Rendering Semitransparent Layered Media

Christopher Oat
3D Application Research Group
ATI Research

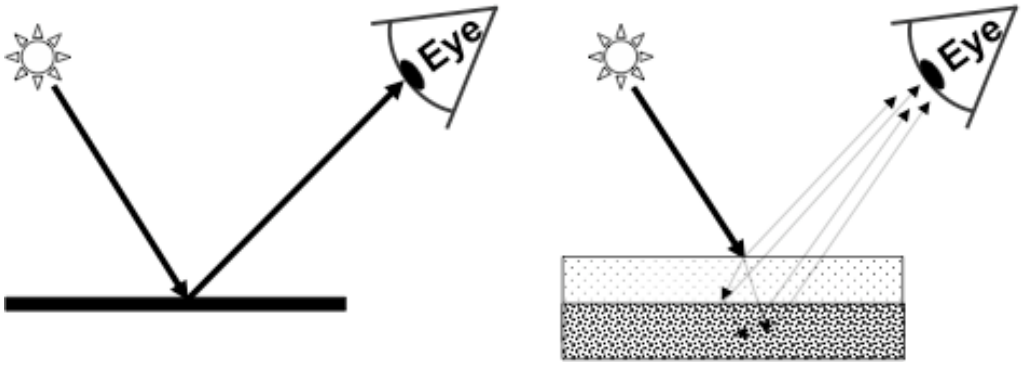

Figure 1. Light is reflected from a surface. (Left) Light is reflected from a simple single layer surface, this is the lighting model used in most games today. (Right) Light enters and scatters, reflecting off many surface layers before reaching the viewer.

Introduction

This article presents a technique for rendering semitransparent, layered media. Modern graphics hardware provides the capability to apply many texture layers onto a single surface. However, simply alpha blending many textures onto a surface does not give the appearance of surface depth that one would expect from semitransparent surface layers. Normal mapping, transparency masking, offset mapping and image filtering are combined here to produce realistic looking layered surfaces. This technique may be used to render multi-layer surfaces with interesting layer interactions.

Background

Many real world surfaces, particularly biological tissues, get their unique appearance from the complex light interactions that occur between surface layers. Figure 1 illustrates the difference in lighting complexity between a simple, single layer surface and a semitransparent, multi-layered surface. The deeper a given light ray travels into a surface the more opportunities there are for scattering, this results in subsurface layers appearing blurry relative to the top most surface layer.

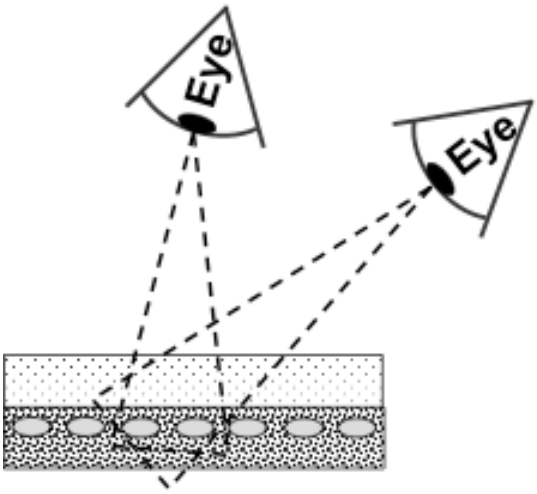

Figure 2. As the view changes, the under-layer will appear to move relative to the top layer.

In addition to appearing blurry, subsurface layers should exhibit parallax. Parallax is the apparent shift in position due to a change in the viewer's line of sight. Figure 2 illustrates two different viewpoints of the same surface. If simple, multi-texture blending was used to render this surface, as the viewpoint changed the under layer would remain fixed relative to the top layer and this would result in the various surface layers appearing flat or squashed. By exploiting the effects of parallax in our shader we can add a sense of depth to the surface layers, this proves to be a very important visual cue that will allow us to render more realistic looking layered surfaces.

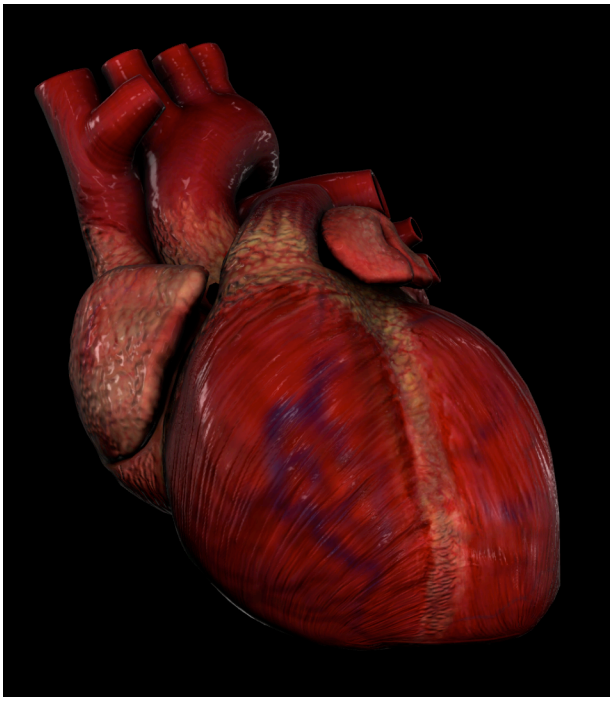

Figure 3. A realistic human heart rendered using multiple semitransparent layers.

Semitransparent Layered Surfaces

The techniques presented in this article have been used to successfully render a realistic human heart as shown in figure 3. This heart model uses two layers to represent the outer most surface and some internal subsurface structure (such as veins and other bits of tissue). A single pass pixel shader is used to compute and blend each layer's contribution to the final heart surface.

Outer Surface Layer

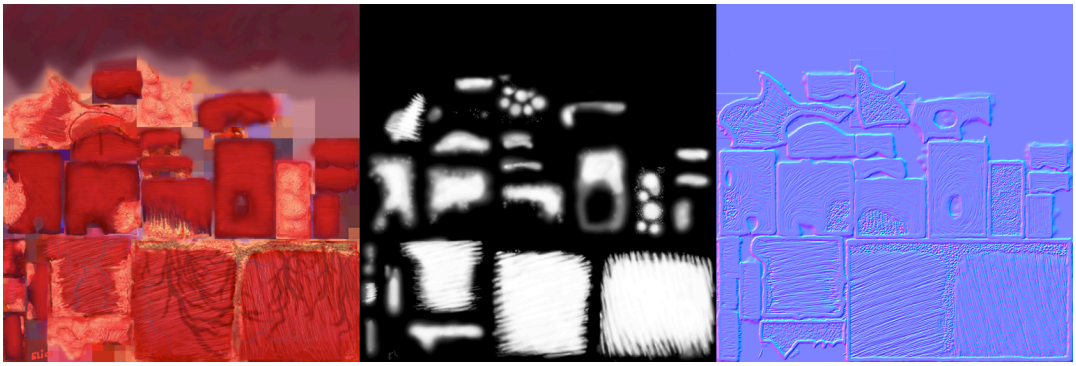

Figure 4. Three textures are used to create the outer most layer of the heart: (Left) Base map, (Center) Transparency map, (Right) Normal map.

The outer most surface layer is composed of a base map, a normal map and a transparency map as shown in figure 4. This layer uses well known rendering techniques to achieve a bumpy, glossy surface. A per-pixel surface normal is sampled from the normal map and used to calculate the diffuse and specular lighting (specular contribution is sampled from a cubic environment map). A transparency map is also used to provide per-pixel transparency values which will be used later to blend the outer and inner layers.

Inner Surface Layer

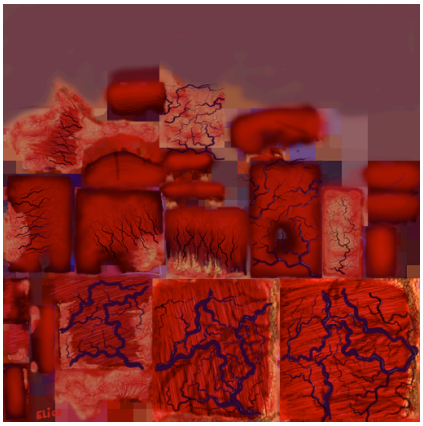

Figure 5. The base map for the heart's inner layer.

The inner layer of the heart is far more interesting than the outer surface layer. This layer only uses a single texture (a base map) as shown in figure 5 but requires slightly more complex pixel shader code to achieve the blurring and parallax effects we desire.

Our first challenge is to calculate the correct texture coordinate from which to sample the inner layer's base map. In order to give the impression of layer depth, a form of parallax mapping is used to offset the outer layer's texture coordinates [Kaneko01] [Welsh04]. These offset coordinates are then used to sample the inner texture layer.

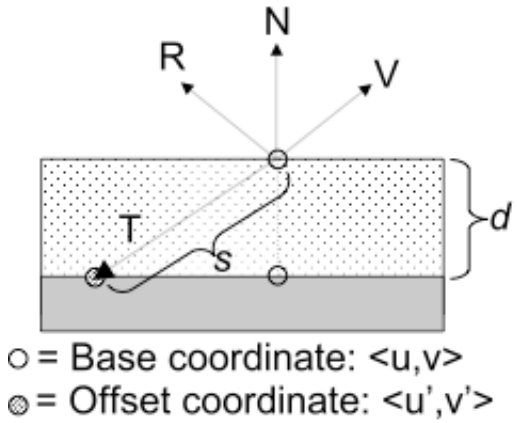

Figure 6. The outer layer's texture coordinate is offset before using it to sample the inner layer's texture.

In order to calculate the offset texture coordinate we must first calculate a transmission vector. The transmission vector points from the outer surface layer's sample point to the inner surface layer's sample point as shown in figure 6. The offset texture coordinate is determined by finding the intersection of the transmission vector with the inner layer. We use a very simple approximation to calculate the transmission vector; the tangent space view vector is reflected about the per-pixel normal (from our normal map) to find a reflection vector. The reflection vector is then reflected about the surface plane such that it points into the surface. Because these vectors are in tangent space, reflecting the reflection vector about the surface plane is simply a matter of negating its z component. The equations in (1) are used to calculate the offset texture coordinate that will be used to sample the inner layer. Unit length vectors are assumed and the distance value should be in "texel units" (<1/texture width, 1/texture height>).

$$\begin{aligned}
\vec{R} &= -\vec{V} - 2 * dot(-\vec{V}, \vec{N}) * \vec{N} \\
\vec{T} &= \langle R_x, R_y, -R_z \rangle \\
s &= d / |T_z| \\
\langle u', v' \rangle &= \langle u, v \rangle + s \langle T_x, T_y \rangle
\end{aligned} \tag{1}$$

We now have an offset texture coordinate that may be used to sample the inner layer's texture map. The next step is to blur the inner layer based on the distance along the transmission vector between the inner and outer sample points. The deeper we see through the top layer, the blurrier the inner layer should appear. In order to achieve the correct amount of blurring, a variable-sized filter kernel is used to sample the inner layer's texture map.

Our filter kernel takes 13 samples, the first sample is centered at the offset texture coordinate position and the remaining 12 samples are taken from nearby texels. Our filter uses stochastic sampling, following a Poisson disc distribution, to determine the 12 "nearby" sample positions. This filter is well suited for our needs because its kernel size can be configured on a per-pixel basis thus allowing us to scale the per-pixel blurriness of the inner layer based on its distance along the view vector from the top layer. HLSL shader code is included below for implementing this filter.

```
//=====================================================================
// Growable Poisson Disc Filter (13 tap)
//
// sampler tSource = source texture being filtered.
// float2 texCoord = texture coordinate for destination texel.
// float2 pixelSize = size of a texel in the source and destination
//                    image. usually this will be a vector like:
//                    <1/width, 1/height>
//
// float radius =     size of kernel in texels (0 will only sample from
//                    texel at texCoordDest, 1.0 will sample from at
//                    most a texel away, etc).
//=====================================================================
float3 GrowableFilterRGB (sampler tSource, float2 texCoord,
                          float2 pixelSize, float radius)
{
   float3 cOut;
   float2 poisson[12] = {float2(-0.326212f, -0.40581f),
                         float2(-0.840144f, -0.07358f),
                         float2(-0.695914f, 0.457137f),
                         float2(-0.203345f, 0.620716f),
                         float2(0.96234f, -0.194983f),
                         float2(0.473434f, -0.480026f),
                         float2(0.519456f, 0.767022f),
                         float2(0.185461f, -0.893124f),
                         float2(0.507431f, 0.064425f),
                         float2(0.89642f, 0.412458f),
                         float2(-0.32194f, -0.932615f),
                         float2(-0.791559f, -0.59771f)};

   // center tap
   cOut = tex2D (tSource, texCoord);


   for (int tap = 0; tap < 12; tap++)
   {
      float2 coord = texCoord.xy + (pixelSize * poisson[tap] * radius);

      // Sample pixel
```

```
      cOut += tex2D (tSource, coord);
   }

   return (cOut / 13.0f);
}
```

We now have a bumpy, glossy outer layer color and a blurry, offset inner layer color. The layers may now be blended using a weighted average based on the per-pixel transparency value and the dot product of the view vector with the surface normal. By adding a view dependant term (N.V) to the weighted average we bias the average towards the outer layer's color at glancing angles where the viewer expects to see more reflection than subsurface diffusion. This also prevents artifacts from cropping up as our texture coordinate offset calculation nears a divide by zero (the z component of the transmission vector approaches zero as the view vector becomes perpendicular to the surface normal).

Pixel Shader

Using these techniques, semitransparent, layered surfaces can be rendered on consumer level programmable graphics hardware. The HLSL code provided below was used to render the multi-layered heart shown in figure 1 and uses the concepts described above. Since the shader code is rather lengthy and all of the interesting math happens at the pixel level, the vertex shader code has not been included here. The vertex shader only computes view and light vectors in tangent space and passes these vectors to the pixel shader.

```
sampler tBaseOuter; // outer layer base map (transparency in alpha)
sampler tBaseInner; // inner layer base map
sampler tBump;      // normal map
sampler tEnv;       // cubic env map

float4  fLayerDepth; // animated layer depth for pulsing heart



struct PsInput
{
   float2 vTexCoord    : TEXCOORD0; // base tex coords
   float3 vInvNormal   : TEXCOORD1; // inverse tangent space matrix
   float3 vInvTangent  : TEXCOORD2;
   float3 vInvBinormal : TEXCOORD3;
   float3 vViewTS      : TEXCOORD4; // tangent space view vec
   float3 vLightTS     : TEXCOORD5; // tangent space light vec
};

float4 main (PsInput i) : COLOR
{

   // Sample outer base, transparency, and normal maps
   float4 cBaseOuter = tex2D(tBaseOuter, i.vTexCoord);
   float3 cNormal = tex2D(tBump, i.vTexCoord);
   float fTransparency = cBase.a;  // transparency in alpha of base

   // Scale and bias normal to convert from [0,1] to [-1,1]
   float3 vNormal = normalize((cNormal * 2.0) - 1.0);
```

```
    // Renormalize interpolated vectors
    float3 vLightTS = normalize(i.vLightTS);
    float3 vViewTS = normalize(i.vViewTS);

    // Recreate the inverse tangent space rotation matrix
    float3x3 mInvTangent = {i.vInvTangent, i.vInvBinormal, i.vInvNormal};

    // Compute tangent space reflection vector then covert to world space
    // for cubemap lookup
    float3 vReflectionTS = reflect(-vViewTS, vNormal);
    float3 vReflectionWS = mul(mInvTangent, vReflectionTS);

    // N.V
    float fNV = saturate(dot(vNormal, vViewTS));

    // Compute diffuse illumination
    float3 cDiffuse = saturate(dot(vNormal, vLightTS));

    // Compute specular illumination (sampled from cubemap)
    float3 cSpecular = texCUBE(tEnv, vReflectionWS);

    // Compute transmission vector
    float3 vTrans = float3(vReflectionTS.xy, -vReflectionTS.z);

    // Distance along transmission vector to intersect inner layer
    // fLayerDepth : distance between layers as heart beats (calculated
    //               on the CPU as sin(time)*4 and in the range [0,4])
    float fTransDistance = fLayerDepth / abs(vTrans.z);

    // Find offset tex coords (inner base map is 512x512)
    float2 vTexelSize = float2(1.0/512.0, 1.0/512.0);
    float2 vOffsetTexCoord = vTexelSize * (fTransDistance * vTrans.xy);
    vOffsetTexCoord = i.vTexCoord + vOffsetTexCoord;

    // Sample inner surface layer with variable size blur filter
    float3 cInnerLayer = GrowableFilterRGB(tBaseInner, vOffsetTexCoord,
                                           vTexelSize, fTransDistance);

    // Final base color is lerp of inner and outer weighted by
    // transparency mask and (N.V)^2
    float3 cBase = lerp(cBaseOuter.rgb, cInnerLayer.rgb,
                        fTransparency * fNV * fNV);

    // Compute final lighting
    float4 o;
    o.rgb = cBase.rgb * cDiffuse;
    o.rgb += cSpecular;
    o.a = 1.0f;

    return o;
}
```

Results and Conclusion

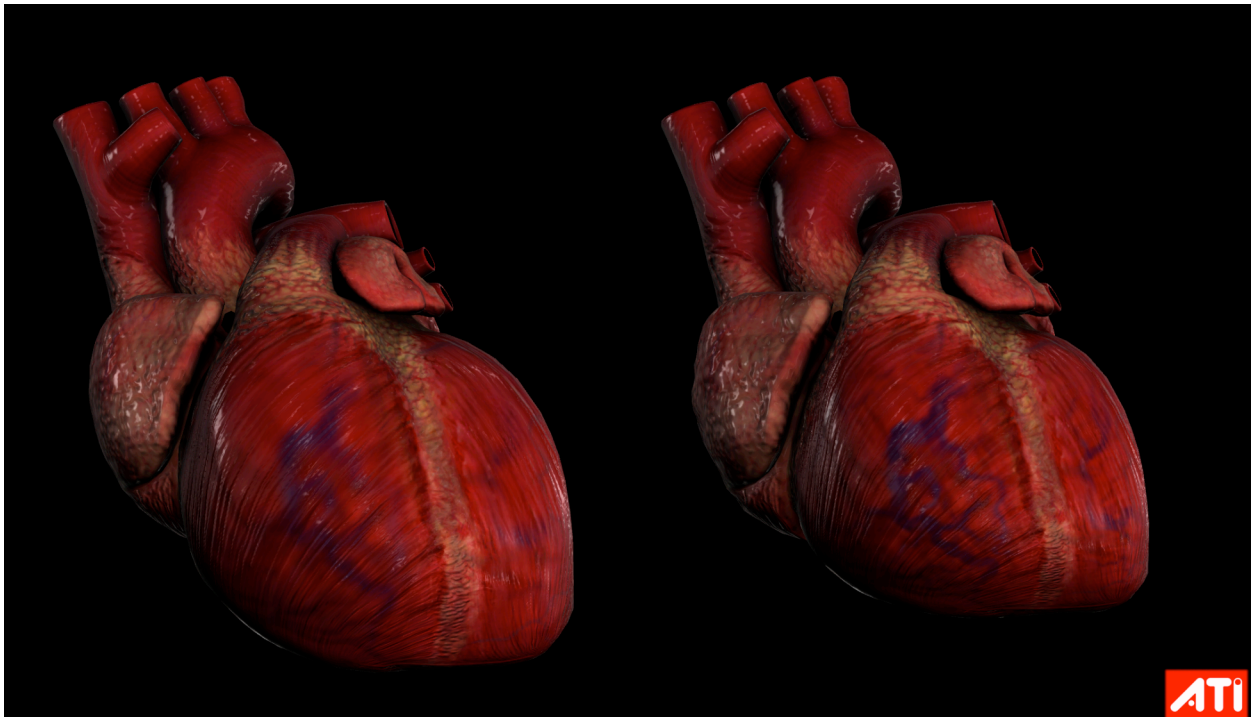

Figure 7. A realistic looking human heart, rendered using the techniques discussed in this article. Two frames of animation are shown here, as the heart contracts the distance between surface layers is reduced and the subsurface structure becomes more visible.

This article presented several tricks for rendering realistic, semitransparent, multi-layered surfaces. Well known rendering techniques such as normal mapping, transparency mapping and environment mapping were combined with a parallax-offset mapping and a variable-size blur filter to render a throbbing human heart with outer and inner surface layers. There are many ways in which the techniques shown here could be extended; notably, relative computational simplicity could be sacrificed to implement more physically correct scattering and reflectance models such as those described in [Pharr02].

References

[Kaneko01] Tomomichi Kaneko, et al. “Detailed Shape Representation with Parallax Mapping,” ICAT, 2001.

[Welsh04] Terry Welsh. “Parallax Mapping with Offset Limiting: A Per-Pixel Approximation of Uneven Surfaces,” 2004.

[Pharr02] Matt Phar. “Layered Media for Subsurface Shaders,” Advanced RenderMan, SIGGRAPH 2002 Course Notes.