Chapter 4

Rendering Gooey Materials with Multiple Layers

Chris Oat[6]
ATI Research

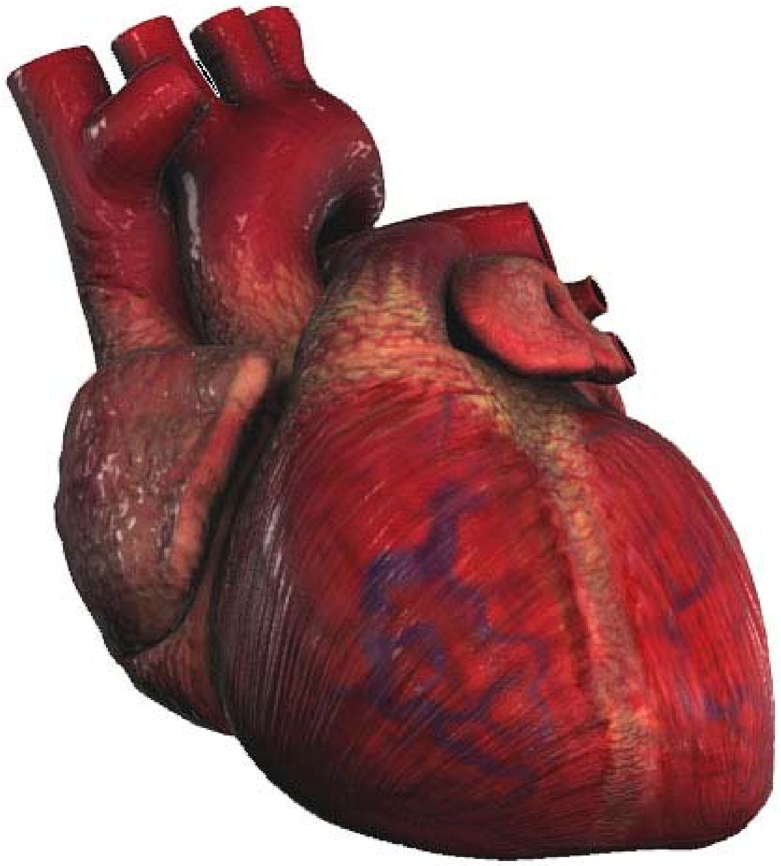


Figure 1. *A human heart rendered in real-time using the multi-layered shading technique described in this chapter.*

[6] coat@ati.com

4.1 Abstract

An efficient method for rendering semi-transparent, multi-layered materials is presented. This method achieves the look of a volumetric material by exploiting several perceptual cues, based on depth and illumination, while combining multiple material layers on the surface of an otherwise non-volumetric, multi-textured surface such as the human heart shown in Figure 1. Multiple implementation strategies are suggested that allow for different trade-offs to be made between visual quality and runtime performance.

4.2 Introduction

Creating immersive and compelling virtual environments requires a keen attention to detail. Over the years, much has been written about achieving photorealism in computer graphics and there is a plethora of information available for those interested in achieving realism through the use of physically based rendering techniques, but physical "correctness" comes at a price and frequently these techniques are difficult to apply or are too slow to evaluate in applications that demand interactive rendering. One area of computer graphics that frequently relies on computationally intensive, physically based rendering is that of volumetric material rendering.

Many real world surfaces, particularly biological materials, exhibit volumetric material properties due to their multiple, semi-transparent layers. Rendering these materials in a physically correct way can be very computationally demanding as it requires simulating complex light interactions due to the tendency of these materials to exhibit subsurface scattering. These physically based techniques are typically too expensive (both in terms of implementation time and render time) for use in applications such as games where "physical correctness" is far less important than rendering efficiency; in games it doesn't matter how nice an image looks if it isn't rendered fast enough to be interactive. We seek to approximate the overall look of volumetric, multi-layered and semi-transparent materials using efficient real-time rendering techniques that can easily be employed by interactive applications.

Instead of finding physically correct ways to simulate the properties of volumetric, multi-layered materials, we instead focus on recreating the important visual aspects of these kinds of materials:

- Inter-layer occlusion (opaque outer layers occlude inner layers)
- Depth parallax (parallax due to layer depth and thickness)
- Light diffusion (light scatters between layers)

We can achieve these properties by combining inexpensive rendering techniques such as: alpha compositing, parallax offsetting, normal mapping and image filtering.

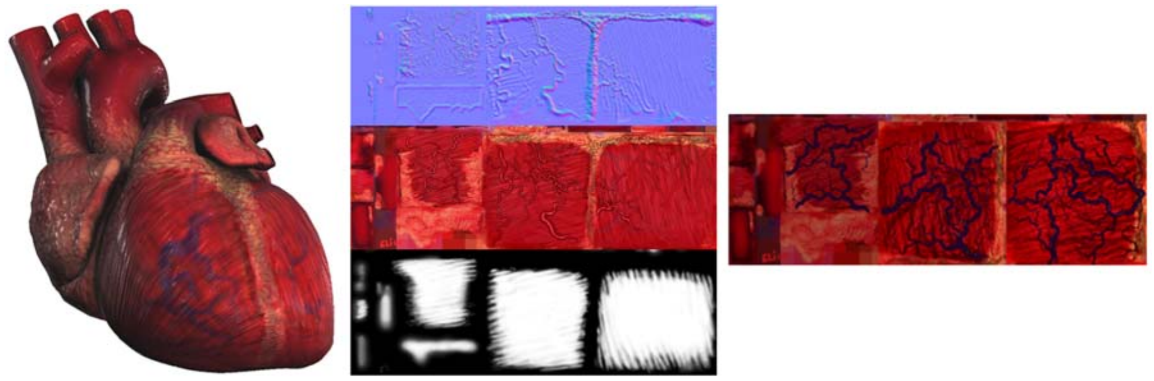


Figure 2. *[Left] Human heart rendered using a two layer material. [Middle] The material's outer layer is composed of a tangent space normal map (top), albedo map (center) and opacity map (bottom). [Right] The material's inner layer uses a single texture to store the inner layer's albedo.*

4.3 Multiple Layers

In computer graphics, volumetric materials are generally approximated with multiple, discrete layers (and often stored using 3D textures). The layers can be semi-transparent and each layer exhibits some amount of light reflectance, transmission, and absorption. We'll be using a two layer material to demonstrate this technique as it is applied to a model of a human heart but in general there's no limit to the number of material layers that may be used.

Our material's two layers are built up from the 2D texture maps shown in Figure 2. The material's outer surface layer is composed of an albedo map, a tangent space normal map for storing the surface detail, and a transparency map that stores the outer layer's varying opacity (more opaque in muscular regions and more transparent in the membrane/gelatinous regions). The material's inner layer requires only a single 2D texture for storing albedo; this layer stores the heart's subsurface vein networks.

In order to get the correct inter-layer occlusion, the outer layer's opacity value is used to blend the two layers together. Layer blending is achieved by performing a simple lerp (linear interpolation) between the two layers based on the outer layer's opacity map. Unfortunately, this simple composite doesn't give us a very volumetric looking material. The result is flat and unrealistic because it doesn't take the material's depth/thickness into account. One inexpensive way to create the impression of layer depth is to use a form of parallax offsetting.

4.4 Depth Parallax

In order to convince the viewer that he or she is looking into a volumetric material, it is important to capture the effects of parallax. Parallax, which is an important perceptual cue for determining distance and depth, is the apparent shift in an object's position relative to other objects due to changes in viewing position [Kaneko01] [Welsh04]. If we use the same texture coordinate to sample and composite the outer and inner layers' textures then the material won't appear volumetric. In order to create the illusion of depth, parallax offsetting is used to adjust the inner layer's texture coordinate so that the inner layer's texture is sampled in a perspective correct way. Figure 3 illustrates how our texture coordinate for sampling from the inner layer's texture must be offset based on both the viewing angle and the distance between the layers.

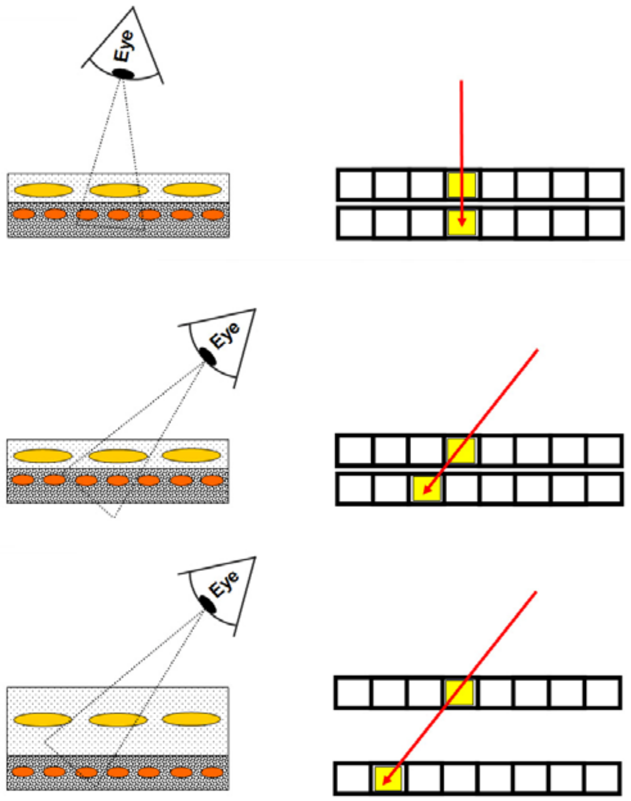
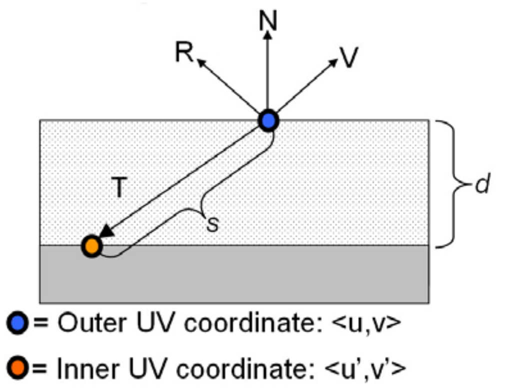


Figure 3. *[Top and Middle] Two different viewpoints of the same material. As the view position changes the sampling location of the inner layer shifts. [Bottom] The shift due to parallax is more pronounced as the thickness of the outer material layer increases.*

A new texture coordinate can be easily computed for the inner layer by making the simplifying assumption that the layers are parallel in tangent space, that is, the distance between the material layers is homogenous. This assumption allows us to avoid performing any kind of ray-tracing in order to find the correct surface point from which to sample the inner layer's texture.



$$\bar{R} = -\bar{V} - 2 * dot(-\bar{V}, \bar{N}) * \bar{N}$$
$$\bar{T} = \langle \bar{R}_x, \bar{R}_y, -\bar{R}_z \rangle$$
$$s = d / |\bar{T}_z|$$
$$\langle u', v' \rangle = \langle u, v \rangle + s \langle \bar{T}_x, \bar{T}_y \rangle$$

Figure 4. *A new texture coordinate is computed for sampling from the inner layer's texture. The inner layer's texture coordinate is computed based on the viewing vector, outer layer's surface normal, and the layer thickness. The distance* ***d*** *between the outer layer and the inner layer is assumed to be homogenous for the entire material and all the vectors are assumed to be unit-length and in tangent space.*

In order to calculate the inner layer's texture coordinate, we must first compute a transmission vector. The transmission vector points from the outer surface layer's sample point to the inner layer's sample point (as shown in Figure 4). We start by computing a reflection vector, which is the view vector reflected about the per-pixel surface normal (sampled from the outer layer's normal map). Since we're working in tangent space, the transmission vector is simply the reflection vector with its z component negated such that it points down into the material. The transmission distance ***s*** (that's the distance along the unit-length transmission vector from the outer layer to the inner layer) is then computed by dividing the layer thickness ***d*** by the z component of the unit-length transmission vector. It's important to note that the transmission distance and layer thickness values are in texture space and therefore are expressed in "texel units" (a texel's width and height are 1/texture width and 1/texture height).

Once we know the transmission vector and the transmission distance, we can find the inner layer's offset texture coordinate by computing the intersection of the transmission vector with the inner surface layer. This new texture coordinate is then used for sampling the inner layer's texture in a perspective correct way, as shown in Figure 5.

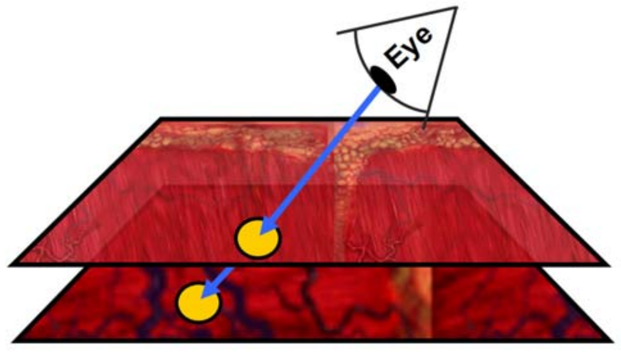


Figure 5. *The parallax offset texture coordinate is used for sampling the inner layer's albedo texture.*

Using a parallax offset texture coordinate to sample the inner layer’s albedo texture will provide a fairly convincing sense of material depth and thickness. An HLSL implementation of the parallax offsetting function is provided in listing 1. This function takes a number of parameters and returns a float3 vector that contains the new texture coordinate for sampling from the inner layer’s texture as well as the transmission distance through the material. The transmission distance will be used later on for computing lighting.

```
// Compute inner layer’s texture coordinate and transmission depth
// vTexCoord: Outer layer’s texture coordinate
// vViewTS: View vector in tangent space
// vNormalTS: Normal in tangent space (sampled normal map)
// fLayerThickness: Distance from outer layer to inner layer
float3 ParallaxOffsetAndDepth ( float2 vTexCoord, float3 vViewTS,
                                float3 vNormalTS, float fLayerThickness )
{
       // Tangent space reflection vector
       float3 vReflectionTS = reflect( -vViewTS, vNormalTS );

       // Tangent space transmission vector (reflect about surface plane)
       float3 vTransTS = float3( vReflectionTS.xy, -vReflectionTS.z );

       // Distance along transmission vector to intersect inner layer
       float fTransDist = fLayerThickness / abs(vTransTS.z);

       // Texel size: Hard coded for 1024x1024 texture size
       float2 vTexelSize = float2( 1.0/1024.0, 1.0/1024.0 );

       // Inner layer’s texture coordinate due to parallax
       float2 vOffset = vTexelSize * fTransDist * vTransTS.xy;
       float2 vOffsetTexCoord = vTexCoord + vOffset;

       // Return offset texture coordinate in xy and transmission dist in z
       return float3( vOffsetTexCoord, fTransDist );
}
```

Listing 1. *An example HLSL implementation of the Parallax offsetting code. Given a texture coordinate on the outer surface layer, a tangent space view and normal vectors, and a layer thickness parameter, this function returns the transmission distance and offset texture coordinates for sampling from the inner layer’s texture map.*

4.5 Light Scattering

In addition to parallax and perspective, a material’s reflectance properties provide important visual cues about the material too. In order to achieve a convincing volumetric look, it is important to approximate the effects of light scattering as it passes through our volumetric, multi-layered material. Once again, instead of focusing on physically correct methods – for light transport through scattering media – we simply focus our efforts on recreating the observable results of this complex behavior.

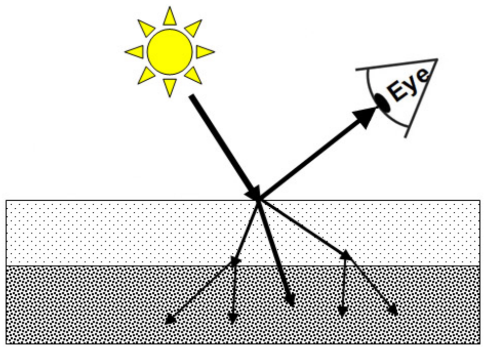


Figure 6. *Light reaches the surface of a material. Some of the light is reflected directly back to the viewer and some of the light is scattered into the material. Since the light that is scattered into the material arrives at inner material layers from many different (scattered) directions, the inner layer will appear more evenly lit (almost as if it were lit by many small light sources from many different directions)*

Figure 6 provides a simplified view of a material that scatters incoming light. In this diagram, some of the light that reaches the material's surface is directly reflected back to the viewer and the rest of the light is scattered down into the material to its subsurface layer.

The outer surface layer's illumination can be approximated using a standard local diffuse lighting model (using a surface normal sampled from the outer layer's normal map to compute N·L). This local lighting term accounts for the light that reaches the surface and is reflected back to the viewer directly (i.e. it doesn't participate in any subsurface scattering). The light that isn't directly reflected back to the viewer is transmitted into the material. This transmitted light may be scattered multiple times before it reaches the material's inner layer and thus will arrive at the inner layer from many different directions (not necessarily the same direction from which it originally arrived at the surface). The over-all effect of this scattering is that the material's inner layer appears more evenly lit – as if it were lit by many small light sources – since it's lit from many different directions. We can achieve this low-frequency lighting effect by rendering the surface layer's diffuse lighting to an off-screen render target and applying a blur kernel to filter out the high-frequency lighting details.

Texture space lighting techniques have been shown to effectively simulate the effects of subsurface scattering in human skin for both offline and real-time applications [Borshukov03] [Sander04]. We use this same basic technique to simulate light scattering as it enters our multi-layered material. First we render our mesh into an off-screen buffer using the mesh's texture coordinates as projected vertex positions (this allows the rasterizer to un-wrap our mesh into texture space as shown in Figure 7).

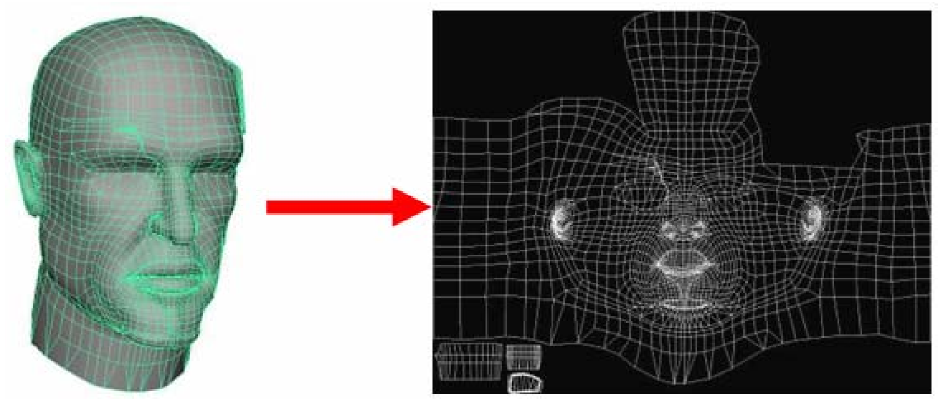


Figure 7. *A mesh is rendered into an off-screen buffer using its texture coordinates as position [Gosselin04].*

The mesh is shaded using a local diffuse lighting model using its 3D world space positions but is rendered into texture space by scaling its texture coordinates such that they're in the correct NDC screen space. The resulting image can be thought of as a dynamic light map for the outer layer of our multi-layered material. We can then use a blurred version of this light map for our material's inner layer (see Figure 8). This gives us smooth, diffused lighting on the material's inner layer. The degree to which the light map needs to be blurred depends on the thickness and scattering properties of the outer layer (the more the outer layer scatters light, the more we need to blur the light map).

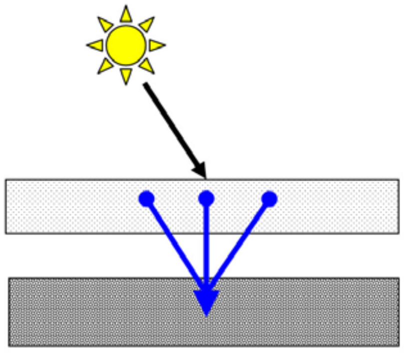


Figure 8. *The average of multiple samples taken from the outer layer's light map is used to light the inner layer. By blurring the outer layer's light map and applying it to the inner layer, we get a nice low-frequency lighting term for the inner layer which creates the appearance of subsurface scattering.*

A resizable Poisson disc filter kernel is ideal for performing the light map blurring since it can be resized dynamically based on the amount of light scattering we wish to simulate. This blur kernel takes a fixed number of taps, where the taps are distributed randomly on a unit disc following a Poisson distribution, from the source texture (see Figure 9). The kernel's size (or the disc's radius, if you prefer) can be scaled on a per-pixel basis to provide more or less blurring as needed. Our kernel's size is proportional to the outer layer's thickness; a thicker outer layer will result in a larger kernel and thus more blurring/scattering.

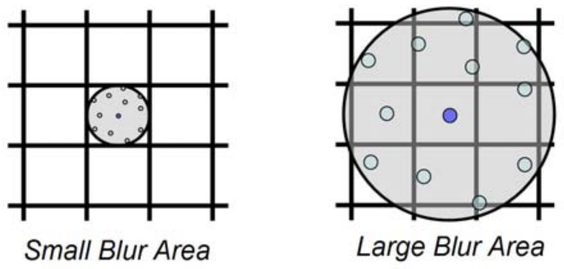


Figure 9. *A resizable Poisson disc filter kernel is useful in applications where the amount of image blurring is dynamic. The individual tap locations are fixed with respect to the disc but the disc itself can be resized to achieve varying amounts of image blurring.*

This blur kernel is straightforward to implement as an HLSL function (listing 2). The individual tap locations are pre-computed for a unit disc and are hard-coded in the function. This function can then be used to compute a dynamic light map for our material's inner layer.

```
// Growable Poisson disc (13 samples)
// tSource: Source texture sampler
// vTexCoord: Texture space location of disc’s center
// fRadius: Radius if kernel (in texel units)
float3 PoissonFilter ( sampler tSource, float2 vTexCoord, float fRadius )
{
   // Hard-coded texel size: Assumes 1024x1024 source texture
   float2 vTexelSize = float2( 1.0/1024.0, 1.0/1024.0 );

   // Tap locations on unit disc
   float2 vTaps[12] = {float2(-0.326212,-0.40581),float2(-0.840144,-0.07358),
                       float2(-0.695914,0.457137),float2(-0.203345,0.620716),
                       float2(0.96234,-0.194983), float2(0.473434,-0.480026),
                       float2(0.519456,0.767022), float2(0.185461,-0.893124),
                       float2(0.507431,0.064425), float2(0.89642,0.412458),
                       float2(-0.32194,-0.932615),float2(-0.791559,-0.59771)};

   // Take a sample at the disc’s center
   float3 cSampleAccum = tex2D( tSource, vTexCoord );

   // Take 12 samples in disc
   for ( int nTapIndex = 0; nTapIndex < 12; nTapIndex++ )
   {
      // Compute new texture coord inside disc
      float2 vTapCoord = vTexCoord + vTexelSize * vTaps[nTapIndex] * fRadius;

      // Accumulate samples
      cSampleAccum += tex2D( tSource, vTapCoord );
   }

   return cSampleAccum / 13.0; // Return average
}
```

Listing 2. *An HLSL function that implements a resizable Poisson disc filter kernel. The source texture's texel size has been hard-coded but could also be passed as an additional argument to the function.*

The filtered light map approximates light scattering as it enters the volumetric material but we must also account for light scattering on its way back out of the material (Figure 10).

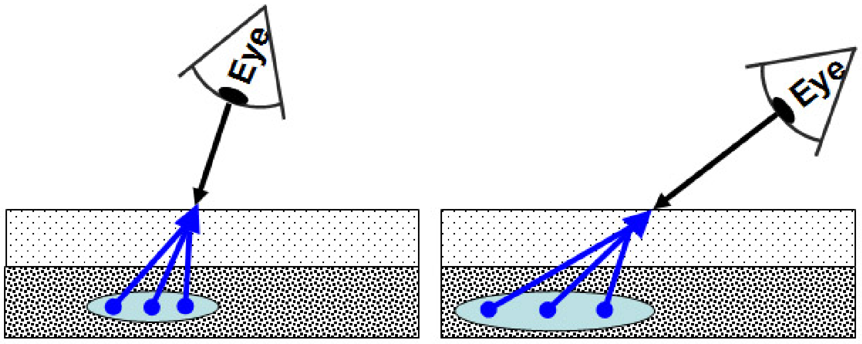


Figure 10. *The Poisson disc sampling function is used for sampling the inner layer's albedo texture to approximate light that is reflected back to the surface, possibly scattering multiple times before it re-emerges at the point where the viewer is looking. The kernel is centered about the parallax offset texture coordinate and its size is a function of the amount of material the viewer is looking through (transmission distance).*

As light scatters down into subsurface material layers, some of the light is reflected back by the material's inner layer. Just as that light scattered as it entered the material, the light also scatters on its way back out. The net result of this additional scattering is that the material's inner layer appears blurry. The deeper the inner layer is from the surface and the more edge-on the viewing angle is, the more material you have to look through to see the inner layer and thus the more blurry it should appear (it's had to scatter through more "stuff" to make it back to your eye). In order to achieve this effect, we use our Poisson disc filtering function again but this time we use it to sample from the inner layer's albedo map. The kernel should be centered about the parallax offset texture coordinate to get a perspective correct sampling location and the kernel's radius should be scaled depending on the transmission distance through the material (this distance is also computed by the parallax offset function from listing 1).

4.6 Putting It All Together

Using the above techniques, we have all the tools necessary to construct a volumetric looking multi-layered material. Our final task is to combine the various techniques into a single shader that can be applied to our mesh. Listing 3 provides HLSL code that uses all the techniques described above to construct a multi-layered material shader that was used to render the beating heart shown in Figure 11.

```
// Sample from outer layer’s base map and light map textures
float3 cOuterDiffuse = tex2D(tLightMap, i.vTexCoord);
float4 cOuterBase = tex2D(tOuterBaseMap, i.vTexCoord); // Opacity in alpha


// Compute parallax offset texture coordinate for sampling from
// inner layer textures, returns UV coord in X and Y and transmission
// distance in Z
float3 vOffsetAndDepth = ParallaxOffsetAndDepth(i.vTexCoord, vViewTS,
                                                vNormalTS, fLayerThicknes);


// Poisson disc filtering: blurry light map (blur size based on layer
// thickness)
float3 cInnerDiffuse = PoissonFilter( tLightMap,
                                      vOffsetAndDepth.xy,
                                      fLayerThickness );


// Poisson disc filtering: blurry base map (blur size based on
// transmission distance)
float3 cInnerBase = PoissonFilter( tInnerBaseMap,
                                   vOffsetAndDepth.xy,
                                   vOffsetAndDepth.z );


// Compute N.V for additional compositing factor (prefer outer layer
// at grazing angles)
float fNdotV = saturate( dot(vNormalTS, vViewTS) );


// Lerp based on opacity and N.V (N.V prevents artifacts when view
// is very edge on)
float3 cOut = lerp( cOuterBase.rgb * cOuterDiffuse.rgb,
                    cInnerBase.rgb * cInnerDiffuse.rgb,
                    cOuterBase.a * fNdotV );
```

Listing 3: *Parallax offsetting and Poisson disc image filtering are used to implement the final multi-layered material shader. This code snippet assumes that the texture-space lighting was performed in a previous pass, the results of which are available via the tLightMap texture sampler.*

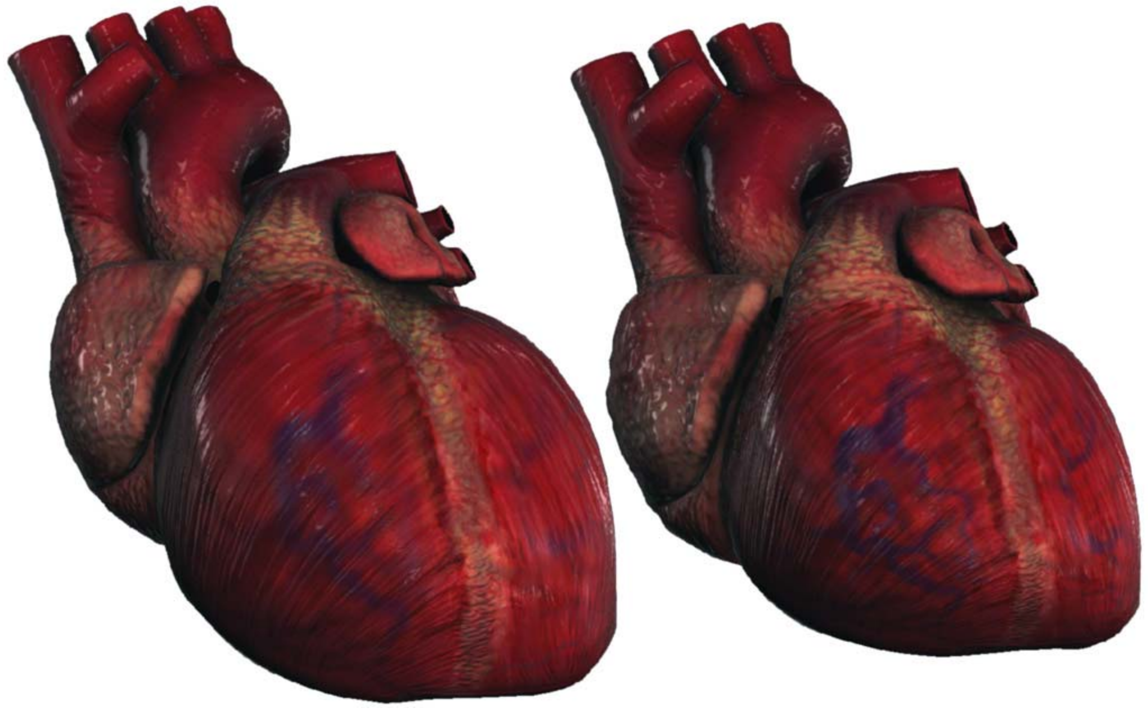


Figure 11. *Two frames of a beating human heart animation that was rendered using the techniques described in this chapter. As the heart expands (left) the distance between material layers is increased which results in a higher degree of depth parallax between layers and a larger blur kernel is used for sampling the dynamic light map and the inner layer's albedo map. When the heart is contracted (right) the layer depth is decreased and the inner material layer is more discernable.*

4.6 Optimizations

The multi-layered material shading technique that's been presented in this chapter has several properties that may make it prohibitively expensive for use in certain rendering scenarios. Because the technique uses a texture-space lighting approach for subsurface scattering, rendering with this technique requires rendering to an off-screen texture which implies making render target changes. Each object in a scene that uses this effect would require its own off-screen render target and thus make batched rendering very difficult if not impossible (this is really only a problem if you plan to apply this effect to many different objects that will be rendered on screen at the same time). In addition to the render target switching overhead, this technique requires significant texturing bandwidth for the multiple texture samples that must be taken from both the outer layer's dynamic light map and the inner layer's albedo map. Two optimizations are suggested that eliminate one or both of these costs by sacrificing some image quality for increased rendering performance.

Our first optimization eliminates the need for a dynamic light map and thus eliminates the need to create and render to an off-screen buffer. The main goal of using the texture space lighting technique to generate a dynamic light map is that it may be used for computing a low-frequency lighting term that may be applied to the material's inner layer.

The dynamic light map essentially allows us to compute lighting for the outer layer once and then re-use these results multiple times when computing the inner layer's lighting term. But we can also achieve a similar effect by taking multiple samples from the outer layer's normal map and computing a bunch of lighting terms on the fly that are then averaged and applied to the inner layer (Figure 12). A new Poisson disc filtering function could be written for sampling from the outer layer's normal map, this new function would compute a diffuse lighting term (N·L) for each tap taken from the normal map and would then average the resulting diffuse lighting terms (we aren't averaging the normals themselves). As before, the filter kernel could be resized depending on the thickness of the various material layers. This optimization entirely eliminates the need for an off-screen renderable texture but still delivers the desired result: high-frequency lighting detail on the outer layer and low-frequency (blurry) lighting on the inner layer.

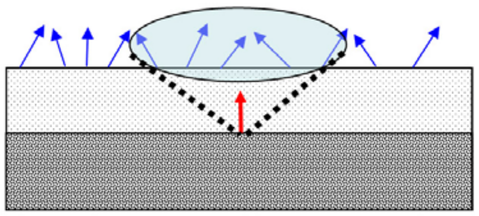


Figure 12. *Multiple samples are taken from the outer layer's normal map (blue vectors) and are used to compute multiple diffuse lighting terms. The resulting diffuse lighting terms are then averaged and used as a final diffuse lighting term for the inner layer. This gives the desired result of a high-frequency outer-layer lighting term and a low-frequency inner-layer lighting term.*

This optimization is not perfect though, it results in a lot of redundant math since diffuse lighting gets computed multiple times for a given normal in the normal map and it does nothing for reducing the amount of texture sampling since we have effectively traded light map sampling for normal map sampling.

In order to make further performance improvements we could eliminate much of the required texture look ups by taking advantage of the mesh's per-vertex normals. Instead of using a filter kernel to achieve low-frequency lighting on the material's inner layer, we instead use the mesh's interpolated vertex normals for computing the inner layer's diffuse lighting term (Figure 13). This drastically reduces the amount of texture bandwidth needed by our shader and gives us more-or-less what we're aiming for; high-frequency lighting detail on the outer layer and smoother, lower-frequency lighting on the inner layer.

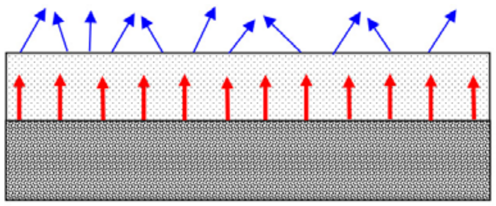


Figure 13. *The outer layer's diffuse lighting is computed using a per-pixel normal (blue) sampled from a normal map for high-frequency lighting detail. The inner layer's diffuse lighting is computed using a per-vertex normal for coarser, lower-frequency lighting.*

4.7 Future Directions

The multi-layered material shader presented in this chapter provides the basic look and feel of a multi-layered, volumetric material but there are many ways in which it might be improved to make it appear more realistic. The example material we've used makes use of two material layers to shade a model of a human heart, but the technique could be applied to materials with many layers by making use of the same kinds of tricks and techniques. Also, the subject of inter-layer shadowing from opaque material layers has been completely ignored. Inter-layer shadowing could potentially be implemented by adjusting the samples taken from the dynamic light map based on the outer layer's opacity. This would keep light from passing through opaque regions of the outer layer and onto inner layer regions that should appear to be in shadow. The shadowing probably becomes more important when you're dealing with many material layers that are very deep/thick since in that scenario it would be possible and even likely that a viewer, looking edge-on into the material, would see below opaque outer regions where one would expect to see inter-layer shadowing occur.

4.8 Conclusion

A method for rendering volumetric, multi-layered materials at interactive frame rates has been presented. Rather than focus on physically correct simulations, this technique exploits various perceptual cues based on viewing perspective and light scattering to create the illusion of volume when combining multiple texture layers to create a multi-layered material. In an attempt to make the technique scalable across a number of different platforms with various rendering performance strengths and weaknesses, two optimizations have been provided that allow an implementation to trade visual quality for performance where appropriate.

4.9 References

BORSHUKOV, G., AND LEWIS, J. P. 2003. Realistic Human Face Rendering for The Matrix Reloaded. In the proceedings of SIGGRAPH 2003, Technical Sketches

GOSSELIN, D. 2004. Real Time Skin Rendering. Game Developer Conference, D3D Tutorial 9

KANEKO, T., TAKAHEI, T., INAMI, M., KAWAKAMI, N., YANAGIDA, Y., MAEDA, T., TACHI, S. 2001. Detailed Shape Representation with Parallax Mapping. In Proceedings of ICAT 2001, pp. 205-208.

SANDER, P.V., GOSSELIN, D., AND MITCHELL, J. L.2004. Real-Time Skin Rendering on Graphics Hardware. In the proceedings of SIGGRAPH 2004, Technical Sketch.

WELSH, T. 2004. Parallax Mapping, ShaderX3: Advanced Rendering with DirectX and OpenGL, Engel, W. Ed., A.K. Peters, 2004