntax for the `BlendingState` that implements this equation.

```
BlendState BlendingAccum {
  BlendEnable[0] = FALSE; // RT2
  BlendEnable[1] = TRUE;  // RTF
  SrcBlend = BLEND_FACTOR;
  DestBlend = INV_BLEND_FACTOR;
};
```

Listing 7.2. Blending state used to blend a Gaussian with the final render target.

Table 7.1 shows the original weights of our skin and marble four-Gaussian fits (w_i), alongside with the modified weights used by our screen-space approach (w_i').

In the case of skin, the first Gaussian is too narrow to be noticeable. Thus the original unblurred image is used instead, saving the calculation of one Gaussian. Note that the weight of the first Gaussian for both fits is 1.0; the reason for this is that the first Gaussian does not need to be mixed with the previously blended Gaussians.

Note that as we are performing each Gaussian on top of the previous one, we have to subtract the variance of the previous Gaussian to get the actual variance. For example, for the widest Gaussian of the skin fit, we would have $2.0062 - 0.2719 = 1.7343$. Also keep in mind that the `width` that should be passed to the shader shown in the Listing 7.1 is the standard deviation, thus in this case it would be $\sqrt{1.7343}$.

Skin	w_i			w_i'		
Variance	**R**	**G**	**B**	**R**	**G**	**B**
0.0064	0.2405	0.4474	0.6157	1.0	1.0	1.0
0.0516	0.1158	0.3661	0.3439	0.3251	0.45	0.3583
0.2719	0.1836	0.1864	0.0	0.34	0.1864	0.0
2.0062	0.46	0.0	0.0402	0.46	0.0	0.0402

Marble	w_i			w_i'		
Variance	**R**	**G**	**B**	**R**	**G**	**B**
0.0362	0.0544	0.1245	0.2177	1.0	1.0	1.0
0.1144	0.2436	0.2435	0.1890	0.8173	0.6616	0.4647
0.4555	0.3105	0.3158	0.3742	0.5101	0.4617	0.4791
3.4833	0.3913	0.3161	0.2189	0.3913	0.3161	0.2189

Table 7.1. Gaussian variances and original weights (w_i) of our Gaussian fits that approximate the three-layer skin and marble diffusion profiles, alongside with the modified weights used by our algorithm (w_i').

7.3.4 Anti-aliasing and Depth-Stencil

For efficiency reasons, when using MSAA we might want to use the resolved render targets (downsampled to MSAA 1x) for performing all the post-processing, in order to save memory bandwidth. However this implies that we cannot use the original MSAA stencil buffer to perform the depth-based blur modulation or the stenciling.

We explored two approaches to solve this problem:

1. Use dynamic branching to perform the stenciling by hand.

2. Downsample the depth-stencil buffer to be able to use hardware stenciling.

Both methods rely on generating multi-sample render targets containing depth and stencil values during the main pass, for which multiple render targets can be used.[1]

We found that in our implementation using hardware stenciling outperforms the dynamic branching method.

7.4 Model Preparation

When using models with hand painted normals (as opposed to scanned models) there are some considerations that must be taken into account. As noticed in [Hable et al. 09], most artists paint normals trying to match the soft aspect of subsurface scattering, without being able to use algorithms that simulate this effect. Thus they usually resort to make the normals much softer than they are physically. However, when using subsurface scattering for rendering, the best results will be obtained using bumpier normals, as subsurface scattering will be in charge of softening the illumination.

7.5 Discussion of Results

Table 7.2 shows the performance of our screen space algorithm (SS) in the case of skin rendering, against the texture space algorithm proposed by Hable et al [Hable et al. 09] (TS). We have chosen [Hable et al. 09] because it outperforms the original [d'Eon and Luebke 07] algorithm for skin rendering. For other materials, for which there is no clear way of applying the approach by Hable et al, we resort to comparisons with [d'Eon and Luebke 07], which is much more flexible.

For skin simulation, the distances we have chosen have the following meaning: near is a close-up of the head, usually found on in-game cut scenes; medium is

[1]When using DirectX 10.1 we can sample from the MSAA depth-stencil buffer directly, with no need to output depth and stencil values using multiple render targets.

Heads	**TS**	**SS**	**Lights**	**TS**	**SS**	**Distance**	**TS**	**SS**
1	102/72	137/73	**2**	102/72	137/73	**Near**	60/52	50/40
3	47/33	67/52	**4**	63/56	90/64	**Medium**	102/72	137/73
5	29/21	50/38	**6**	54/43	69/54	**Far**	120/82	220/91

Table 7.2. Performance measurements (fps) of our algorithm (SS), using both 1×/8× MSAA, in comparison with the texture space approach proposed in Hable et al. [Hable et al. 09] (TS). For each test we fixed the following parameters: the number of heads, the number of lights and the distance were set to 1, 2, and medium, respectively.

the distance typically found for characters of first-person shooters; at far distance the head covers very little screen area, and the effects of subsurface scattering are negligible. In the following text we will quote the performance for 1×MSAA (see Table 7.2 for a more detailed report). We can see how our algorithm scales better as we increase the number of heads (from 29 to 50 fps for 5 heads), as the number of lights increases (from 54 to 69 fps for six lights) and as we fly away from the model (from 120 to 220 fps for far distance). In the case of 10 heads, two lights and far distance, we obtain a speedup factor of 3.6×. We performed an additional test to explore the impact of the early-z optimization on both algorithms, which shows that when rendering five heads behind a plane that hides them from the camera, our screen space algorithm manages to perform at 2× with respect to the texture-space approach. Similar speed-ups could be found when the objects fall outside of the viewport.

As we have discussed, there is no clear extension to Hable et al.'s algorithm that would allow for other translucent materials. We present an example of such materials by simulating marble (see Figure 7.2) and compare it with the approach of d'Eon et al. In this example, we achieve a speedup factor of up to 2.8× when rendering five Apollo busts with two lights at medium distance.

In general, the best speed ups are found when there are many objects (or very large objects that would require huge irradiance textures) at very different distances (for example, when rendering tree leaves). In this kind of situation, no single irradiance map size would be a perfect fit.

In the image shown in Figure 7.2 (top), where subsurface scattering is dominating the scene, the costs of the bloom, depth-of-field, and subsurface scattering effects are 2.7ms, 5.8ms, and 14ms, respectively. On the other hand, in a more typical scene with three heads at medium distance, the costs are 2.7ms, 5.8ms, and 4.6ms.

Our performance measurements were taken using a shadow map resolution of 512 × 512, diffuse and normal maps with a resolution of 1024 × 1024. The human head and Apollo bust models have 3904 and 14623 triangles, respectively,

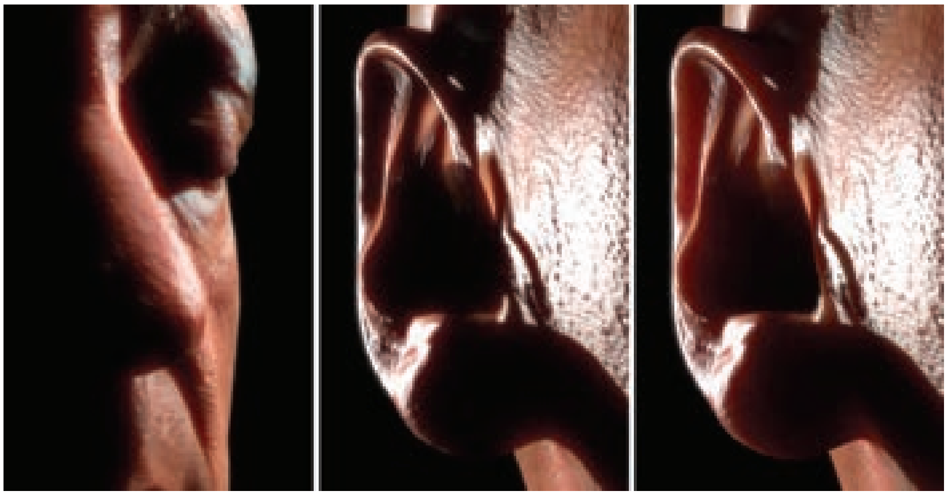

Figure 7.7. Screen space limitations and artifacts. Small haloes produced by incorrect diffusion from nose to cheek (left). Screen- vs. texture-space comparison where we can see how we cannot account for the diffusion produced in thin, high curvature features, as in screen space we do not have information from behind (Center and right).

both obtained from XYZRGB.[2] All renderings were performed on a machine equipped with a GeForce 8600M GS at a resolution of 1280×720. As recommended in [Gritz and d'Eon 07], we used sRGB render targets, which is crucial for multi-pass, physically based shaders like ours. For the specular highlights we used the Kelemen-Szirmay-Kalos approach [d'Eon and Luebke 07]. The rendered images shown in this article are filtered with a Gaussian bloom filter at the end of the rendering pipeline. Figure 7.2 (top) was rendered with an additional depth-of-field effect.

Quality-wise, the screen space approach manages to maintain the full *fleshiness* that the Hable et al. loses as result of using a small jittered kernel. Furthermore, it does not exhibit the stretching artifacts produced by texture space algorithms, especially by the Hable et al. approach [Hable et al. 09]. Finally, it is free from the seam problems imposed by the model's UV unwrapping (refer to the high resolution images available at http://www.akpeters.com/gpupro for examples).

Our algorithm can introduce its own artifacts however. In certain camera and light conditions, small haloes may appear at depth discontinuities, as shown in Figure 7.7 (left). In this situation, light incorrectly scatters from the nose to the cheek. This artifact could be seen as the screen-space counterpart of both

[2]See http://www.xyzrgb.com/.

the stretching and seam problems of the texture-space approach. However, these haloes are low frequency artifacts and thus they generally do not hamper the soft appearance of translucent materials. Also, as shown in Figure 7.7 (middle and right), our algorithm is unable to reproduce the scattering produced in high-curvature, thin features like the ears or nose, as we do not have information from the back facing surfaces. In the case of skin, these issues do not seem to damage its general appearance, as has been proven by our previous work [Jimenez et al. 09]. In addition, the usage of very blurry profiles (like the one required for marble) can lead to aliasing problems, as the MSAA resolve is performed before applying subsurface scattering.

Another important difference is that our shader uses pre-scatter texturing (which means that we applied subsurface scattering after the full diffuse color is applied), in contrast with the Hable et al. approach [Hable et al. 09] that uses post-scatter texturing because the usage of a small irradiance map would excessively blur the high frequency details of the diffuse map. We believe that what looks best is a matter of subjective preference.

From an ease-of-use perspective, our algorithm offers various advantages. Being a post-process, integrating our technique into an existing pipeline will not require major changes. It is composed of two short shaders that can be encapsulated in a small class. Additionally, it does not require special UV unwrapping nor does it require dealing with seams, which can account for significant artist time. As we have described, there is no need to take care of special situations such as the objects being out of the viewport, or too far away from the camera, etc., as they are implicitly taken into account by the algorithm.

7.6 Conclusion

We have presented an efficient algorithm capable of rendering realistic subsurface scattering in screen space. This approach reduces buffer management and artist effort. It offers similar performance to the Hable et al. approach [Hable et al. 09] when the object is at moderate distances and scales better as the number of objects increases. Our method generalizes better to other materials. At close-ups it does lose some performance in exchange for quality, but it is able to maintain the *fleshiness* of the original d'Eon approach [d'Eon and Luebke 07]. However, in such close-ups, there is a good chance that the player will be focusing closely on the character's face, so we believe it is worth spending the extra resources in order to render the character's skin with the best quality possible. We believe that our subsurface scattering algorithm has a very simple implementation, has few requirements, and makes a nice balance between performance, generality, and quality.

7.7 Acknowledgments

We would like to thank Luisa García for her endless support, Christopher Oat for his very detailed review and XYZRGB Inc. for the high-quality head scans. This research has been partially funded by the Spanish Ministry of Science and Technology (TIN2007-63025) and the Gobierno de Aragón (OTRI 2009/0411). Jorge Jimenez was funded by a research grant from the Instituto de Investigación en Ingeniería de Aragón.

Bibliography

[d'Eon and Luebke 07] Eugene d'Eon and David Luebke. "Advanced Techniques for Realistic Real-Time Skin Rendering." In *GPU Gems 3*, edited by Hubert Nguyen, Chapter 14, pp. 293–347. Addison Wesley, 2007.

[Gillham 06] David Gillham. "Real-time Depth-of-Field Implemented with a Postprocessing-Only Technique." In *ShaderX5*, edited by Wolfgang Engel, Chapter 3.1, pp. 163–175. Charles River Media, 2006.

[Gritz and d'Eon 07] Larry Gritz and Eugene d'Eon. "The Importance of Being Linear." In *GPU Gems 3*, edited by Hubert Nguyen, Chapter 24, pp. 529–542. Addison Wesley, 2007.

[Hable et al. 09] John Hable, George Borshukov, and Jim Hejl. "Fast Skin Shading." In *ShaderX7*, edited by Wolfgang Engel, Chapter 2.4, pp. 161–173. Charles River Media, 2009.

[Jensen et al. 01] Henrik Wann Jensen, Steve Marschner, Marc Levoy, and Pat Hanrahan. "A Practical Model for Subsurface Light Transport." In *Proceedings of ACM SIGGRAPH 2001*, pp. 511–518, 2001.

[Jimenez and Gutierrez 08] Jorge Jimenez and Diego Gutierrez. "Faster Rendering of Human Skin." In *CEIG*, pp. 21–28, 2008.

[Jimenez et al. 09] Jorge Jimenez, Veronica Sundstedt, and Diego Gutierrez. "Screen-Space Perceptual Rendering of Human Skin." *ACM Transactions on Applied Perception*. (to appear).