Algorithms for Hardware Accelertaed Hair Rendering

Tae-Yong Kim*
tae@rhythm.com

Rhythm & Hues Studio

*formerly at the University of Southern California

In this talk, I will discuss issues related to hair rendering and introduce practical algorithms for hardware accelerated hair rendering. More specifically, I will introduce a simple antialiasing algorithm amenable for hardware accelerated hair drawing, the opacity shadow maps algorithm for hair self-shadow computation, and a programmable shader implementation of Kajyia-Kay hair shading model. All the examples are shown in an OpenGL-fashion, but it should be straightforward to adapt these algorithms to other standard APIs such as Direct3D.

Topics covered are

- Issues in rendering hair with graphics hardware
- A brief overview of self-shadow computation algorithms (shadow buffer, deep shadow maps, and opacity shadow maps)
- Self-shadows generation with graphics hardware (opacity shadow maps)
- Bin sort based visibility ordering for antialiased hair drawing
- Local shading computation with programmable graphics hardware

Additional Materials

1. Tae-Yong Kim and Ulrich Neumann, Opacity Shadow Maps, Eurographics Rendering Workshop 2001 (reprinted in the course note).
2. Tae-Yong Kim, Modeling, Rendering, and Animating Human Hair, Ph. D. Dissertation, University of Southern California, 2002 (available at http://graphics.usc.edu/~taeyong)

1. Introduction

Hair is considered one of the most time-consuming objects to render. There are many reasons why hair rendering becomes such a time-consuming task.

First of all, when rendering hair, we deal with a very complex geometry. The number of hair strands often ranges from 100,000 (for the case of human hair) to some millions (animal fur). Moreover, each hair strand can have geometrically non-trivial shape. For example, let's assume that each hair strand is drawn with 20 triangles. A simple multiplication says that we'd be dealing with a large geometry consisting of several or tens of million triangles! This geometric intricacy complicates any task related to hair rendering.

Second issue is the unique nature of the hair geometry. A hair strand is extremely thin in diameter (~0.1 mm), but can be as long as it grows. This property causes a severe undersampling problem, aliasing. The sampling theorem dictates that the number of samples to reconstruct a signal (in our case, hair geometry) should be higher than the maximum frequency of the signal. Assume that we draw a hair strand as thin triangle strips. According to the sampling theory, the size of a pixel[1] should be smaller than half the thickness of the thinnest hair. In practice, this is equivalent to having an image resolution of 10,000 by 10,000 pixels when the entire screen is approximately covered by somebody's hair. Moreover, when hairs are far away, the required sampling rate should increase! The current display devices hardly reach this limit, and are not likely to reach this limit in the near future. Thus, correct sampling becomes a fundamental issue for any practical hair rendering algorithms.

Third issue is the optical property of hair fibers. A hair fiber not only blocks, but also transmits and scatters the incoming light. As an aggregated form, hairs affect the amount of lighting onto each other. For example, a hair fiber can cast shadows onto other hairs as well as receive lights transmitted through other hairs. Due to the unique geometric shape of hair, the amount of light a hair fiber reflects and scatters varies depending on the relationship between hair growth direction, light direction, and eye position. This effect is known as anistropic reflectance, and defines one of the most prominent characteristics of a hair image (you can easily notice that the direction of the highlight is always perpendicular to the direction of hair growth).

All these issues (number of hairs, sampling issues, and complexity of lighting) make hair rendering a computationally demanding task. In a naïve form, a software renderer[2] (that is not parallelized, and does not utilize any graphics hardware capability) will demand significant computation time. Fortunately, recent progresses in graphics hardware shed some lights. The fastest GPUs at the time of this writing (march, 2003) can now render up to 80 million triangles per second, or 2 ~ 3 million triangles per frame (30 fps). More promisingly, the raw performance of current GPUs increases at a faster rate than that of the general purpose CPUs. So, it seems natural to consider hardware acceleration methods for hair rendering. However, one should

---

[1] A pixel is essentially a point sample. The extent of a sampling region and the pixel sample (color, depth…) should be differentiated. For convenience, we let the size of a pixel denote that of the sampling region.

[2] Here a 'software renderer' refers to a rendering program that is solely dependent on general purpose CPUs. In contrast, a 'hardware renderer' refers to a rendering program that utilizes specialized graphics hardware (such as OpenGL API). In the note, the term 'hardware' will not really mean a dedicated hardware for hair rendering although there is no reason why there can't be such hardware!

keep in mind that most existing graphics cards are not designed for small objects such as hairs. These create a number of difficulties when we use graphics hardware for hair rendering.

2. Tiny, tiny triangles

A hair fiber is naturally represented with a curved cylinder. Thus, it is tempting to draw a hair as some tessellated version of a cylinder (Figure 1).

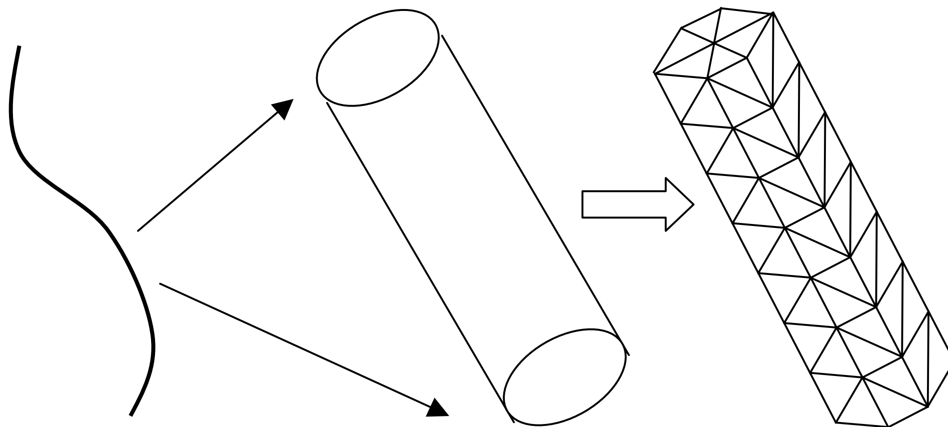


Figure 1. A hair as tessellated cylinder.

This model is totally valid if we were living in a microscopic world where we see just a few hairs in our view. In practice, we deal with so many hairs that this naïve method would generate too many triangles. Moreover, a hair is so thin that the curved shape of the cylinder will be rarely noticeable. Alternatively, we can approximate hair as a flat ribbon that always faces towards the camera (Figure2). In practice, this model approximates hair very well since variation of color along hair's thickness is often ignorable.

We can further simplify the geometry and draw hair as a connected line strips (Figure

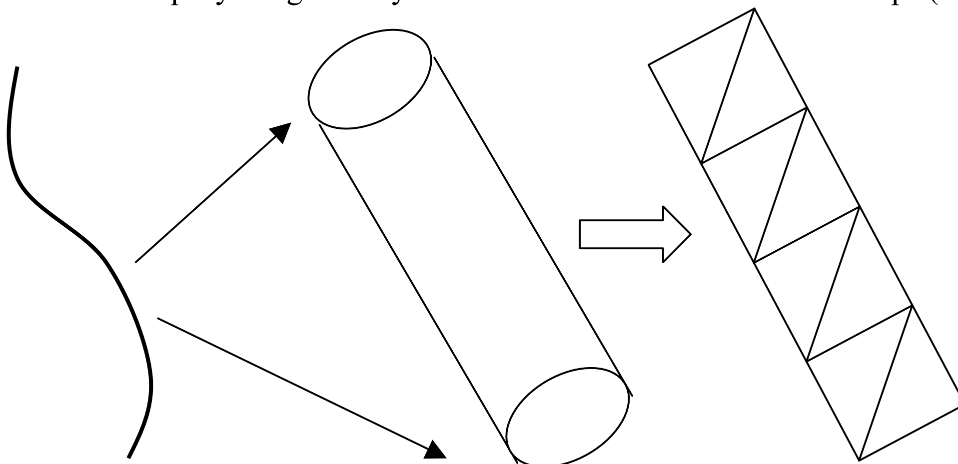


Figure 2. Hair as a flat ribbon.

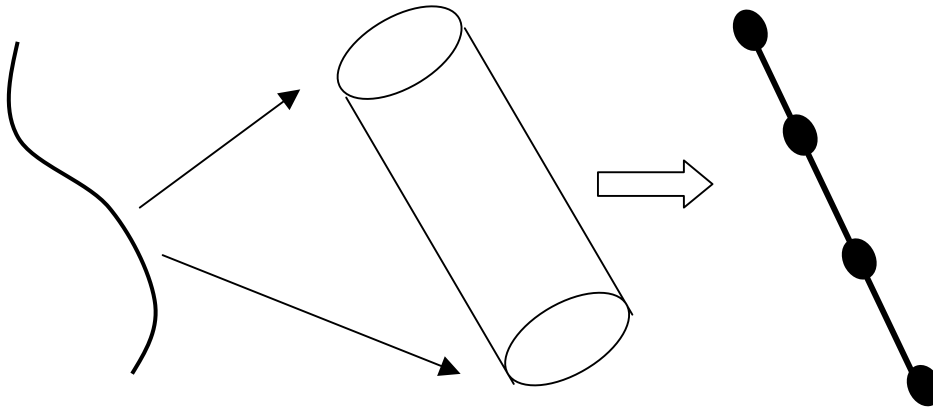


Figure 3. Hair as a line strip.

3). Although mathematically a line should be infinitesimally thin, a line in this case is associated with some artificial thickness value (often a pixel's width).

For the convenience of discussion, I will use the line strip as our hair representation, but discussions and algorithms here equally apply to the polygonal ribbon representation. Let's assume that a hair strand is approximated with a number of points p0, p1,… pn-1 and its associated colors c0,c1,…cn-1. The following code will draw a hair as a connected line strip.

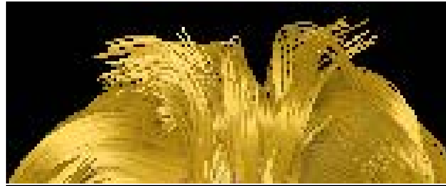
```
DrawHair(p0,p1,..,pn-1,c0,c1,….cn-1)
{
        glBegin(GL_LINE_STRIP)
        glColor3fv(c0);
        glVertex3fv(p0);
        glColor3fv(c1);
        glVertex3fv(p1);
        …
        glColor3fv(cn-1);
        glVertex(pn-1);
        glEnd()
}
```

Routine1. DrawHair

Optimistically, by calling this function repeatedly, you might think that we will able to draw as many hairs as we want. Unfortunately, it is not that simple…

Figure 4. Importance of antialiasing in hair rendering

Without Antialiasing

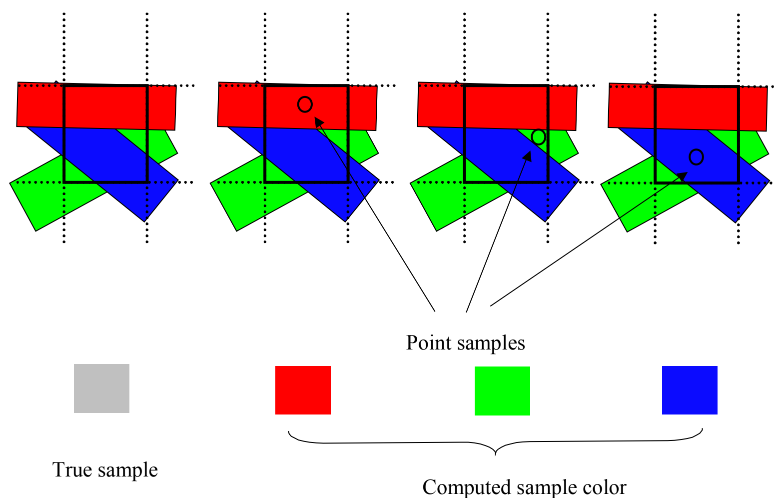


With Antialiasing

When many lines are drawn, the approach will suffer from severe aliasing artifacts as shown in the image above (without antialiasing). Current graphics hardware almost always relies on the Z-buffer algorithm to determine whether a pixel's color should be overwritten. The z-buffer algorithm is a point sampling algorithm. A pixel's color (or depth) is determined entirely by a limited number of point samples (the default setting being just one sample per pixel).

See Figure 5. Assume that three lines cover a pixel and each line's color is red, green, and blue, respectively. If each line covers exactly one third of the pixel's extent, the correct color sample of the pixel should be an averaged color of the three colors - gray. Unfortunately, a single point sample will cause the pixel to change in color to either of the three. So, instead of gray (a true sample), the pixel's color will alternate in red, blue, and green, depending on the point sample's position.

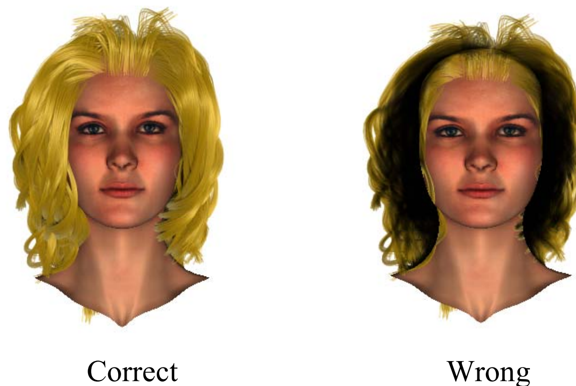
Figure 5. Consequence of point sampling



Now we are aware that the Z-buffer, the most common sampling algorithm in many graphics hardware, is not designed for small objects such as hair. In a point sample-based algorithm such as Z-buffering, the number of point samples determines the quality of the final image. The required number of samples is closely related to the complexity of the scene. The rule of thumb is that there should be at least as many samples as the number of objects that fit in a pixel. That's why we often don't see much aliasing when we draw relatively large triangles, but in hair. There are ways to increase the number of samples. The most common method is the accumulation buffer. In this method, the number of samples per pixel corresponds to the number of accumulation steps performed. However, accumulation buffers tend to be slow in many OpenGL implementations and the accumulation steps must be performed at every frame.

The thickness of a hair is often much smaller than the size of a pixel. So, it seems natural to draw a line with small alpha value and the attempt will prove fine for one line. However, as more lines (hairs) are drawn, we will encounter a similar problem we had before. The alpha blending in OpenGL requires that the scene should be sorted by the distance from the camera. Otherwise, the image will not look right – you will *see through* the pixels (Figure 6).

Figure 6. Alpha blending needs correct visibility ordering



Correct Wrong

In short, we need to 1) sample each hair correctly, 2) draw each hair with the correct thickness, and 3) blend the colors of all the hairs correctly. Many current graphics hardware offer decent, if not perfect, hardware accelerated antialiased line drawing features. To draw each hair with correct sampling, we can exploit the feature. To draw each hair with the correct thickness, we set the alpha value of each line to a small (<1.0) value. To blend the colors correctly, we use the alpha blending with the correct visibility order. Both hardware line antialiasing and alpha blending require a correct visibility order. So, we will do the visibility ordering by ourselves, departing from the troublesome Z-buffering.

2.1 Further Readings

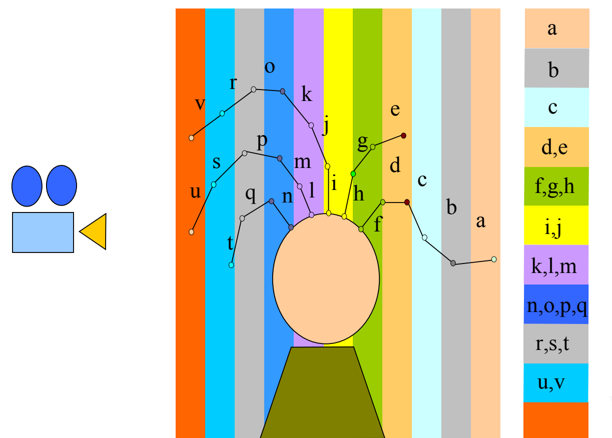
For rigorous discussions on issues with point sampling, refer to the following papers.

- D. Mitchell, Consequences of stratified sampling in graphics, SIGGRAPH Proceedings, 1996, 277-280
- T. Lokovic and E. Veach. Deep shadow maps, SIGGRAPH Proceedings, 2000, 385-392
- A. R. Smith. A Pixel is Not A Little Square, A Pixel is Not A Little Square, A Pixel is Not A Little Square! (And a Voxel is Not a Little Cube), Tech Memo 6, Microsoft, Jul 1995

3. Visibility ordering for antialiased hair drawing

For correct visibility computation, we need to draw hair far to near (or near-to-far as long as it is consistent). Assume that each hair is broken into line segments, and we draw a large number of such line segments for the entire hair model. Antialiasing can be performed in two steps. First, the visibility order of a given hair model is determined based on the distance to the camera (Figure 7). The bounding box of all the segments is sliced with planes perpendicular to the camera. Each *bin*, a volume bounded by a pair of adjacent planes (drawn as a colored bar in Figure 7), stores indices of segments whose farthest end point is contained by the bin. After other objects (e.g., a head mesh) are drawn, the depth buffer update is disabled. Then, the segments are drawn as antialiased lines such that the ones indexed by the farthest bin are drawn first.

Figure 7. Bin-based Visibility Ordering



The end points of line segments are grouped into a set of bins that slice the entire volume (Figure 7). The bin enclosing each point is found by

$$i = \left\lfloor N \frac{D - D_{\min}}{D_{\max} - D_{\min} + \varepsilon} \right\rfloor, 0 \le i < N$$

, where i is the index for the bin and N is the number of bins. D is the distance from the point to the image plane. D_{min} and D_{max} are minimum and maximum of such distances, respectively. ε is a constant such that $\varepsilon < \frac{D_{max} - D_{min}}{N}$.

Given a point $\vec{p}$ and the camera positioned at $\vec{c}$ looking in direction of $\vec{d}$, the distance to the image plane is computed by

$$D = (\vec{p} - \vec{c}) \cdot \vec{d}$$

Precise visibility ordering of lines is difficult to obtain by depth sorting alone. When lines are nearly parallel to the image plane, the errors are small, provided the spacing between bins is dense enough. However, when line segments extend over many bins, the visibility order cannot be determined either by maximum depth or minimum depth. Such lines could be further subdivided. However, on the other hand, the pixel coverage of such a line tends to be small. For example, when a line is perpendicular to the image plane, the pixel coverage of the line is at most a single pixel. In practice, using maximum depth for every line produces good results.

In the drawing pass, each bin is accessed from the farthest to the closest. For each bin, the line segments whose farther points belong to the bin are drawn. Each line segment is drawn with hardware antialiasing. The color of each line segment is accumulated using

glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA)

The bin-based visibility ordering algorithm can be summarized as follows.

Visibility ordering pass:

For every line segment,

1. compute the depth of the end points.
2. using the larger depth, compute the bin location
3. add the index of the line to the bin.

Drawing pass:

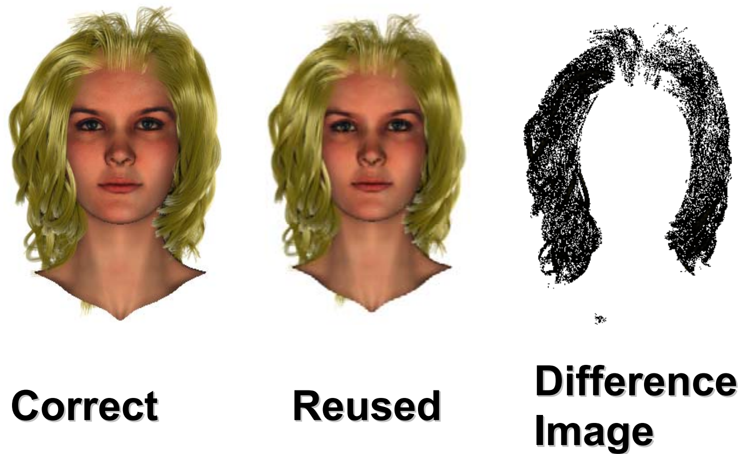
1. Enable the depth buffer
2. Draw all the other scene objects
3. Disable the depth buffer
4. For each bin, from the farthest bin to the nearest,

    Draw all the line segments whose indices are stored in the bin

Note that since the line segments are drawn in a correct order, all the hardware features can be now happily exploited (antialiased line drawing, alpha blending). Although simple, the method is fast and converges to exact ordering as hair strands are drawn with more segments. The visibility-ordering algorithm runs at about 700,000 lines per second on a Pentium III 700Mhz CPU. Another benefit is that we can separate the visibility ordering pass from the actual drawing pass. For example, during interactive modeling, the viewpoint does not change much from frame to frame. This coherence enables performing the visibility ordering periodically and reusing the computed order for subsequent frames. In Figure 8, the image in the

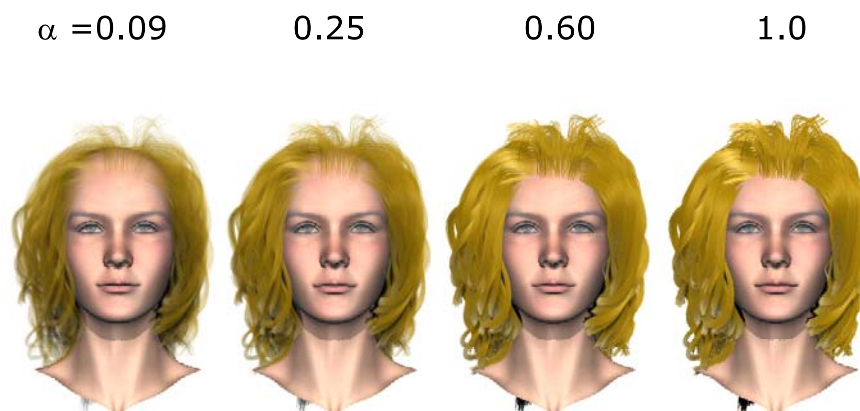
middle was drawn with a previously computed visibility order where the head model was rotated by 30 degree along the y-axis. Although the difference image shows that there's some discrepancy against the correct order (left image), visual degradation is not so objectionable. In contrast, depth buffer based super-sampling methods (e.g., accumulation buffer) must compute visibility at every frame.

Figure 8. Coherence of Visibility Order



In addition, the alpha values of line segments can control the perceived thickness of hair strands. As hair strands become thinner, super-sampling methods would require more samples while alpha value changes suffice in the visibility-ordered hair model (Figure 9).

Figure 9. Thickness Change



4. Self-shadows

Hairs cast shadows onto each other, as well as receive and cast shadows from/to other objects in the scene. Especially, self-shadows create crucial patterns that distinguish one hairstyle from others. Without self-shadows, the underlying structure of a hair model cannot be correctly visualized (Figure10). This section introduces an efficient shadow generation algorithm that makes full use of graphics hardware accelerator.

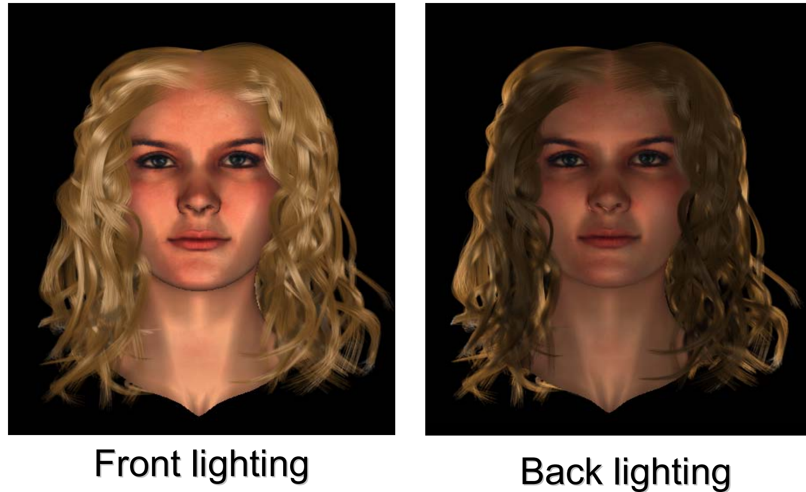
Figure 10. Self-shadows are crucial for volumetric hair

No shadows **With shadows**

There are mainly two issues with self-shadows in hair rendering – the thin geometry of the hair fiber and the translucency. The thin geometry of hair causes serious aliasing artifacts in shadow computation in a very similar way as in the previous section. This is not surprising if we note that the shadow computation is just one instance of the more general visibility computation problem. For hair drawing, we computed the visibility of each hair from the camera. For shadow computation, we need to compute the visibility of each hair from the *light source*. If the hair *sees* more lights, it will receive more illumination from the source, and if the hair can't see the light, it will be left dark (or shadowed).

Another important aspect in hair self-shadowing is that a hair fiber often does not completely block the incoming light. It not only reflects and scatters the incoming illumination, but often lets the light pass through. This unique property of the hair fiber is the most clearly observable in the 'back lighting' situation where the silhouette shines brightly when the light is put behind (Figure 11).

Figure 11. Translucency of hair



Front lighting Back lighting

In the previous section, we discussed the problems in using Z-buffer for hair rendering. When the Z-buffer is used for shadow computation, it is called 'shadow map'. In this *depth-based shadow map* (DBSM), the scene is rendered from the light's point of view and the depth values are stored. Each point to be shadowed is projected to the light's camera and the point's depth is checked against the depth in the shadow map.

One attractive feature of the traditional shadow map is that the shadow map can be generated with hardware by rendering the scene from the light's point of view and storing the resulting depth buffer. However, severe aliasing artifacts can occur with small semi-transparent objects. As discussed in the previous section, in a dense volume made of small primitives, depths can vary radically over small changes in image space. The discrete nature of depth sampling limits DBSM in handling such objects. The binary decision in depth testing inherently precludes any translucency. Thus, DBSM is unsuited for volumetric objects such as hairs.

The transmittance $\tau(p)$ of a light to a point p can be written as

$$\tau(p) = \exp(-\Omega), \text{ where } \Omega = \int_0^l \sigma_t(l')dl' \quad (1)$$

In (1), l is the length of a path from the light to the point, σ_t is the extinction (or a density) function along the path. Ω is the *opacity* value at the point.

The deep shadow maps (DSM) algorithm originally presented at SIGGRAPH 2000 proposed that each pixel stores a piecewise linear approximation of the transmittance function instead of a single depth, yielding more precise shadow computation than DBSM. Deep shadow maps account for the two important properties of hair shadows.

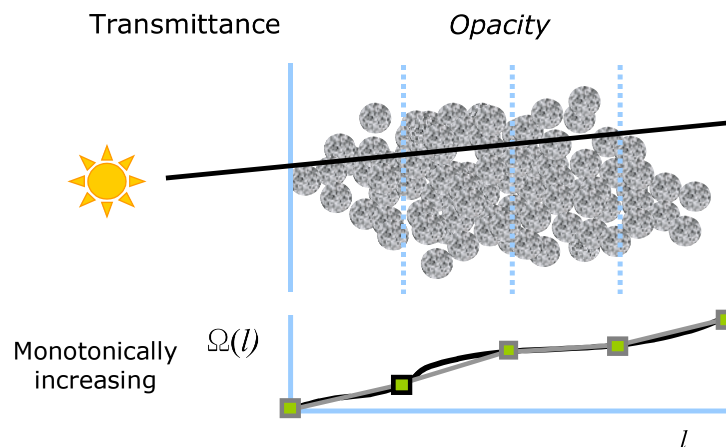
Partial visibility: In the context of shadow maps, the transmittance function can be viewed as a partial visibility function from the light's point of view. If more hairs are seen along the path from the light, the light will be more attenuated (occluded), resulting in less illumination (shadow). As noted earlier (recall Figure 5), visibility can change drastically over the pixel's extent. The transmittance function handles this partial visibility problem by correctly integrating and filtering all the contributions from the underlying geometry.

Translucency: a hair fiber not only reflects, but also scatters and transmits the incoming light. Assuming that the hair fiber transmits the incoming light only in a forward direction, the translucency is also handled by the transmittance function.

Despite the compactness and quality, however, due to the underlying data structure (linked list), a hardware implementation becomes tricky with deep shadow maps.

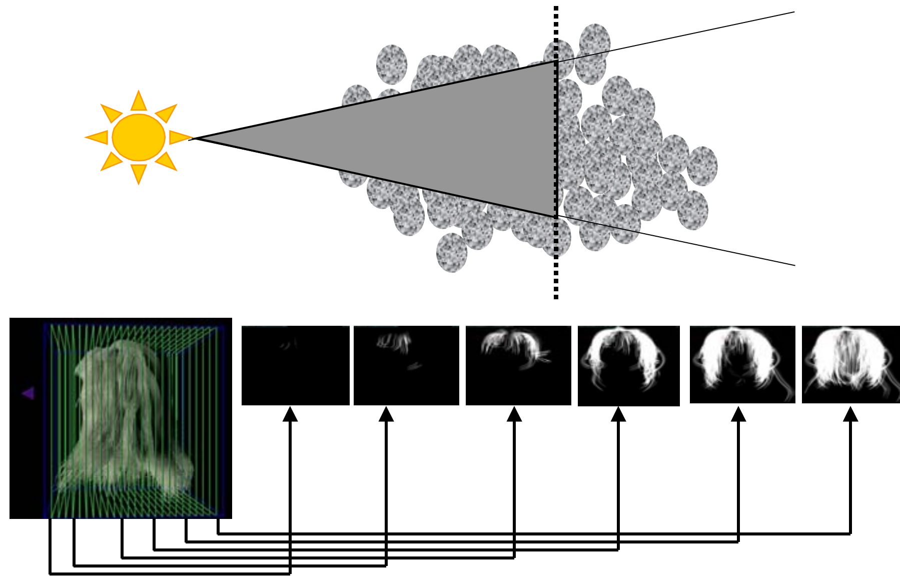
The Opacity Shadow Maps (or OSM) was originally designed as a fast alternative to DSM for computing the transmittance function. Opacity shadow maps (or OSM) algorithm uses a set of parallel opacity maps oriented perpendicular to the light's direction (Figure 12).

Figure 12. Opacity Shadow Maps



By approximating the transmittance function with discrete planar maps, opacity maps can be efficiently generated with graphics hardware. On each opacity map, the hair model is rendered from the light's point of view, clipped by the map's depth (Figure 13). Instead of storing depth values, each pixel stores Ω, the line integral of densities along the path from the light to the pixel. The opacity values from adjacent maps are then sampled and interpolated during rendering.

Figure 13. Opacity Maps



4.1 Basic Algorithm

Opacity shadow maps heavily rely on graphics hardware and operate on any bounded volumes represented by standard primitives such as points, lines and polygons. (In our context, hairs are represented as a cluster of lines.) The hair volume is sliced with a set of opacity map planes perpendicular to the light's direction. The scene is rendered to the alpha buffer, clipped by each map's depth. Each primitive contributes its associated alpha value. The alpha value is a user-controllable parameter that depends on the size (thickness) and the optical property of hair. It also depends on the resolution of the opacity maps. Each pixel in the map stores an alpha value that approximates the opacity relative to the light at the pixel's position. The opacity values of adjacent maps are sampled and linearly interpolated at the position of each shadow computation point, to be used in a shadowed shading calculation.

The pseudo code in Routine 2 uses the following notation. P is the set of all the shadow computation sample points (or simply *shadow samples*). N is the number of maps and M is the number of shadow samples. D_i is the distance from the opacity map plane to the light $(1 \leq i \leq N)$. P_i is a set of shadow samples that reside between D_i and D_{i-1}. p_j is j_{th} shadow sample $(1 \leq j \leq M)$. *Depth*(p) returns a distance from p to the light. $\Omega(p_j)$ stores the opacity at p_j. $\tau(p_j)$ is the transmittance at p_j. B_{prev} and $B_{current}$ are the previous and current opacity map buffers.

Routine 2. Opacity Shadow Maps

```
1.  D_0 = Min (Depth(p_j)) for all p_j in P (1 ≤ j ≤ M)
2.  for (1 ≤ i ≤ N)                                          (Loop 1)
3.      Determine the opacity map's depth D_i from the light
4.  for each shadow sample point p_j in P (1 ≤ j ≤ M)        (Loop 2)
5.      Find i such that D_{i-1} ≤ Depth(p_j) < D_i
6.      Add the point p_j to P_i.
7.  Clear the alpha buffer and the opacity maps B_prev, B_current.
8.  for (1 ≤ i ≤ N)                                          (Loop 3)
9.      Swap B_prev and B_current.
10.     Render the scene clipping it with D_{i-1} and D_i.
11.     Read back the alpha buffer to B_current.
12.     for each shadow sample point p_k in P_i              (Loop 4)
13.         Ω_prev = sample(B_prev , p_k)
14.         Ω_current = sample(B_current , p_k)
15.         Ω = interpolate (Depth(p_k), D_{i-1}, D_i, Ω_prev, Ω_current)
16.         τ(p_k) = e^{-κΩ}
```

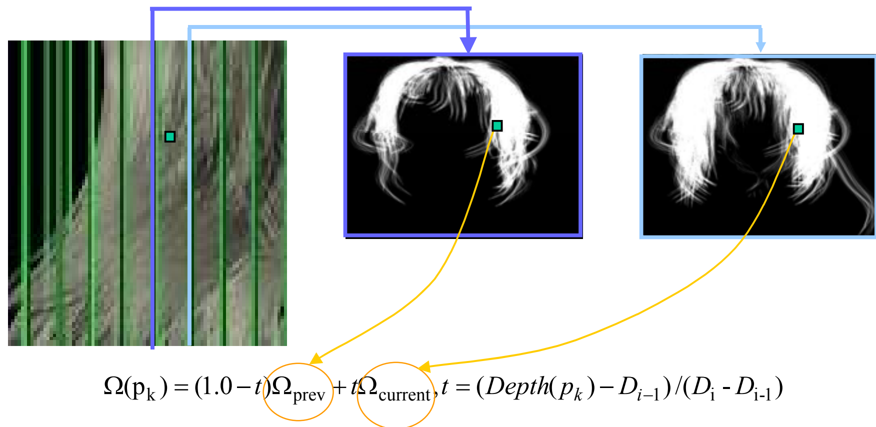
In loop 1, the depth of each map is determined. Uniform slice spacing is reasonable for evenly distributed volumes. Prior to shadow computation, shadow samples are prepared. In our hair representation, the primitives (line segments) tend to be very small and thus the end points of lines often suffice. Thus, for each hair strand, we choose the endpoints of line segments as shadow samples. More samples can be taken if needed. When many samples are required for each primitive, it may be useful to pre-compute the visibility and use only the visible points as shadow samples. Loop 2 prepares a list of shadow samples that belong to each buffer. Note that this procedure is exactly the same as the bin-based visibility sorting method in section 3. Thus, the same code can be reused here.

Each pixel in the map stores the opacity value, which is a summation that produces the integral term Ω in equation (1). Thus each line segment can be rendered antialiased with full hardware support in any order (the order can be arbitrary due to the commutative nature of integration). The alpha buffer is accumulated each time the volume is drawn with the OpenGL blend mode *glBlendFunc(GL_ONE,GL_ONE).* The depth buffer is disabled. Clipping in line 10 ensures correct contribution of alpha values from the primitives and culls most primitives, speeding up the algorithm.

As loop 3 and 4 use only two opacity map buffers at a time, the memory requirement is independent of the total number of opacity maps computed. In loop 4, the shadow is computed only once for each sample. So, the amortized cost of the algorithm is linear in the number of samples. The overall complexity is *O(NM)* since the scene is rendered for each map, but the rendering cost is low with hardware acceleration.

The sample function in loop 4 can be any standard pixel sampling function such as a box filter, or higher-order filters such as Bartlett filter and Gaussian filter. For the examples shown here, a 3x3 averaging kernel is used. Such filtering is possible because alpha values are stored instead of depths. The sampled opacity values Ω_{prev}, $\Omega_{current}$ are linearly interpolated for each point p_k (Figure 14).

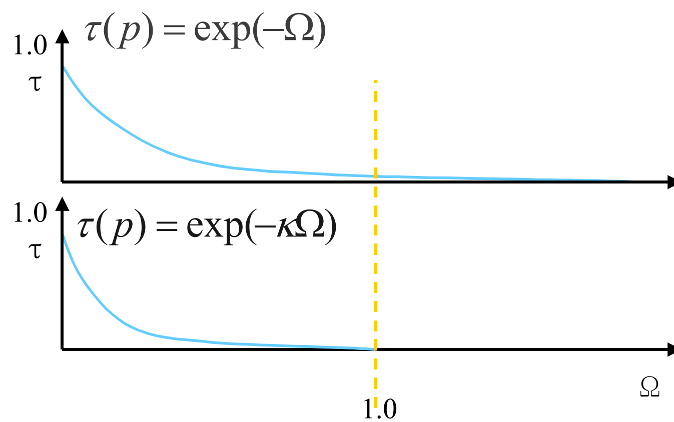
Figure 14. Interpolating the opacity values



A higher order interpolation may be used. For example, four buffers can be used for a cubic-spline interpolation.

A volume turns opaque as the opacity Ω reaches infinity. The quantization in the alpha channel limits the maximum amount of opacity that a pixel can represent. A constant κ in line 15 controls the scaling of opacity values such that $e^{-\kappa} = 2^{-d}$, where d is the number of bits per pixel (for example, κ is about 5.56 for 8 bit alpha buffer). Thus, an opacity value of 1.0 represents a complete opaqueness.

Figure 15. Scaling the opacity



4.2 Additional Clipping

A simple modification can yield a significant speedup in the basic algorithm. When shadow sample points coincide with the end points of the line segments, we can exploit the fact that the line segments are depth-sorted. At each opacity map generation step, we can check if the line segments reside in the current depth range (D_{i-1} and D_i). The line 10 in Routine 2 (shown below)

10. Render the scene clipping it with D_{i-1} and D_i.

can be augmented as

10.a For each line segment (p1,p2) , compute Depth(p1) and Depth (p2).

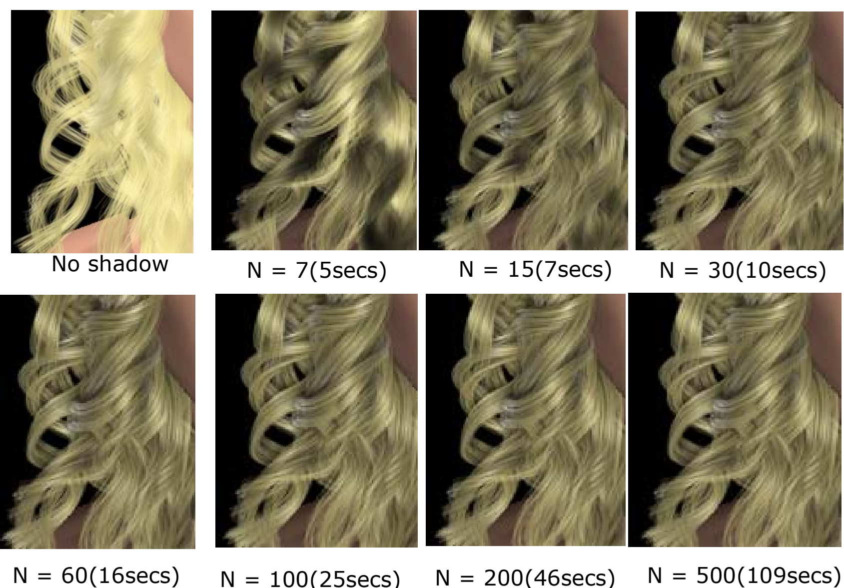
10.b Draw the line only if D_{i-1} <.Depth(p1) < D_i or D_{i-1} <.Depth(p2) < D_i

With this additional scene culling scheme, the observed time complexity becomes close to *O(N)*.

4.3 Examples

Figure 16 illustrates a test scene where the number of maps (N) was varied. Note that the tradeoff between speed and image quality can be achieved by varying the number of maps. The rendering time is linear in the number of maps when the basic algorithm was used. With the modification in section 4.2, the rendering time becomes sub-linear in the number of maps (about 12 secs for N = 500) since the number of primitives drawn to each map decreases as the number of maps increases.

Figure 16. Results



4.4 Further Readings

More details of the original deep shadow maps algorithm as well as an excellent discussion on aliasing in hair shadows can be found in

- T. Lokovic and E. Veach. Deep shadow maps, SIGGRAPH Proceedings, 2000, 385-392

For more discussions and results for the Opacity Shadow Maps algorithm, refer to

- Tae-Yong Kim and Ulrich Neumann, Opacity Shadow Maps, Eurographics Rendering Workshop 2001 (reprinted in the course note).

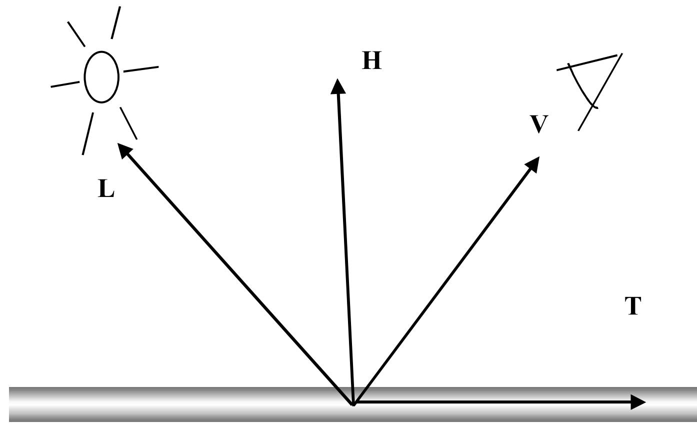
A modification on the transmittance function computation in OSM is suggested in

- Johnny Chang, Jingyi Jin, Yizhou Yu, A Practical Model for Mutual Hair Interactions, ACM SIGGRAPH Symposium on Computer Animation, July 2002 (reprinted in the course note).

5. Shading Model

A shading model describes how much a hair fiber reflects or transmits in a given direction when the hair fiber is fully lit. Since global aspects such as shadows are not accounted for, the term *local shading model* is often used. It is often assumed that the shape of a hair strand on its local neighborhood is a straight cylinder (Figure 17). A hair shading model is often constructed the function of three vectors, L, the light's direction, V, the viewing direction, and T, the tangent vector.

Figure 17. Hair shading geometry



The most commonly used shading model is the one originally developed by Kajiya and Kay. The model is composed of a Lambertian diffuse component and an anistropic specular component. In the Kajiya-Kay model, a Lambertian cosine falloff function is used for diffuse lighting. The closer the light is to the normal, the more illumination is received.

$$\Psi_{Diffuse} = K_d \sin(T, L)$$

, where K_d is a scaling parameter for the diffuse illumination and (V_1,V_2) denotes the angle between two vectors V_1,V_2.

A non-diffuse (specular) illumination is computed using the viewing vector V. The specular illumination becomes the biggest when the half vector H = (L + T) / 2 becomes perpendicular to the tangent vector. The Kajiya-Kay model computes

$$\Psi_{Specular} = K_s[(T \cdot L)(T \cdot V) + \sin(T, L)\sin(T, V)]^p$$

Thus, the amount of light a hair fiber scatters in the direction of V can be written as

$$\Psi_{Hair} = \Psi_{Diffuse} + \Psi_{Specular} = K_d \sin(T, L) + K_s[(T \cdot L)(T \cdot V) + \sin(T, L)\sin(T, V)]^p$$

Multiplying the transmittance function τ computed from section 4, the (shadowed) color of a point on the hair fiber can be expressed as

$$\Psi_{Hair} = \tau(\Psi_{Diffuse} + \Psi_{Specular})$$

To be accurate, the transmittance function and shading model should be computed at every pixel sample. However, the colors tend to smoothly vary along hair's length. In practice, computing the shaded color only at the end points of line segments often yield good results (analogous to the Gouraud shading for polygons).

5.1 Further Readings

Improvements on the original Kajiya-Kay model were introduced by Banks [1994] and Goldmann [1997]. Read the following papers for more details.

- J. Kajiya and T. Kay, Rendering fur with three-dimensional textures, SIGGRAPH Proceedings, Vol. 23, pp. 271-280, 1989.
- D. C. Banks, Illumination in diverse codimensions, SIGGRAPH Proceedings, pp. 327-334, 1994
- D. Goldman, Fake Fur Rendering, SIGGRAPH Proceedings, pp. 127-134, 1997.

6. Data structure

Since shadows are view-independent, it is often convenient to precompute the shadow values at the end points of each line segment and reuse the shadow values during the viewpoint change. For each line segment, we use the following data structure.

Pos1	Tangent1	Color1	Shadow1
Pos2	Tangent2	Color2	Shadow2

The position of each end point comes from the hair model. The tangent vectors are derived from the position. With these position and tangent vectors, the (unshadowed) colors are computed with the shading model. The shadow values are computed with the opacity shadow maps algorithm.

Then, the entire procedure of hair rendering using graphics hardware can be summarized as

Routine 3

Setup pass:

Compute the visibility order

Compute shadow values

Drawing pass

For each line segment L_i ordered due to the visibility order

Set thickness (alpha value)

Compute the shaded color (L_i.color1)

Compute the shaded color (L_i.color2)

C1 = L_i.color1 * L_i.shadow1

C2 = L_i.color2 * L_i.shadow2

glBegin(GL_LINES)

glColor3fv(C1)

glVertex3fv(L_i.Pos1)

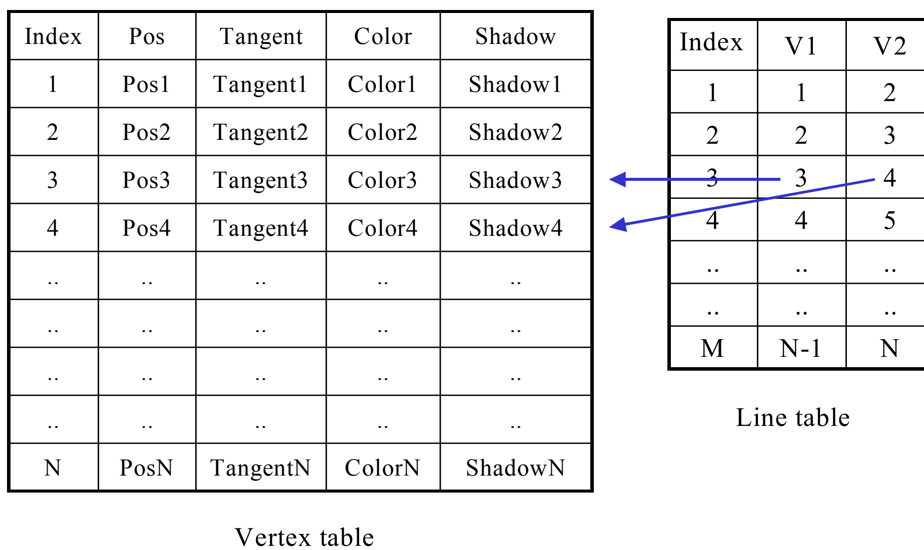
glColor3fv(C2)

glVertex3fv(L_i.Pos2)

glEnd()

Since line segments are connected, all the information shown above is duplicated. A more efficient implementation is thus to store all the position, color, tangent information in a vertex table and let each line be represented by indices to the table, as shown in Figure 18.

Figure 18. Data structures



Index	Pos	Tangent	Color	Shadow
1	Pos1	Tangent1	Color1	Shadow1
2	Pos2	Tangent2	Color2	Shadow2
3	Pos3	Tangent3	Color3	Shadow3
4	Pos4	Tangent4	Color4	Shadow4
..	..	..	..	..
..	..	..	..	..
..	..	..	..	..
..	..	..	..	..
N	PosN	TangentN	ColorN	ShadowN

Vertex table

Index	V1	V2
1	1	2
2	2	3
3	3	4
4	4	5
..	..	..
..	..	..
M	N-1	N

Line table

7. Hair shading with Programmable Vertex Shader

The computation of shaded color can be further accelerated with the use of programmable vertex shader. In this case, the color field will be computed on the fly inside the vertex shader. Routine 3 will change as follows.

Routine 4

Setup pass:

Compute the visibility order

Compute shadow values

Drawing pass:

For each line segment L_i ordered due to the visibility order

Set thickness (alpha value)

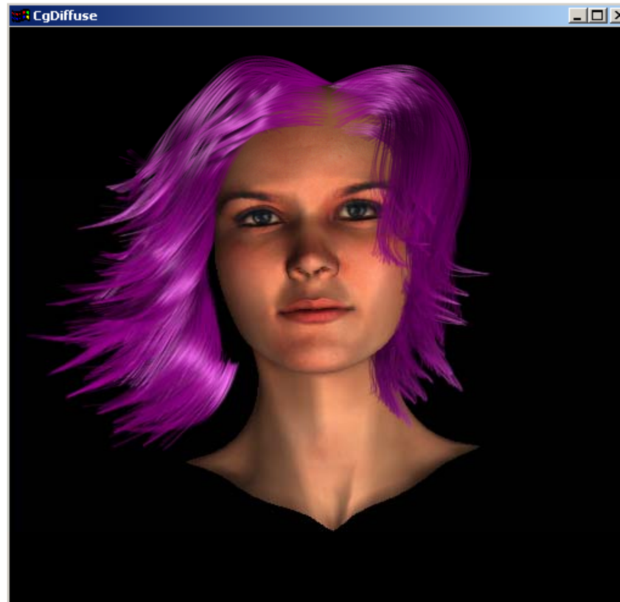
Draw L_i with programmable shader

The inputs to the vertex shader are the camera position, the light position, shading parameters, position, tangent vector, and shadow values for each vertex. An example implementation of a hair shader is given in Figure 19 as nVidia CG program.

Figure 19. cg program for Kajiya-Kay shading model

```
// Input structure to the vertex program
struct PerVertexInput {
    float4 Position : POSITION; // Position in model space
    float3 Normal : NORMAL; // Tangent Vector in model space
    float3 Shadow : TEXCOORD0;
};

// Structure output by the vertex program
struct Output {
    float4 Position : POSITION; // Position in clip space
    float4 DiffuseLight : COLOR0; // Diffuse light
    float4 SpecularLight : COLOR1; // Specular light
};

// Program executed for every vertex
Output main(PerVertexInput IN, // Per-vertex input
            uniform float4x4 ModelViewProj, // Transform matrix from model space to clip space
            uniform float3 LightPos, // Light position in model space
            uniform float3 EyePos, // Eye position in model space
            uniform float4 Kd, // Diffuse coefficient
            uniform float4 Ks, // Specular coefficient
            uniform float  Shininess, // Specular exponent
            )
{
    Output OUT;
    float4 diffuse,specular;
    float3 L,V,T;
    float  sinD;
    float  sinTL,sinTV;
    float  dotTL,dotTV;
    float4 P;

    P = IN.Position;

    // Transform the position from model space to clip space
    OUT.Position = mul(ModelViewProj, P);

    // Compute the light vector:
    // Normalized vector from vertex to light position
    L = normalize(LightPos - P.xyz);
    // Compute the eye vector:
    // Normalized vector from vertex to eye position
    V = normalize(EyePos - P.xyz);

    // Compute the diffuse light:
    sinD = dot(L,IN.Normal);

    diffuse = Kd;
    diffuse.xyz *= sqrt ( 1 - sinD * sinD) ;
    diffuse.xyz *= (IN.Shadow.x);

    OUT.DiffuseLight = diffuse;

    dotTL = dot(IN.Normal,L);
    dotTV = dot(IN.Normal,V);
    sinTL = sqrt ( 1- dotTL * dotTL);
    sinTV = sqrt ( 1- dotTV * dotTV);

    specular.xyz = Ks.xyz * pow (dotTL * dotTV + sinTL * sinTV, Shininess);
    specular.xyz *= (IN.Shadow.x);
    OUT.SpecularLight.xyz = specular.xyz;

    return OUT;
}
```

Figure 20. Hair shaded in realtime



7.1 Further Readings

For more details on the CG compiler, refer to CG manual that can be downloaded from http://www.nvidia.com