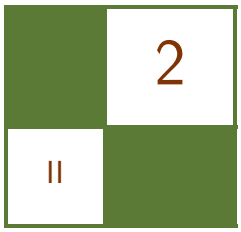


Implementing Fur Using Deferred Shading

Donald Revie

This chapter is concerned with implementing a visually pleasing approximation of fur using deferred shading rather than attempting to create an accurate physical simulation. The techniques presented can also be used to create a number of materials that are traditionally difficult to render in deferred shading.

2.1 Deferred Rendering

For the purposes of this chapter, the term *deferred rendering* can be extended to any one of a group of techniques characterized by the separation of lighting calculations from the rendering of light-receiving objects within the scene, including deferred shading [Valient 07], deferred lighting [Mittring 09], inferred lighting [Kircher 09], and light-prepass rendering [Engel 09]. The fur-rendering technique being presented has been implemented in deferred shading but should be applicable to any rendering solution based on one of these techniques.

This separation of light-receiving objects from light sources is achieved by storing all relevant information about light-receiving objects in the scene as texture data, collectively referred to as a *geometry buffer* or G-buffer because it represents the geometric scene.

When rendering the lights, we can treat the G-buffer as a screen-aligned quad with per-pixel lighting information. Rendering the G-buffer discards all occluded geometry, effectively reducing the three-dimensional scene into a continuous screen-facing surface (Figure 2.1). By using the two-dimensional screen position, depth, and normal information, a pixel shader can reconstruct any visible point in the scene from its corresponding pixel. It is this surface information that is used to calculate lighting values per pixel rather than the original scene geometry.

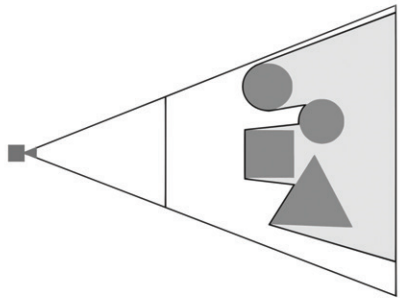


Figure 2.1. G-buffer surface.

In deferred rendering the format of the G-buffer (Figure 2.2) defines a standard interface between all light-receiving materials and all light sources. Each object assigned a light-receiving material writes a uniform set of data into the G-buffer, which is then interpreted by each light source with no direct information regarding the original object. One key advantage to maintaining this interface is that geometric complexity is decoupled from lighting complexity.

This creates a defined pipeline (Figure 2.3) in which we render all geometry to the G-buffer, removing the connection between the geometric data and individual objects, unless we store this information in the G-buffer. We then calculate lighting from all sources in the scene using this information, creating a light-accumulation buffer that again discards all information about individual lights. We can revisit this information in a material pass, rendering individual meshes again and using the screen-space coordinates to identify the area of the light-accumulation buffer and G-buffer representing a specific object. This material phase is required in deferred lighting, inferred lighting, and light pre-pass rendering to complete the lighting process since the G-buffer for these techniques

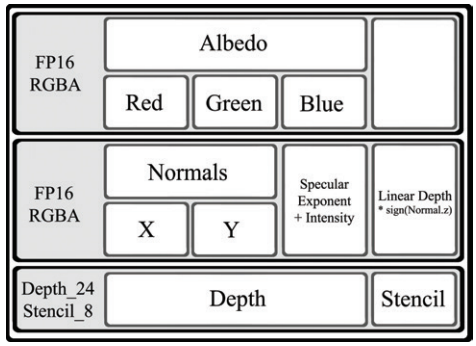


Figure 2.2. Our G-buffer format.

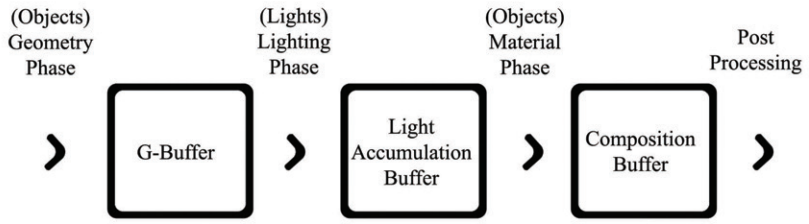


Figure 2.3. General deferred rendering pipeline.

does not include surface color. After this, a post-processing phase acts upon the contents of the composition buffer, again without direct knowledge of individual lights or objects.

This stratification of the deferred rendering pipeline allows for easy extensibility in the combination of different materials and lights. However, adherence to the interfaces involved also imposes strict limitations on the types of materials and lights that can be represented. In particular, deferred rendering solutions have difficulty representing transparent materials, because information regarding surfaces seen through the material would be discarded. Solutions may also struggle with materials that reflect light in a nontypical manner, potentially increasing the complexity of all lighting calculations and the amount of information required within the G-buffer. Choosing the right phases and buffer formats are key to maximizing the power of deferred rendering solutions.

We describe techniques that address the limitations of rendering such materials while continuing to respect the interfaces imposed by deferred rendering. To illustrate these techniques and demonstrate ways in which they might be combined to form complex materials, we outline in detail a solution for implementing fur in deferred shading.

2.2 Fur

Fur has a number of characteristics that make it difficult to represent using the same information format commonly used to represent geometry in deferred rendering solutions.

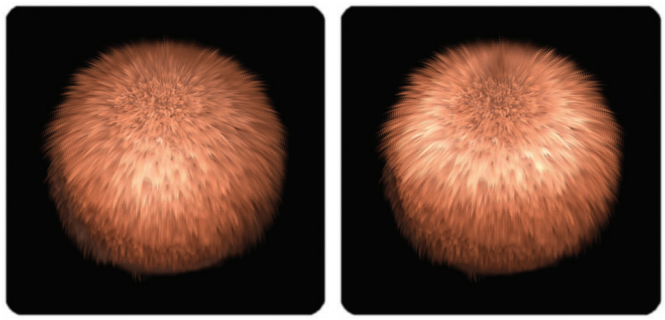
Fur is a structured material composed of many fine strands forming a complex volume rather than a single continuous surface. This structure is far too fine to describe each strand within the G-buffer on current hardware; the resolution required would be prohibitive in both video memory and fragment processing. As this volumetric information cannot be stored in the G-buffer, the fur must be approximated as a continuous surface when receiving light. We achieve this by ensuring that the surface exhibits the observed lighting properties that would normally be created by the structure.

(a) (b)

Figure 2.4. Fur receiving light from behind (a) without scattering and (b) with scattering approximation.

The diffuse nature of fur causes subsurface scattering; light passing into the volume of the fur is reflected and partially absorbed before leaving the medium at a different point. Individual strands are also transparent, allowing light to pass through them. This is often seen as a halo effect; fur is silhouetted against a light source that illuminates the fur layer from within, effectively bending light around the horizon of the surface toward the viewer. This is best seen in fur with a loose, "fluffy" structure (see Figure 2.4).

The often-uniform, directional nature of fur in combination with the structure of individual strands creates a natural grain to the surface being lit. The reflectance properties of the surface are anisotropic, dependent on the grain direction. Anisotropy occurs on surfaces characterized by fine ridges following the grain of the surface, such as brushed metal, and causes light to reflect according to the direction of the grain. This anisotropy is most apparent in fur that is "sleek" with a strong direction and a relatively unbroken surface (see Figure 2.5).



(a) (b)

Figure 2.5. Fur receiving light (a) without anisotropy and (b) with anisotropy approximation.

2.3 Techniques

We look at each of the characteristics of fur separately so that the solutions discussed can be reused to represent other materials that share these characteristics and difficulties when implemented within deferred rendering.

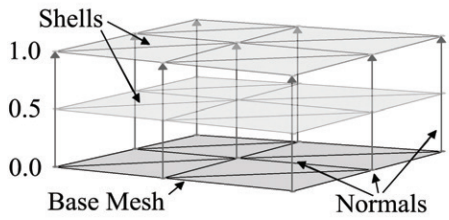


Figure 2.6. Concentric shells.

2.3.1 Volumetric Fur Rendering Using Concentric Shells

It is common to render volumetric structures in real time by rendering discrete slices of volumetric texture data into the scene and using alpha blending to combine the results, such as light interacting with dust particles in the air [Mitchell 04]. Provided enough slices are rendered, the cumulative result gives the appearance of a continuous volume featuring correct perspective, parallax, and occlusion.

The concentric shells method of rendering fur [Lengyel 01] represents the volumetric layer of fur as a series of concentric shells around the base mesh; each shell is a slice through the layer of fur parallel to the surface. These shells are constructed by rendering the base mesh again and pushing the vertices out along the normal of the vertex by a fraction of the fur layer depth; the structure of the fur is represented by a volume texture containing a repeating section of fur (see Figure 2.6). By applying an offset parallel to the mesh surface in addition to the normal we can comb the fur volume (see Figure 2.7, Listing 2.1).

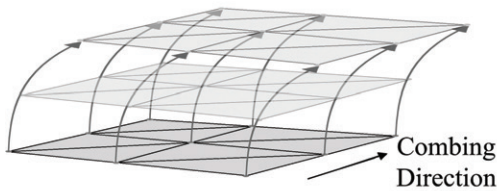


Figure 2.7. Combing.

```
// Get shell depth as normalized distance between base and
// outer surface.
float shellDepth = shellIndex * (1.f/numShells);

// Get offset direction vector
float3 dir = IN.normal.xyz + (IN.direction.xyz * _shellDepth);
dir.xyz = normalize(dir.xyz);

// Offset vertex position along fur direction.
OUT.position = IN.position;
OUT.position.xyz = (dir.xyz * _ shellDepth * furDepth
    * IN.furLength);
OUT.position = mul(worldViewProjection, OUT.position);
```

Listing 2.1. Vertex offsetting.

This method of fur rendering can be further augmented with the addition of *fins*, slices perpendicular to the surface of the mesh, which improve the quality of silhouette edges. However, fin geometry cannot be generated from the base mesh as part of a vertex program and is therefore omitted here (details on generating fin geometry can be found in [Lengyel 01]).

This technique cannot be applied in the geometry phase because the structure of fur is constructed from a large amount of subpixel detail that cannot be stored in the G-buffer where each pixel must contain values for a discrete surface point. Therefore, in deferred shading we must apply the concentric shell method in the material phase, sampling the lighting and color information for each hair from a single point in the light-accumulation buffer. The coordinates for this point can be found by transforming the vertex position of the base mesh into screen space in the same way it was transformed originally in the geometry phase (Listing 2.2).

```
// Vertex shader.
// See (Listing 3.1.1) for omitted content.
// Output screen position of base mesh vertex.
OUT.screenPos = mul(worldViewProjection, IN.position);

// ------------------------------
// Pixel shader.
IN.screenPos /= IN.screenPos.w;

// Bring values into range (0,1) from (-1,1).
float2 screenCoord = (IN.screenPos.xy + 1.f.xx) * 0.5f.xx;

// Sample lit mesh color
color = tex2D(lightAccumulationTexture, screenCoord).
```

Listing 2.2. Sampling lit objects.

This sampling of lighting values can cause an issue specific to rendering the fur. As fur pixels are offset from the surface being sampled, it is possible for the original point to have been occluded by other geometry and thus be missing from the G-buffer. In this case the occluding geometry, rather than the base mesh, would contribute to the coloring of the fur leading to visual artifacts in the fur (Figure 2.8). We explore a solution to this in Sections 2.3.4 and 2.4.4 of this article.

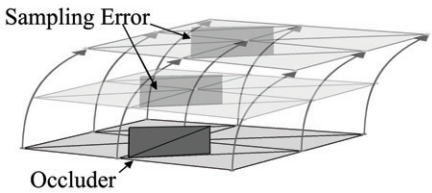


Figure 2.8. Occlusion error.

2.3.2 Subsurface Scattering

Scattering occurs in materials wherein light enters the surface at one point, is transmitted through the medium beneath the surface being reflected and refracted by internal structures and is partially absorbed, before exiting the surface at a different point (Figure 2.9). This light exiting the surface softens the appearance of lighting on the surface by creating a subtle glow.

Much work has been done on the approximation of subsurface scattering properties in skin that is constructed of discrete layers, each with unique reflectance properties. One such solution is to apply a weighted blur to the light accumulated on the surface [Hable 09, Green 04]. In existing forward shaded solutions, this blurring is typically applied in texture space.

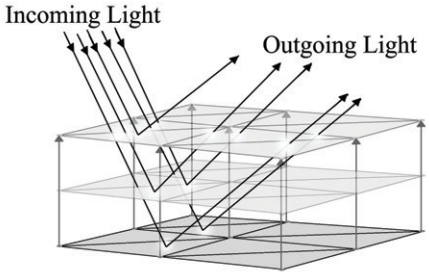


Figure 2.9. Simple subsurface scattering.

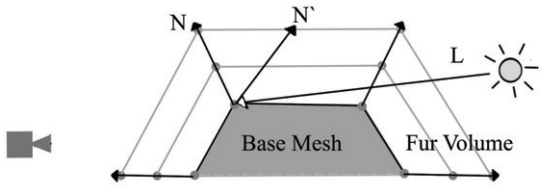


Figure 2.10. Rim glow (N.L < 0) (N‘.L > 0).

In deferred rendering, this technique can be applied in both the geometry phase and the material phase. In the geometry phase the scattering can be approximated by blurring the surface normals written into the G-buffer or by recalculating the mesh normals as a weighted sum of neighboring vertex normals [Patro 07].

Blurring can be performed in the material phase, in texture space, by sampling the accumulated lighting in the same manner as that used for the fur rendering. The texture coordinates of the mesh would then be used as vertex positions to write those values into the mesh's texture space before applying a blur. Once blurred, these values are written back into the light-accumulation buffer by reversing the process. Alternatively, the material-phase blurring could be performed in screen space by orienting the blur kernel to the local surface, using the normals stored in the G-buffer at each pixel.

One issue with this solution is that scattering beneath a surface will also allow light entering the back faces of an object to be transmitted through the medium and exit the surface facing the viewer. In materials such as skin and fur, which form a scattering layer over a more solid structure, this transfer of light appears most often around the silhouette edges of the object. We can adjust for this by bending normals at the silhouette edge of the mesh to point away from the viewer and sample lighting from behind the object (see Figure 2.10 and Listing 2.3). In doing so, these pixels will no longer receive direct lighting correctly; this must then be accounted for during the blur phase (see Sections 2.3.4 and 2.4.4 for details of our solution).

```
// Get normal based for back face samples.
// Glow strength and falloff are supplied by material values.

half NdotV = saturate(dot(normal.xyz, -view);
half rimWeight = glowStrenth * pow(1.f - NdotV), glowFalloff);
normal.xyz += view.xyz * rimWeight;
normal.xyz = normalize(normal.xyz);
```

Listing 2.3. Pushing edge pixels around edges.

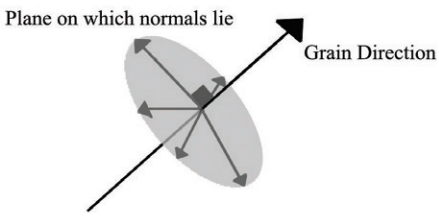


Figure 2.11. Strand normals.

2.3.3 Anisotropy

Anisotropic light reflection occurs on surfaces where the distribution of surface normals is dependent on the surface direction; such surfaces are often characterized by fine ridges running in a uniform direction across the surface, forming a "grain." The individual strands in fur and hair can create a surface that exhibits this type of lighting [Scheuermann 04].

This distinctive lighting is created because in anisotropic surfaces the ridges or, in this case, the strands are considered to be infinitely fine lines running parallel to the grain. These lines do not have a defined surface normal but instead have an infinite number of possible normals radiating out perpendicularly to their direction (see Figure 2.11). Therefore, the lighting calculation at any point on the surface must integrate the lighting for all the normals around the strand. This is not practical in a pixel shader; the best solution is to choose a single normal that best represents the lighting at this point [Wyatt 07].

In forward shading, anisotropy is often implemented using a different lighting calculation from those used to describe other surfaces (Listing 2.4) [Heidrich 98]. This algorithm calculates lighting based on the grain direction of the surface rather than the normal.

```
Diffuse =  sqrt(1 - (< L,T >)^2)
Specular = sqrt(1 - (< L,T >)^2) sqrt(1 - (< V, T > )^2)
    - < L, T >  < V, T >
```

Listing 2.4. Anisotropic light calculation.

In deferred shading we cannot know in the geometry phase the nature of any light sources that might contribute to the lighting on the surface and are bound by the interface of the G-buffer to provide a surface normal. Therefore, we define

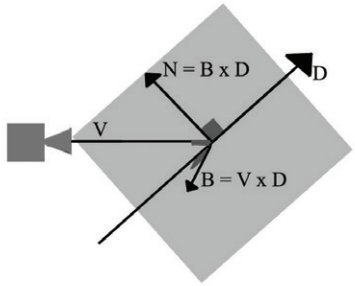
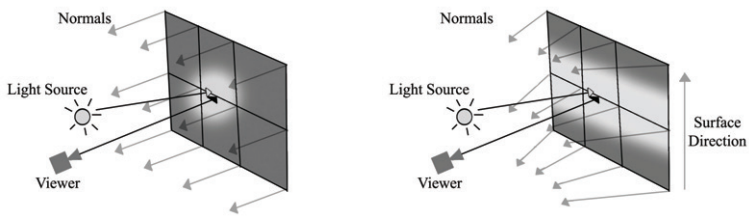


Figure 2.12. Normal as tangent to plane.

the most significant normal as the normal that is coplanar with the grain direction and the eye vector at that point (see Figure 2.12). We calculate the normal of this plane as the cross product of the eye vector and the grain direction, the normal for lighting is then the cross product of the plane's normal and the grain direction (see Listing 2.5).

```
// Generate normal from fur direction.
IN.direction = IN.direction -(dot(IN.direction, normal) * normal);
IN.direction.xyz = normalize(IN.direction.xyz);
half3 binorm = cross(IN.eyeVector, IN.direction);
half3 grainNorm = cross(binorm, IN.direction);
normalize(grainNorm);
```

Listing 2.5. Anisotropic normal calculation.



(a) (b)

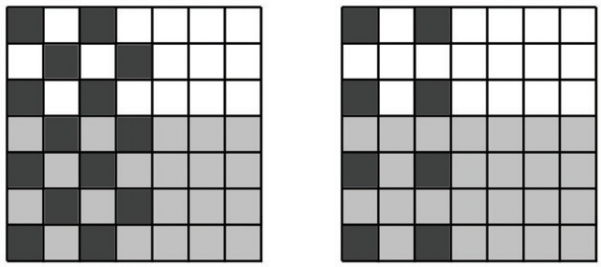
Figure 2.13. (a) Isotropic highlight and (b) anisotropic highlight.

By calculating surface normals in this way we create the effect of curving the surface around the view position, resulting in the lighting being stretched perpendicular to the surface grain (Figure 2.13). While this method does not perfectly emulate the results of the forward-shading solution, it is able to generate

this characteristic of stretched lighting for all light sources, including image-based lights.

2.3.4 Stippled Rendering

Stippled rendering is a technique in which only some pixels of an image are written into the frame buffer, leaving other pixels with their original values. This technique was originally inspired by the stippled alpha transparencies used in games before the widespread availability of hardware alpha blending, also referred to as screen-door transparency [Mulder 98]. The values for the transparent object are written to only some of the pixels covered by the object so as not to completely obscure the scene behind it (see Figure 2.14 and Listing 2.6).



(a) (b)

Figure 2.14. Stipple patterns (a) 1 in 2 and (b) 1 in 4.

```
// Get screen coordinates in range (0, 1).
float2 screenCoord = ((IN.screenPos.xy/IN.screenPos.w)
   + 1.f.xx) * 0.5h.xx;
// Convert coordinates into pixels.
int2 sample = screenCoord.xy * float2(1280.f, 720.f);

// If pixel is not the top left in a 2x2 tile discard it.
int2 tileIndices = int2(sample.x \% 2,  sample.y \% 2);
if((tileIndices.x != 0) || (tileIndices.y != 0))
discard;
```

Listing 2.6. One in four Stipple pattern generation.

In deferred shading, transparent objects are written into the G-buffer using a stipple pattern. During the material phase, the values of pixels containing data for the transparent surface are blended with neighboring pixels containing

information on the scene behind. By varying the density of the stipple pattern, different resolutions of data can be interleaved, allowing for multiple layers of transparency. This technique is similar to various interlaced rendering methods for transparent surfaces [Pangerl 09, Kircher 09].

The concept of stippled rendering can be extended further to blend multiple definitions of a single surface together. By rendering the same mesh multiple times but writing distinctly different data in alternating pixels, we can assign multiple lighting values for each point on the object at a reduced resolution. During the material phase the object is rendered again, and this information is deinterlaced and combined to allow for more complex lighting models. For example, a skin material could write separate values for a subsurface scattering pass and a specular layer, as interleaved samples. The material pass would then additively blend the specular values over the blurred result of the diffuse lighting values.

2.4 Fur Implementation Details

Ease of use and speed of implementation were key considerations when developing the fur solution. We found that to enable artists to easily apply the fur material to meshes, it was important to provide flexibility through a fine degree of control, coupled with real-time feedback. We also wished to ensure minimal changes to existing assets and work methods. It was also important that the technique have minimal impact on our rendering framework, and that it work well with our existing asset-creation pipeline.

To this end, the solution has been implemented with minimal code support; all shader code is implemented within a single effect file with multiple passes for the geometry and material phases of rendering. Annotations provide the renderer with information on when and where to render passes. For real-time feedback, a separate technique is provided within the effect file that renders the fur in a forward-shaded fashion suitable for use within various asset-creation packages.

2.4.1 Asset Preparation

Combing direction. The fur solution is applicable to any closed mesh with per-vertex position, normal, tangent, binormal, and a single set of two-dimensional texture coordinates. This is a fairly standard vertex format for most asset-creation packages.

In addition, we require an RGBA color per vertex to define the combing direction and length of fur at a given vertex (see Figure 2.15). The RGB components encode combing direction as a three-component vector in the object's local space compressing a range of $[-1, 1]$ to $[0, 1]$; this vector is also used to describe the surface direction when generating the anisotropic surface normals. The alpha channel of the color is used to scale the global fur length locally at each vertex.

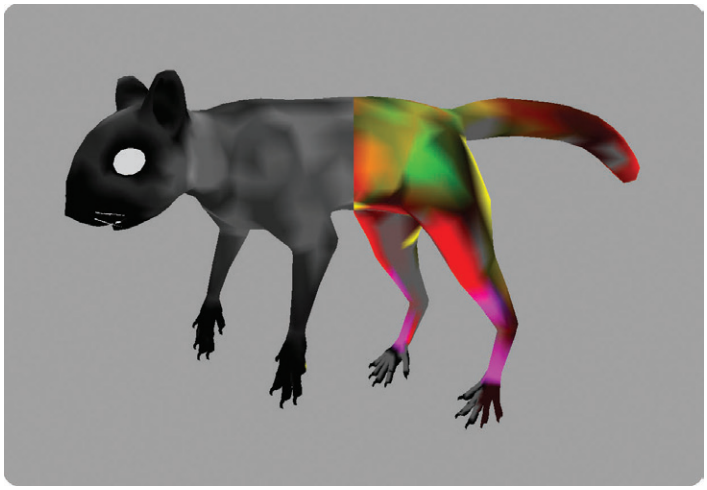


Figure 2.15. Fur length (left) and direction (right) encoded as color.

A color set was chosen to encode this data for a number of reasons. First, many asset-creation tools allow for easy "painting" of vertex colors while viewing the shaded mesh in real time. This effectively allows the author to choose a direction represented as a color and then comb sections of the fur appropriately using the tool, setting an alpha component to the color trims the length of the fur locally. Second, the approach of encoding direction as color is already familiar to most authors through the process of generating world- and tangent-space normal maps. The process has proven to be quite intuitive and easy to use.

As part of the loading process, we transform the vectors encoded in this color channel from the local space of the mesh into its tangent space and at the same time orthonormalize them, making them tangential to the mesh surface. Thus when the mesh is deformed during animation, the combing direction of the fur will remain constant in relation to the surface orientation. This is the only engine side code that was required to fully support the fur-rendering technique (see Listing 2.7).

```
// Build local to tangent space matrix.
Matrix tangentSpace;
tangentSpace.LoadIdentity();
tangentSpace.SetCol(0, tangent);
tangentSpace.SetCol(1, binormal);
tangentSpace.SetCol(2, normal);
tangentSpace.Transpose();

// Convert color into vector.
```

```
Vector3 dir(pColour[0], pColour[1], pColour[2]);
dir = (dir * 2.f) - Vector3(1.f);

// Gram Schmidt orthonormalization.
dir = dir - (dot(dir, normal) * normal); dir.Normalise();

// Transform vector into tangent space.
tangentSpace.TransformInPlace(dir);

// Convert vector into color.
dir = (dir + Vector3(1.f)) * 0.5;
pColour[0] = dir.getX();
pColour[1] = dir.getY();
pColour[2] = dir.getZ();
```

Listing 2.7. Processing of fur directions.

Texture data. To provide the G-buffer with the necessary surface information, the material is assigned an RGB albedo map and a lighting map containing per pixel normal information and specular intensity and exponent at any given pixel. In addition to this, a second albedo map is provided to describe the changes applied to lighting as it passes deeper into the fur; over the length of the strands, the albedo color that is used is blended from this map to the surface color. This gives the author a high degree of control over how the ambient occlusion term is applied to fur across the whole surface, allowing for a greater variation.

To represent the fur volume required for the concentric shell rendering, a heightfield was chosen as an alternative to generating a volumetric data set. While this solution restricts the types of volume that can be described, it requires considerably less texture information to be stored and accessed in order to render the shells. It is more flexible in that it can be sampled using an arbitrary number of slices without the need to composite slices when undersampling the volume, and it is far simpler to generate with general-image authoring tools.

2.4.2 Geometry Phase

The geometry phase is split into two passes for the purpose of this technique. The first pass renders the base mesh to the G-buffer. In the vertex shader the position, tangent, and normal are transformed into view space and the combing direction is brought into local space in the range [-1, 1]. The pixel shader generates a new normal, which is coplanar to the eye and combing vectors to achieve anisotropic highlights (Figure 2.16).

The second pass renders the top layer of the fur in a stipple pattern, rendering to one in every four pixels on screen. The vertex shader is identical to the first pass, but pushes the vertex positions out along the vertex normals offset by the

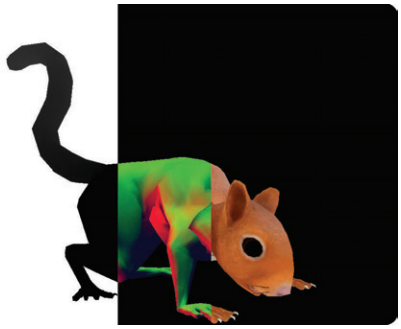


Figure 2.16. Geometry pass 1 (depth/normals/albedo).

global fur length scaled by the vertex color alpha. The pixel shader identifies likely silhouette edges using the dot product of the view vector and the surface normals; the normals at these points are adjusted by adding the view vector scaled by this weight value. The unmodified normals are recalculated to use the anisotropic normals like those of the first pass (Figure 2.17).

This second pass solves the occlusion issue when constructing concentric fur shells from the light-accumulation buffer, since both samples are unlikely to be occluded simultaneously while any part of the strand is still visible. The second pass allows light calculations to be performed for both the surface of the mesh and also the back faces where light entering the reverse faces may be visible.

In order to avoid incorrect results from *screen-space ambient occlusion* (SSAO), edge detection, and similar techniques that rely on discontinuities in the G-buffer, these should be calculated before the second pass since the stipple pattern will create artifacts.

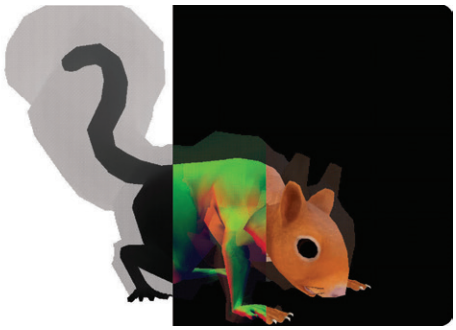


Figure 2.17. Geometry pass 2 (depth/normals/albedo).

2.4.3 Light Phase

During the light phase both the base and stipple samples within the G-buffer receive lighting in the same manner as all other values in the G-buffer, adherence to a common interface allows the fur to receive lighting from a wide range of sources.

Figure 2.18. Light-accumulation buffer.

2.4.4 Material Phase

The material phase of rendering involves reading the values from the light-accumulation buffer and interpreting these based on specific qualities of the material, in this case by constructing shells of fur. In deferred shading, since the majority of the lighting values are already correct in the light-accumulation buffer, a copy of these values is required onto which the material phase of the fur can be composited (see Figure 2.18).

The stipple values, being distributed on the outermost shell of the fur, will occlude the layers of fur beneath. To correct this, all fur surfaces must be rendered again using the outermost shell, while sampling color values from the light-accumulation buffer and depth values from the linear depth stored in the G-buffer (see Figure 2.19). For most pixels, these color and depth values are written directly into the composition buffer, however, where a stipple value would be sampled the neighboring pixel is used instead, effectively erasing all stipple values from the light-accumulation and depth buffers.

The buffer now contains the base mesh of the object only, providing a basis on which to composite the volumetric layer of fur. Rendering of the fur is performed by a series of passes, each pass rendering a concentric shell by offsetting the vertex positions. The pass also constructs positions in screen space, from which both the sample corresponding to the base mesh and the stipple sample corresponding to the outermost shell can be obtained.

Figure 2.19. Stipple obliteration pass.

In the pixel shader these two samples are retrieved from the light-accumulation buffer, their respective linear depths in the G-buffer are also sampled to compare against the depth of the sample coordinates and thus correct for occlusion errors. If both samples are valid, the maximum of the two is chosen to allow for the halo effect of scattering around the edges of the object without darkening edges where there is no back lighting. The contribution of the albedo map to the accumulated light values is removed by division and then reapplied as a linear interpolation of the base and top albedo maps to account for ambient occlusion by the fur. The heightfield for the fur volume is sampled at a high frequency by applying an arbitrary scale to the mesh UVs in the material. The smoothstep function is used to fade out pixels in the current shell as the interpolation factor equals and exceeds the values stored in the heightfield, thus individual strands of fur fade out at different rates, creating the impression of subpixel detail (see Figure 2.20).

Figure 2.20. Shell pass (16 shells).

Figure 2.21. Final image.

2.5 Conclusion

This article has described a series of techniques used to extend the range of materials that can be presented in a deferred rendering environment, particularly a combination of these techniques that can be used to render aesthetically pleasing fur at real-time speeds.

2.6 Acknowledgments

Thanks to everyone at Cohort Studios for showing their support, interest, and enthusiasm during the development of this technique, especially Bruce McNeish and Gordon Bell, without whom there would be no article.

Special thanks to Steve Ions for patiently providing excellent artwork and feedback while this technique was still very much in development, to Baldur Karlsson and Gordon McLean for integrating the fur, helping track down the (often humorous) bugs, and bringing everything to life, and to Shaun Simpson for all the sanity checks.

Bibliography

[Engel 09] W. Engel. "The Light Pre-Pass Renderer." In *ShaderX*^{7}, pp. 655–666. Hingham, MA: Charles River Media, 2009.

[Green 04] S. Green. "Real-Time Approximations to Sub-Surface Scattering." In *GPU Gems*, pp. 263–278. Reading, MA: Addison Wesley, 2004.

[Hable 09] J. Hable, G. Borshakov, and J. Heil. "Fast Skin Shading." In *ShaderX*^{7}, pp. 161–173. Hingham, MA: Charles River Media, 2009.

[Heidrich 98] W. Heidrich and Hans-Peter Seidel. "Efficient Rendering of Anisotropic Surfaces Using Computer Graphics Hardware." In Proceedings of Image and Multi-Dimensional Digital Signal Processing Workshop,Washington, DC: IEEE, 1998.

[Kircher 09] S. Kircher and A Lawrance. "Inferred Lighting: Fast Dynamic Lighting and Shadows for Opaque and Translucent Objects." In *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, Sandbox '09, pp. 39–45. New York: ACM, 2009.

[Lengyel 01] J. Lengyel, E. Praun, A. Finkelstein, and H. Hoppe. "Real-Time Fur Over Arbitrary Surfaces." In *I3D '01 Proceedings of the 2001 Symposium on Interactive 3D Graphics*, pp. 227–232. New York, ACM Press, 2001.

[Mitchell 04] J. Mitchell. "Light Shafts: Rendering Shadows in Participating Media." Game Developers Conference, 2004. Availablwe online at http://developer.amd.com/media/gpu_assets/Mitchell_LightShafts.pdf.

[Mittring 09] M. Mittring. "A Bit More Deferred - CryEngine3." Triangle Game Conference, 2009.

[Mulder 98] J. D. Mulder, F. C. A. Groen, and J. J. van Wijk. "Pixel Masks for Screen-Door Transparency." In *Proceedings of the Conference on Visualization '98*, pp. 351–358. Los Alamitos, CA: IEEE Computer Society Press, 1998.

[Pangerl 09] D. Pangerl. "Deferred Rendering Transparency," In *ShaderX*^{7}, pp. 217–224. Hingham, MA: Charles River Media, 2009.

[Patro 07] R. Patro, "Real-Time Approximate Subsurface Scattering," Available at http://www.cs.umd.edu/\simrob/Documents/sss.pdf, 2007.

[Scheuermann 04] T. Scheuermann, "Hair Rendering and Shading." Game Developer's Conference, 2004.

[Wyatt 07] R. Wyatt, "Custom Shaders and Effects." Available at http://www.insomniacgames.com/research_dev/, 2007.

[Valient 07] M. Valient. "Deferred Rendering in Killzone 2." Develop Conference, July 2007.