Real-Time Rendering of Human Hair using Programmable Graphics Hardware

Martin Koster, Jörg Haber, Hans-Peter Seidel

MPI Informatik, Saarbrücken, Germany

E-mail: {koster,haberj,hpseidel}@mpi-sb.mpg.de

Abstract

We present a hair model together with rendering algorithms suitable for real-time rendering. In our approach, we take into account the major lighting factors contributing to a realistic appearance of human hair: anisotropic reflection and self-shadowing. To deal with the geometric complexity of human hair, we combine single hair fibers into hair wisps, which are represented by textured triangle strips. Our rendering algorithms use OpenGL extensions to achieve real-time performance on recent commodity graphics boards. We demonstrate the applicability of our hair model for a variety of different hairstyles.

Keywords: *hair rendering, programmable graphics hardware, anisotropic reflection, shadow maps, opacity maps*

1 Introduction

Hair undoubtedly plays an important role in our society. Psychological studies have confirmed that people pay attention to the hairstyle of their counterpart right after their face and before their clothing. This result reinforces the demand for a realistic presentation of human hair in computer graphics applications. Despite the fact that GPU's evolved very rapidly over the past years, it is still difficult to render about 100 000 single hair strands in real-time. Further rendering challenges arise from the specific material properties of hair. On the one hand, hair exhibits anisotropic reflection behavior like brushed metal or satin. This material property, however, is not supported by standard graphics API's like OpenGL or DirectX. On the other hand, hair can be seen as a dense volume that exhibits large amounts of self-shadowing. Due to the huge number of single hair strands, it is computationally very expensive to cast shadows on each hair fiber. The fact that hair can be considered as a semi-transparent material makes the problem of shadowing and self-shadowing even harder.

In this paper, we propose a complete wisp model based on textured triangle strips. Our hair model features the most important lighting effects for human hair: anisotropic reflection behavior and self-shadowing. To achieve real-time rendering, we present efficient implementations of these effects exploiting the capabilities of recent commodity graphics boards.

Since standard OpenGL lighting is not sufficient for realistic display of hair, our renderer heavily employs OpenGL extensions that give access to latest hardware features. The rendering system itself is designed as a combination of several plug-ins. Thus, the hair renderer does not depend on particular graphics hardware. With this plug-in infrastructure, it is possible write different hair renderer that are optimally adjusted to different hardware platforms such as, for example, NVIDIA or ATI accelerators.

Section 2 reviews previous approaches to modeling and rendering of hair. In Section 4, the theoretical basis of our rendering algorithms is discussed. Efficient implementations and extensions of the presented algorithms are described in Section 5. Section 6 shows some results achieved with our hair model and rendering algorithms.

2 Previous Work

Rendering of human hair is a very active field of research. Several different models have been proposed for representing human hair in graphics applications. They differ both in the internal representation of hair geometry and in the way optical properties of hair are modeled. The model introduced by Koh *et al.* [14] combines hair strands to a single wisp, which is represented by a NURBS surface. Kim and Neumann [13] presented a multiresolution hair model built upon a hierarchy of hair clusters. The user can edit every level of this hierarchy down to a single hair strand, which is represented by 3D line segments. Due to the massive amount of geometry, hairstyles itself are not displayed in real-time. A wisp model based on trigonal prisms has been proposed by Chen *et al.* [4]. Within the prisms, hair wisps are generated with 2D hair distribution maps. Daldegan *et al.* [5] present a model with physics-based animation and collision detection. Hair strands are described by 3D curves. The rendering is performed using a modified ray tracing approach. Anjyo *et al.* [1] introduced a modeling method for hair based on differential equations. Line segments describing the strands of a particular hairstyle are bent according to their mechanical properties. A model for anisotropic reflection based on the Blinn-Phong model [3] is used to render the hairstyles in a ray tracing approach. The model by Hadap *et al.* [7] focuses on hair animation using fluid dynamics in order to treat hair-hair and hair-air interactions. Individual hairs are modeled using 3D lines. In summary, these models are focused on modeling realistic looking hair, but are not designed for a fast real-time rendering due to their large amount of geometric primitives.

For anisotropic reflection, several algorithms have been introduced. Most of them are not suitable for real-time rendering. Kajiya [9] bases his approach for anisotropic reflection on the general Kirchhoff solution for scattering of electromagnetic waves. Poulin *et al.* [20] build on the concept of small, adjacent cylinders, which represent the scratches that are responsible for anisotropic reflection. Both models show a good visual quality but are complex to evaluate and are thus not suitable for real-time rendering. Marschner *et al.* [18] introduce a model based on a detailed measurement of scattering from hair fibers that explains the appearance of multiple specular highlights. The results are quite impressive and resemble the reflection behavior of human hair. The shading, however, is computed using expensive ray tracing. The approach proposed by Heidrich and Seidel [8] is based on the Banks model [2] and has been optimized for efficient OpenGL rendering. The model introduced by Ward [15] is also an anisotropic reflection model with mathematical simplicity and could be used for real-time applications. A good overview of these and further anisotropic models is given in [10].

Shadowing and self-shadowing of hair has also been investigated in the literature. The strand hair model proposed by LeBlanc *et al.* [16] is based on the well-known shadow mapping algorithm introduced by Williams [22]. Other models employing this z-buffer based approach are the strand models by Chen *et al.* [4] and by Daldegan *et al.* [5]. Although shadow maps are supported by recent graphics hardware, they can not handle semi-transparent primitives correctly. *Deep Shadow Maps* have been proposed by Lokovic and Veach [17]. They support semi-transparent objects and volumetric primitives, making them well suited for fine geometry like hair and fur but also for smoke and fog. A deep shadow map is defined as a rectangular array of pixels in which every pixel stores a visibility function that depends on the depth z of a fragment. Since the computation of these visibility functions for every pixel is very expensive, an efficient implementation using hardware acceleration is not possible. Thus, deep shadow maps are primarily used in ray tracing approaches. *Opacity Shadow Maps*, as proposed by Kim and Neumann [12], are based on deep shadow maps. The algorithm approximates the light transmittance of a volume with a set of discrete planar maps. Every fragment of this volume contributes with its associated alpha value to the maps. The computation of these maps takes place in a preprocessing step. The opacity shadow maps algorithm is used for the multiresolution hair model by Kim and Neumann [13]. Although being much less expensive than deep shadow maps, the original opacity shadow maps algorithm is still not suitable for real-time rendering. However, we will show in this paper that an optimized version of the algorithm can be used to achieve real-time performance with our hair model on recent commodity graphics hardware.

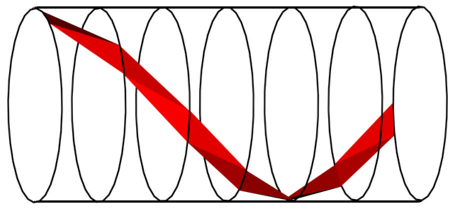


Figure 1. A hair patch projected onto a cylindrical surface. Segments are slightly displaced to achieve a curly effect.

3 The Hair Model

In our approach, we represent a hair wisp by a textured triangle strip, denoted as a *hair patch* subsequently. This reduces the geometric complexity of our hair model and thus accelerates the rendering process. Since we focus on the rendering of hair patches in this paper, this section only reviews the modeling process briefly.

A hair patch is a surface composed of quadrilateral segments, which are in turn decomposed into triangle strips. The number of segments is variable and depends on the type of the desired hairstyle. In contrast to other hair models based on surfaces (see for instance [14]), we can easily model curly hair wisps. To this end, the vertices of the hair patch are projected onto a cylindrical surface. This process is depicted in Figure 1. Segments are slightly tilted to create a curly effect.

The design of a hairstyle is done in a hair editing tool [21]. To create a hair patch on the human scalp, the designer defines its location and its parameters such as *size, stiffness, texture, number & radius of curls, number of segments*, and *hair form*. The actual patch creation is performed automatically in three steps: creating a planar surface composed of quadrilateral segments, curling the planar surface, and bending the surface to avoid hair/hair and

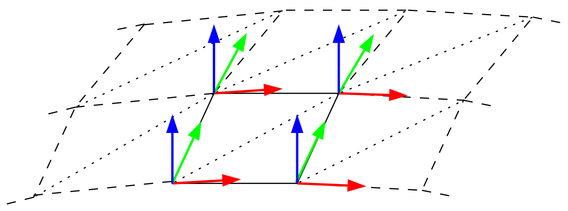


Figure 2. Normal (blue), tangent (red) and binormal (green) are computed per vertex. Thus, every vertex of a hair patch defines its own tangent space.

hair/head collisions. After creating a hair patch, its normal, tangent and binormal are computed automatically per vertex (cf. Figure 2). These local coordinate frames are needed during rendering for the evaluation of our anisotropic reflection model and to apply bump maps for a more realistic appearance of the hair.

4 Rendering Algorithms

This section reviews algorithms for anisotropic reflection and shadowing that have been modified and extended for use with our hair model.

4.1 Anisotropic Reflection

Anisotropic reflection is probably the most characteristic optical property of illuminated hair. In general, the cause of anisotropy is a micro-structure with long, thin features that are aligned in one predominant direction. Anisotropic reflection occurs because the distribution of the surface normals along the features (scratches, fibers) is different from the distribution across them. A distant observer only sees the lighting result, but not the micro-structure.

Several algorithms for computing anisotropic reflection have been discussed in Section 2. We decided to built our algorithm upon the anisotropic reflection model proposed by Heidrich *et al.* [8] due to its simplicity and its capability of exploiting graphics hardware. A further advantage of this model is the possibility of combining it with *dot3 bump mapping*.

For the classical Phong illumination model [19], we need to evaluate the well-known equation:

$$\begin{aligned} I_{\text{out}} &= I_{\text{ambient}} + I_{\text{diffuse}} + I_{\text{specular}} \\ &= k_a I_a + (k_d (L \cdot N) + k_s (V \cdot R)^{k_e})\, I_{in} \end{aligned}$$

where L is the direction towards the light source, N is the surface normal, V is the direction towards the viewer, and R is the reflected light direction. For the the extended Blinn-Phong model [3], we have a slightly different specular term:

$$I_{\text{out}} = k_a I_a + (k_d (L \cdot N) + k_s (H \cdot N)^{k_e})\, I_{in}$$

where H is the half-vector between L and V.

These equations are hard to evaluate when we want to represent anisotropic material. Thin hair, for instance, has no "traditional" surface normal. Instead of one single normal, each point of the one-dimensional fiber exhibits an infinite number of normals, which are orthogonal to the fiber's tangent. The correct approach would be to integrate over all these normals, which is obviously too hard to compute. Therefore, the most significant normal N' that maximizes the dot product $L \cdot N$ is chosen from this circle. This normal N' is co-planar with L and V. The new problem that arises is the computation of the normal N'. Fortunately, there is no need to explicitly compute N'. Instead, the dot products $L \cdot N'$ and $V \cdot R$ can be expressed by the vectors L, V, and the tangent vector T:

$$\begin{aligned} L \cdot N' &= \sqrt{1 - (L \cdot T)^2} \\ V \cdot R &= \sqrt{(1 - (L \cdot T)^2)\,(1 - (V \cdot T)^2)} - \\ &\quad (L \cdot T)\,(V \cdot T) \end{aligned}$$

For the Blinn-Phong model, the representation of the specular term is even more compact:

$$H \cdot N' = \sqrt{1 - (H \cdot T)^2}$$

Using these substitutions, we can efficiently evaluate the Phong and the Blinn-Phong model for any point in space, The computations only depend on the dot products $L \cdot T$ and $V \cdot T$ (Phong) or $H \cdot T$ (Blinn-Phong).

Yet, this model does not take into account self-shadowing. Since we combine this technique with a shadow algorithm, we may neglect self-shadowing at this stage. The shadow algorithm presented in the next section cancels out undesired specular highlights that are in shadow.

4.2 Shadows

For the shadowing of hair, we have developed a modified version of the opacity shadow maps algorithm proposed by Kim and Neumann [12]. The original algorithm works as follows: a hair volume is subdivided into several parallel slices perpendicular to the light direction. Figure 3 illustrates this subdivision. Each of these slices is rendered to the alpha buffer one after another. Thus, each fragment contributes with its associated alpha value to a map denoted as an *opacity map*. Each time a map is rendered, a fixed set of sample points that lie between two maps is shadowed. A detailed description of the shadowing of a fragment is

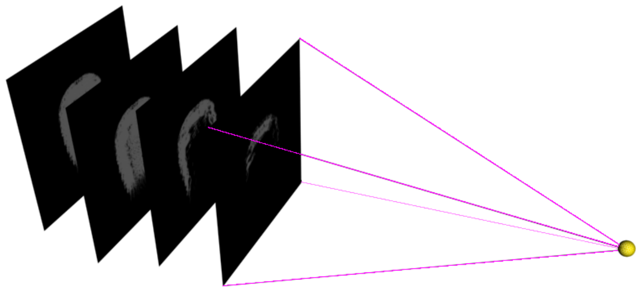


Figure 3. The hair volume is sliced into a set of planar maps storing alpha values.

given later in this section. Maps that are not needed anymore are dropped to reduce the amount of memory needed. In our patch-based hair model, evaluating and precomputing shadows for a fixed set of sampling points (such as vertices) would not only lead to disturbing visual artifacts, but in addition it would be impossible to cast realistic shadows onto the head.

In contrast to the original algorithm, the goal of our modified opacity maps version is to compute shadows on a per-pixel basis, allowing us also to cast shadows of the hair patches onto the head. To this end, the shadowing process is divided into two steps:

1. Compute all opacity maps, if an update is necessary.
2. Compute shadow of fragments by back-projecting them into the maps.

Parts of this algorithm can be efficiently implemented in hardware. The pseudo code of the modified algorithm shown below uses the following notation: n is the number of maps, d_i denotes the the depth of the ith map from the light source, `BoundingBox` is the set of vertices defining the bounding box of the hair model, `depth`(p_i) retrieves the depth of the vertex p_i from the light source, and `maps` is the set of opacity maps. The function *interpolate*(a, b, α) linearly interpolates between a and b as follows: $\alpha a + (1-\alpha)b$. The transmittance function is denoted as τ, κ is the proposed value of 5.56, and Φ is the computed shadow of a fragment. In the pseudo code, a difference is made between vertices p_i in the geometry stage and fragments f_i in the rasterizer stage. The procedure *applyInverseLightTransform*() sets modeling and projection parameters for rendering the scene from the light source, and *applyInverseCameraTransform*() restores the original camera parameters.

In *createMaps*(), the scene is rendered to the alpha buffer n times from the position of the light source, clipped by the depth of each map (lines 10 and 14). To determine the depth of each map, we have to find the nearest and farest point from the light source. Instead of searching through all

Pseudo code:

```
1.   maps[1..n];
2.
3.   procedure createMaps()
4.   begin
5.     applyInverseLightTransform();
6.     for all $p_i$ in BoundingBox :
7.       $d_1 = \min(\texttt{depth}(p_i))$;
8.       $d_n = \max(\texttt{depth}(p_i))$;
9.     clearPBuffer();
10.    renderSceneBetween(nearPlane, $d_1$);
11.    readAlphaBufferTo(maps[1]);
12.    for $i = 2, \ldots, n$ :
13.      $d_i$ = interpolate($d_1$, $d_n$, $(i-1)/(n-1)$);
14.      renderSceneBetween($d_{i-1}$, $d_i$);
15.      readAlphaBufferTo(maps[$i$]);
16.  end;
17.
18.  procedure computeShadow()
19.  begin
20.    applyInverseCameraTransform();
21.    for each fragment $f_i$ :
22.      $\Omega_{f_i}$ = scale * project($f_i$, maps);
23.      if (linearShadowing)
24.        $\tau(f_i) = 1.0 - \Omega_{f_i}$;
25.      else
26.        $\tau(f_i) = \exp(-\kappa * \Omega_{f_i})$;
27.      $\Phi(f_i) = 1.0 - \tau(f_i)$;
28.  end;
```

the vertices, we only check the corner vertices defining the bounding box of the hair model (line 6). This step is therefore independent from the used hairstyle. Each pixel in the map stores an alpha value Ω that approximates the opacity relative to the light at the position of the pixel. Values between these maps are computed by linear interpolating the alpha values of two neighboring maps depending on the depth z of the fragment. For a fragment f_i with depth z_i, its opacity value $\Omega(f_i)$ is computed as:

$$\begin{aligned} \Omega(f_i) &= t_i\, \Omega_{\text{curr}} + (1 - t_i)\, \Omega_{\text{prev}} \ , \\ t_i &= (z_i - d_{\text{prev}})/(d_{\text{curr}} - d_{\text{prev}}) \ , \end{aligned}$$

where Ω_{curr} and Ω_{prev} are the opacity values sampled from the two neighboring maps and d_{curr} and d_{prev} are the depths of these two maps. This step is performed by the *project*() function in line 22. `scale` is a floating point factor that scales the intensity of the shadow to allow for shadows that are not too dark. Similar to OpenGL's fog mode, we have implemented two shadowing modes: *linear* shadowing and *exponential* shadowing, which differ in the computation of the transmittance function. After the retrieval of a fragment's opacity, the transmittance for the fragment in linear shadowing mode is simply computed as: $\tau(f_i) = 1 - \Omega(f_i)$. In exponential shadowing mode, we compute: $\tau(f_i) = \exp(-\kappa\,\Omega(f_i))$. Finally, the shadow $\Phi(f_i)$ of a fragment f_i is computed as $\Phi(f_i) = 1.0 - \tau(f_i)$. We found that the optimal shadowing mode depends on the hairstyle: for smooth, flat patches, the linear mode usually works better, while for voluminous, curly hair, the exponential mode is suited better.

5 Implementation

The algorithms presented in Section 4 are implemented in two hair rendering plug-ins using OpenGL. Since there were no graphics cards with support for fragment programs available when we started our project, one plug-in is targeted for GeForce3 class graphics cards using vertex programs, texture shaders and register combiner. A second plug-in using vertex and fragment programs was implemented, when the NVIDIA GeForce FX and the ATI Radeon 9500 were introduced. The enhanced features of these cards are supported through OpenGL extensions.

Anisotropic lighting is combined with `dot3` bump mapping to create the illusion of a fine hair structure on the flat patches. A good overview of bump mapping and similar techniques is given in [11]. All computations are performed in local tangent space. Thus, all vectors are transformed using the following matrix:

$$M = \begin{pmatrix} T_x & T_y & T_z & 0 \\ B_x & B_y & B_z & 0 \\ N_x & N_y & N_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where $T = (T_x, T_y, T_z)$ denotes the tangent vector, $B = (B_x, B_y, B_z)$ the binormal and $N = (N_x, N_y, N_z)$ the normal vector. A look-up texture is precomputed and the vectors $L \cdot T$ and $H \cdot T$ serve as texture coordinates to look up the diffuse and specular components as defined in Section 4.1. The dot product is computed per-pixel either in the texture shader stage on the GeForce3 or in the fragment program on the GeForce FX / Radeon 9500.

The Opacity Maps are created during the initialization of the hair model and are only updated if the relative position of the head to the light source changes. In contrast to the original opacity shadow maps algorithm, we store all maps in video memory at the same time. This consumes a certain amount of memory, but it allows to compute shadows on a per-fragment basis rather than on a per-vertex basis as done in [12]. The maps itself are stored in a 3D texture of size $X \times Y \times n$, where $X \times Y$ is the resolution and n the number of maps. We typically use a resolution of 512x512 pixels and n = 16 maps. Such 3D textures are hardware accelerated on graphics boards with GeForce3 and higher GPU's.

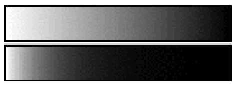
Texture Unit 0	**Texture Unit 1**	**Texture Unit 2**
`texture_2d()`	`texture_3d()`	`dependent_ar(tex1)`
(R_0, G_0, B_0, A_0)	(0, 0, 0, A_1)	
hair texture	opacity map texture	transmittance function look-up texture

Figure 4. Using the `dependent_ar()` texture shader in texture unit 2, the transmittance function is evaluated depending on the result of the retrieved alpha value in texture unit 1. Texture unit 2 contains either a linear or an exponential look-up table.

The advantage of 3D textures is the ability to automatically generate mip-maps using the `SGIS_generate_mipmap` extension. By using the `GL_LINEAR` texture filter, linear interpolation between two neighboring maps is achieved at no extra costs. Memory allocation for all necessary textures is done in a preprocessing step. To speed up the map creation, only the necessary OpenGL states are enabled. Both time-consuming lighting and writing to the color channels of the frame buffer are disabled. Texturing is necessary, because the alpha values are stored together with the RGB values in the hair texture. The blending equation set to `glBlendFunc(GL_ONE, GL_ONE)` and depth testing is disabled, thus the sorting order of the patches is unimportant. Maps are rendered n times to an offscreen pixel buffer, clipped by the depth of the actual map. The alpha buffer then is copied to to its corresponding layer of the 3D texture by a fast `glCopyTexSubImage`. To soften the shadow edges and to avoid artifacts, the user may enable a separable averaging kernel of any size, see Figure 12. The necessary functions for convolution are defined in the `ARB_imaging` extension. Thus, the alpha buffer gets blurred, while transferring its contents to the 3D texture. After the creation of the maps, the scene is rendered from the view of the camera. Every fragment to be shadowed is now transformed into the 3D texture using *projective texturing* [6], which is performed in a vertex program. This part of the implementation is the same for all architectures supporting vertex programs. In the fragment stage, however, we use different plug-ins handling the hardware features of the given graphics card.

5.1 Rendering using Register Combiner

The plug-in for GeForce3 class graphics cards uses vertex programs, texture shaders, and register combiners. In the texture shader stage, a transmittance value is looked up, depending on what shadowing method is used. The opacity

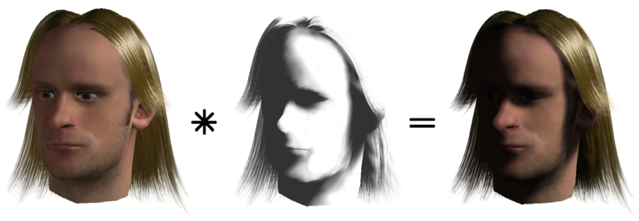


Figure 5. The result of the anisotropic lighting pass (left) is combined with the result of the shadowing pass (middle) via multiplicative blending.

value of the fragment is used as the texture coordinate for the look-up texture that stores either the linear or the exponential transmittance function as described in Section 4.2. Figure 4 illustrates the configuration of the different texture stages.

Because of limitations in the fragment stage, rendering on the GeForce3 is done in two passes. First, the anisotropic lighting is computed and then shadowing is applied. The shadowing pass outputs a grey-scale texture that is combined with the result of the anisotropic lighting pass via multiplicative blending, see Figure 5.

5.2 Rendering using Fragment Programs

On graphics boards with support for fragment programs, anisotropic lighting and shadowing is performed in a single rendering pass. Instead of looking up the value of the transmittance function in a precomputed texture, it is calculated on-the-fly for every pixel. This is possible due to the extended arithmetic instruction set allowing to evaluate the exponential function for every pixel. The fragment program was written in Cg from NVIDIA, see Table 1. The resulting assembler code, however, has been optimized manually, thus reducing the code from 31 to 19 assembler instructions.

5.3 Self-Shadowing Artifacts and Offsetting

Since we use alpha maps with relatively large alpha values, some self-shadowing artifacts possibly appear at the border of a map. For a better explanation of this phenomenon, we consider a fragment directly illuminated by the light source, see Figure 6. This fragment lies exactly in the middle of two opacity maps. The map closer to the light source contains the alpha value 0, because the fragment lies on the upper surface of a hair patch. The other map contains an alpha value of 1. Linear interpolation results in a alpha value of 0.5. Therefore a shadow of 0.5 is computed for this fragment (in linear shadowing mode), although it is directly exposed to the light source.

```
struct fpInput {
  float2 decalTexCoord  : TEXCOORD0;
  float2 tgtTexCoord    : TEXCOORD1;
  float3 opacTexCoord   : TEXCOORD3;
  float3 L              : TEXCOORD5;
  float3 H              : TEXCOORD6;
};

float4 main (fpInput IN,
             uniform sampler2D decalTex,
             uniform sampler2D tgtTex,
             uniform sampler2D lookupTex,
             uniform sampler3D opacTex,
             uniform float3 specCol,
             uniform float3 diffCol,
             uniform float scale,
             uniform float linear) : COLOR
{
  float4 decalCol, lookupCol, fragmentCol;
  float3 T;  // tangent vector
  float2 lookupTexCoord;  // L.T and H.T
  float omg;  // omega

  // color from hair texture
  decalCol = tex2D(decalTex, IN.decalTexCoord);

  // expand tangent vector to [-1,1]
  T = 2*(tex2D(tgtTex, IN.tgtTexCoord).rgb - 0.5);

  // dependent texture lookup (anisotropic illum.)
  lookupTexCoord.x = dot(IN.L, T);
  lookupTexCoord.y = dot(IN.H, T);
  lookupCol = tex2D(lookupTex, lookupTexCoord);

  // calculate unshadowed fragment color
  fragmentCol.rgb =
    lookupCol.rgb * decalCol.rgb * diffCol +
    float3(lookupCol.a,lookupCol.a,lookupCol.a) *
    specCol;
  fragmentCol.a = decalCol.a;

  // compute light transmittance in hair volume
  omg = scale * tex3D(opacTex, IN.opacTexCoord).a;

  // linear/exponential shadowing?
  if (linear > 0)
    fragmentCol.rgb *= 1.0 - omg;
  else
    fragmentCol.rgb *= exp(-5.56 * omg);

  return fragmentCol;
}
```

Table 1. Fragment program written in Cg for anisotropic reflection and shadowing.

To handle this problem, the maps are shifted along the light direction by adding a small offset to the r-texture coordinate that defines the depth within the opacity texture, see Figures 7 and 13.

The maps are stored in the video memory of the graphics board and the computation of the transmittance function and the shadow of the fragment is done on the GPU using fragment programs and register combiners, respectively. No

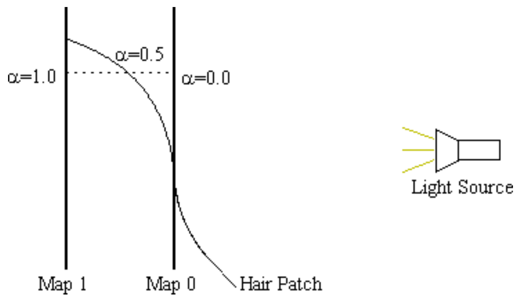


Figure 6. Fragment with a wrongly interpolated alpha value. The result is a shadowed fragment, although it should be fully illuminated.

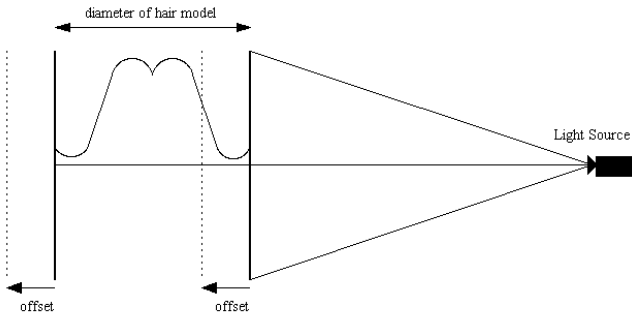


Figure 7. To avoid self-shadowing artifacts, opacity maps are sightly shifted along the light direction by a small offset.

data transfer between video and main memory is needed. We found that 16 maps are typically sufficient for shadow approximation for most hairstyles (see also Figure 13).

6 Results

Rendering our hair model with the real-time algorithms presented in Sections 4 and 5 yields a realistic appearance of human hair for a variety of hairstyles, see Figures 8–10. A comparison of different rendering techniques is depicted in Figure 11.

We have tested our hair rendering algorithms on several different graphics boards: an ATI FireGL X1 and NVIDIA's GeForce3 and GeForce FX 5800 Ultra. For constant lighting conditions (i.e. light position and/or direction do not change), the complete head model is displayed with a frame rate of about 120 fps on the ATI FireGL and the GeForce FX and about 70 fps on the GeForce3 . These high frame rates are mainly due to the fact that the opacity maps are cached in the 3D texture. For varying lighting conditions or animated hair geometry, the opacity maps have to be updated during rendering. In this

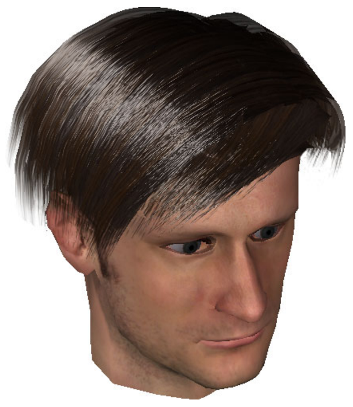


Figure 8. A short hairstyle with strong anisotropic reflection.

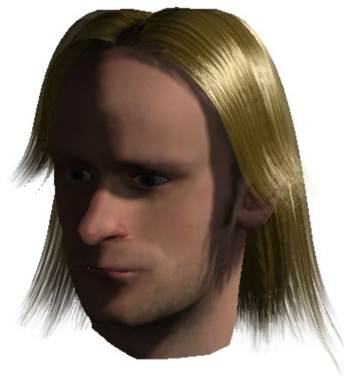


Figure 9. A long hairstyle with anisotropic reflection casts a shadow on the head. Self-shadowing of the head is also achieved with opacity shadow maps.

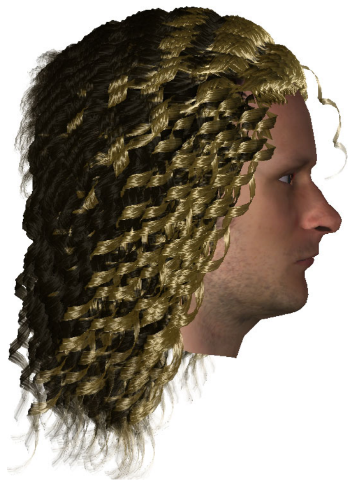


Figure 10. A complex curly hairstyle. In particular, such voluminous hairstyles exhibit self-shadowing of hair wisps.

case, hairstyles such as the ones shown in Figures 8 and 9 achieve frame rates of about 35 fps on the ATI FireGL. The GeForce FX unfortunately drops down to about 11 fps for varying lighting conditions. According to recent discussions in the OpenGL forum (`http://www.opengl.org/discussion_boards/`), the ATI card seems to be much faster in performing the `glCopyTexSubImage()` call than the GeForce FX. This observation is confirmed by our simulations. The rendering plug-in using the register combiner on the GeForce3 yields about 5 fps for moving light sources.

7 Conclusion

We have presented a hair model and rendering algorithms that handle the most important lighting factors for a realistic appearance of human hair: anisotropic reflection and self-shadowing. Our algorithms are designed for real-time rendering on recent commodity graphics boards. We achieve frame rates of up to 120 fps for constant lighting conditions, and up to 35 fps for moving light sources.

The bottleneck of the computations for variable lighting conditions is the update of the opacity shadow maps: for n maps, the scene has to be rendered n times from the position of the light source. Higher frame rates will probably be achieved by optimizing this step, for instance by rendering only those hair patches that lie between the two maps that are considered during each rendering pass. This, however, requires additional computations for the intersection tests that have to be performed when the scene is changed. We are currently investigating the applicability of space partitioning data structures (octrees, BSP trees) to speed up these intersection tests.

References

[1] K.-i. Anjyo, Y. Usami, and T. Kurihara. A Simple Method for Extracting the Natural Beauty of Hair. In *Computer Graphics (SIGGRAPH '92 Conf. Proc.)*, volume 26, pages 111–120, July 1992.

[2] D. C. Banks. Illumination in diverse codimensions. Technical Report TR-94-6, Institute for Computer Applications in Science and Engineering, Jan. 1994.

[3] J. F. Blinn. Models of Light Reflection for Computer Synthesized Pictures. In *Computer Graphics (SIGGRAPH '77 Conf. Proc.)*, volume 11, pages 192–198, July 1977.

[4] L.-H. Chen, S. Saeyor, H. Dohi, and M. Ishizuka. A System of 3D Hair Style Synthesis based on the Wisp Model. *The Visual Computer*, 15(4):159–170, 1999.

[5] A. Daldegan, N. M. Thalmann, T. Kurihara, and D. Thalmann. An Integrated System for Modeling, Animating and Rendering Hair. In *Computer Graphics Forum (Proc. Eurographics '93)*, volume 12, pages 211–221, Sept. 1993.

[6] C. Everitt. Projective Texture Mapping. Available from `http://developer.nvidia.com/object/Projective_Texture_Mapping.html`, 2001.

[7] S. Hadap and N. Magnenat-Thalmann. Modeling Dynamic Hair as a Continuum. In *Computer Graphics Forum (Proc. Eurographics 2001)*, volume 20, pages C329–C338, Sept. 2001.

[8] W. Heidrich and H.-P. Seidel. Efficient rendering of anisotropic surfaces using computer graphics hardware. In *Image and Multi-dimensional Digital Signal Processing Workshop (IMDSP) '98*, pages 315–318, July 1998.

[9] J. T. Kajiya. Anisotropic Reflection Models. In *Computer Graphics (SIGGRAPH '85 Conf. Proc.)*, volume 19, pages 15–21, July 1985.

[10] J. Kautz and H.-P. Seidel. Towards Interactive Bump Mapping with Anisotropic Shift-Variant BRDFs. In *Proc. of the 2000 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 51–58, Aug. 21–22 2000.

[11] M. J. Kilgard. A Practical and Robust Bump-mapping Technique for Today's GPUs. Available from `http://developer.nvidia.com/object/Practical_Bumpmapping_Tech.html`, 2002.

[12] T. Kim and U. Neumann. Opacity Shadow Maps. In *Proc. Eurographics Rendering Workshop 2001*, pages 177–182, 2001.

[13] T.-Y. Kim and U. Neumann. Interactive multiresolution hair modeling and editing. *ACM Transactions on Graphics*, 21(3):620–629, July 2002.

[14] C. K. Koh and Z. Huang. A Simple Physics Model to Animate Human Hair Modeled in 2D Strips in Real Time. In *Proc. Computer Animation and Simulation (CAS 2001)*, pages 127–138, Sept. 2001.

[15] G. J. W. Larson. Measuring and Modeling Anisotropic Reflection. In *Computer Graphics (SIGGRAPH '92 Conf. Proc.)*, volume 26, pages 265–272, July 1992.

[16] A. M. LeBlanc, R. Turner, and D. Thalmann. Rendering Hair using Pixel Blending and Shadow Buffers. *Journal of Visualization and Computer Animation*, 2(3):92–97, July–Sept. 1991.

[17] T. Lokovic and E. Veach. Deep Shadow Maps. In *Computer Graphics (SIGGRAPH 2000 Conf. Proc.)*, pages 385–392, 2000.

[18] S. R. Marschner, H. W. Jensen, M. Cammarano, S. Worley, and P. Hanrahan. Light scattering from human hair fibers. *ACM Transactions on Graphics*, 22(3):780–791, July 2003.

[19] B.-T. Phong. Illumination for Computer Generated Pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[20] P. Poulin and A. Fournier. A Model for Anisotropic Reflection. In *Computer Graphics (SIGGRAPH '90 Conf. Proc.)*, volume 24, pages 273–282, Aug. 1990.

[21] C. Schmitt, M. Koster, J. Haber, and H.-P. Seidel. Modeling Hair using a Wisp Hair Model. Research Report MPI-I-2004-4-001, MPI Informatik, Saarbrücken, Germany, Jan. 2004.

[22] L. Williams. Casting Curved Shadows on Curved Surfaces. In *Computer Graphics (SIGGRAPH '78 Conf. Proc.)*, volume 12, pages 270–274, Aug. 1978.

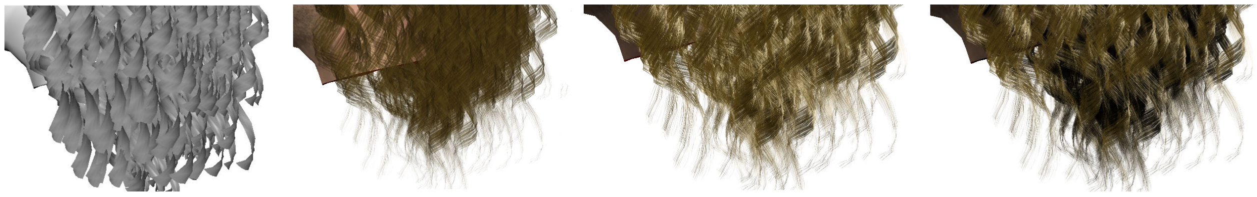


Figure 11. A cluster of hair rendered with different techniques. Left to right: untextured Gouraud shading, textured Gouraud shading (textures with alpha channel), anisotropic shading with per-pixel bump-mapping, anisotropic shading combined with self-shadowing.

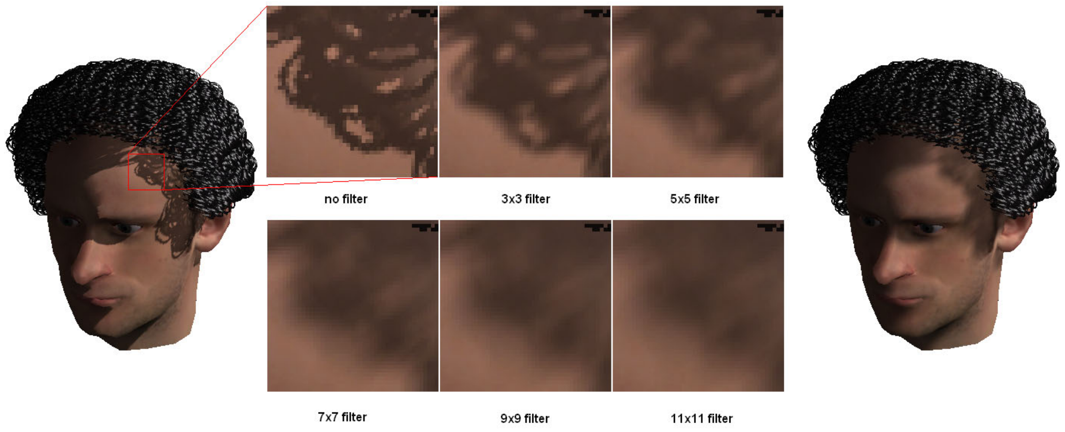


Figure 12. Using an averaging kernel blurs the opacity textures and creates diffuse soft shadows. The maps have a resolution of 512x512 pixels.

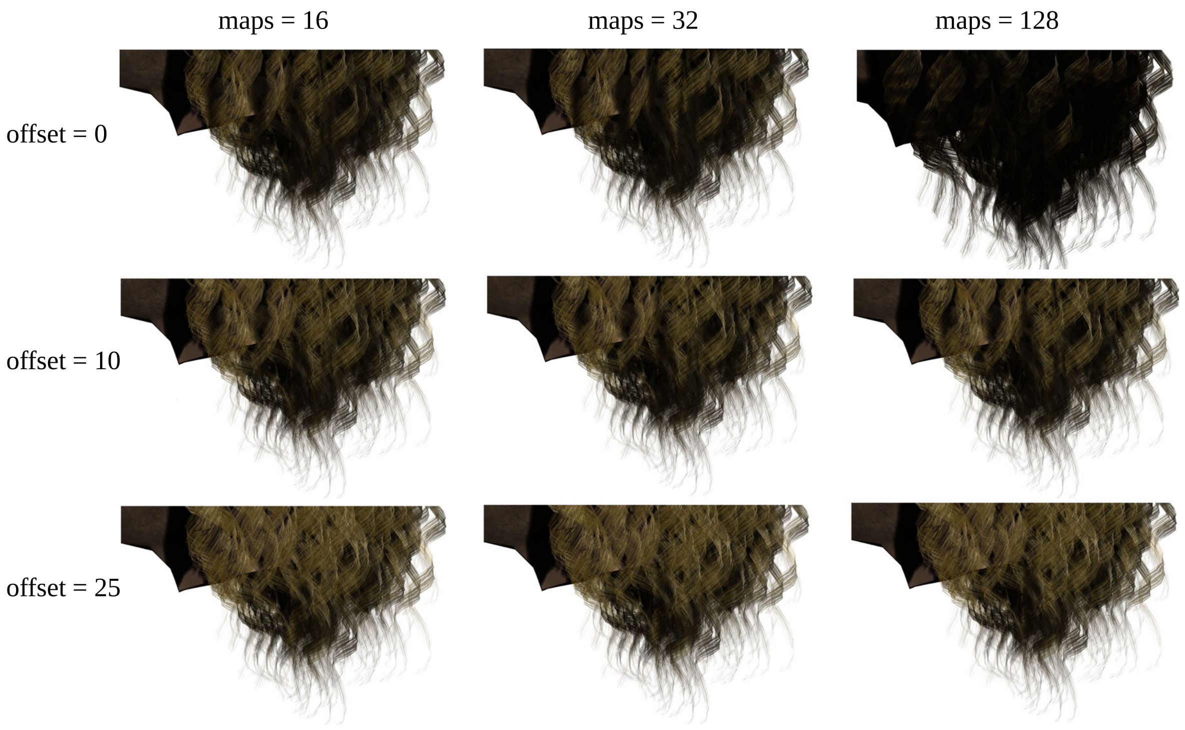


Figure 13. Cluster of hair rendered with 16, 32 and 128 opacity maps and with a different offset. Images without this offset seem too dark. An offset of 25 has shown good results.