# Real-Time Hair Simulation and Visualization for Games M.Sc. Thesis

Henrik Halén
Department of Computer Science
Lund University

Martin Wester
Department of Science and Technology
University of Linköping

Advisors

Torbjörn Söderman
Lead Programmer
EA Digital Illusions CE AB

Jonas Kjellström
Lead Programmer
EA Digital Illusions CE AB

Petrik Clarberg
Department of Computer Science
Lund Institute of Technology

Ken Museth
Department of Science and Technology
University of Linköping

Tomas Akenine-Möller
Department of Computer Science
Lund Institute of Technology

February 19 - 2007

## Abstract

This thesis evaluates, improves and develops methods for generating, simulating and rendering hair in real-time. The purpose is finding techniques which make use of recent hardware to present an as good as possible visual result while keeping performance at such a level that integration into a game scene is viable. The Kajiya-Kay and Marschner lighting models for hair are evaluated, including recent resource saving discretizations to the Marschner model. Two shadowing methods are adapted and investigated for the nature of hair and real-time applications, and one new method is presented as a lower quality and faster alternative for translucent occluders. For dynamics, two models are developed and an existing model for simulating trees is adapted. The implementation uses and takes into account the capabilities and limits of modern graphics hardware, including various techniques that greatly reduces the amount of data sent to the graphics card. The implementation also includes a Maya pipeline for modeling hair. The result is a state-of-the-art rendering and simulation of hair for real-time game purposes.

# Index

Henrik Halén, LTH, Sweden
Martin Wester LiTH, Sweden

# 1 Background

In all areas in real-time computer graphics quality is weighted against speed. Realistically rendered and convincingly moving hair is a relatively new area for real-time applications, especially if the hair is to be incorporated in a game scene. In such a setting it may not be allowed to make use of more than a small fraction of the resources available. Effects such as the ones presented in this thesis are not required for a character to be convincing, not all people have hair. Since this is, or has been, a difficult area, most games avoid it completely, which can be acceptable in some settings; most soldiers wear helmets. In other genres there is a great excess of headwear or other excuses. The argument is usually that bad hair is worse than no hair, which is acceptable since hair is not required. However the lack of hair can limit the believability of characters.

Beyond increased resources, recent hardware introduces new logical capabilities. A point has been reached through previous work where hair can be a believable real-time element. Modern mathematical models describing the nature of hair strands are no longer exclusive to off-line applications. The natural step taken here is improving on recent work, not only visually, but more importantly performance-wise to such a degree that what is presented is usable in settings not dedicated to the task. In most games only a fraction of the resources can be spent on hair rendering and simulation.

## *1.1 Previous Work*

A lot of research has been done regarding the subject of creating hair in computer graphics. It has been an ongoing subject since the 80s. Most of the work has been done for offline purposes due to lack of processing power. However during the most recent years there have also been attempts to make real-time implementations.

### 1.1.1 Hair Modeling

In the past games have either cleverly avoided hair or made a very low detail representation which could be modeled manually. With next generation hardware it is going to be possible to have a more detailed representation of hair; however this also reduces the efficiency of hair modeling. The main obstacle here has been dealing with or representing complex geometry such as hair in a simple way which makes it easy and fast to create. It would be an impossible task to model each strand manually. The obvious solution is to only model a few strands and let the hair follow these guide strands, used by Kim and Neumann [KN02] for example. This is also commonly used by offline hair creation applications such as *Ornatrix* or *Autodesk Maya Hair*. Another common technique is to create force fields, Yizhou Yu [Y01], which can manipulate the growth of the hair locally or globally. This can also be used to create tools like combs.

### 1.1.2 Hair Rendering

Rendering hair has been proven to work well in real-time applications [ND05], [KHS04], [S04]. [ND05] has produced the best visual result so far within this area. For real-time games though, rendering the amount of data used by [ND05] is not feasible when using it together with all other effects not to mention rendering of multiple hair styles in the same scene. One way to reduce the data is to render polygons and make use of textures [KHS04], [S04]. The general problem here is to capture the transparent effect of hair strands and not burden the graphics card with computationally heavy pixel shaders and too much overdraw.

Lighting models in real-time computer graphics are preferably local models and fairly mathematically simple to keep the amount of shader instructions low. There are two shading models describing light reflections from hair. One of them is the Kajiya-Kay light model [KK89], which is a simple model well suited for real-time computer graphics. The second model is the Marschner model [MJC03], a much more complex model which needs to be adjusted to fit real-time graphics, a solution was demonstrated by [ND05].

Hair shadows is much like volume shadows since each hair strand is semi transparent and blocks a fraction of the light. Traditional shadow techniques for games (shadow maps, stencil shadow volumes) do not capture this behavior. Several techniques have been suggested to solve this problem by discretizing the density of the hair in light space [BMC05], [MKBR04], [KN01]. Although shadow techniques based on layered volume shadows such as [KN01] are rendered in real-time, they can be hard to fit in a larger rendering system for games, mainly because one needs to render separate shadow layers for each hair. This could also make it costly to let the hair cast shadows on other things besides itself.

### 1.1.3 Hair Simulation

For real-time simulation today, if all strands are drawn individually or in small groups, simulating all the ~40.000 strands of a head covered with hair is not an option. There is a great number of different solutions to the problem. Most deal with the obvious approach of simulating only a small number of strands and binding the vast number of visual strands to them in some manner. For instance, NVIDIA does this in the Nalu and Luna demos. [BKNC03] and [WLLFM03] both present extensions to the obvious approach with hierarchical adaptive level-of-detail, systems where the number of simulated strands are effectively reduced or increased as needed. [VMT04] present a completely different method where a lattice is simulated fast enough for real-time applications. Strands are encapsulated by and bound to the lattice. Additionally, constraints are added as springs connecting lattice nodes not in the immediate vicinity of each other.

Obviously, dealing with self-collisions in a convincing way for a head seemingly full of hair is not an easy task. All but none of the high-quality simulation algorithms attempt only to simulate at a resolution much lower than the apparent visual result. With this constraint, decent looking self collision gets hard to find. [CJY02] present a state of the art offline algorithm for dealing with self collisions. We are quickly approaching the level of detail where realistic self collision will be needed for real-time simulation, but today, and especially for games today it is not in the scope of this thesis.

## *1.2 Outline*

This thesis will present different techniques for rendering, simulating and creating hair. These techniques will be measured in terms of performance, visual quality and evaluated with respect to usability for games with today's graphics hardware.
Chapter four to six cover a more detailed description of different techniques within the topics modeling, rendering and simulation of hair. Chapter seven will describe our implementation details of suitable techniques. Chapter eight will present performance and evaluation of the implemented methods.

# 2 Hair Modeling

In real life hair has very complex geometry, a single hairstyle consists of hundreds of thousands hair strands. This poses several problems in real-time graphics for games:

- The data has to be reduced to fit the vertex and pixel processing capabilities of the graphics card
- The graphics artists must have a fast work flow and therefore tools are needed to facilitate the creation of hair strands
- There must be methods of creating levels of detail to optimize rendering performance when drawing less visible hairstyles far away
- There must be methods to simulate the hair style

## *2.1 Lines or Polygons*

Today's graphics cards can render both lines and polygons, although the graphics pipeline is constructed to favor polygon rendering rather than line rendering. However, NVIDIA has chosen to render lines in their demos (Nalu and Luna), described in *Hubert Nguyen - Hair Animation and Rendering in the Nalu Demo*, [ND05]. These produce good results when the amount of strands is high enough to produce a thick covering feeling.

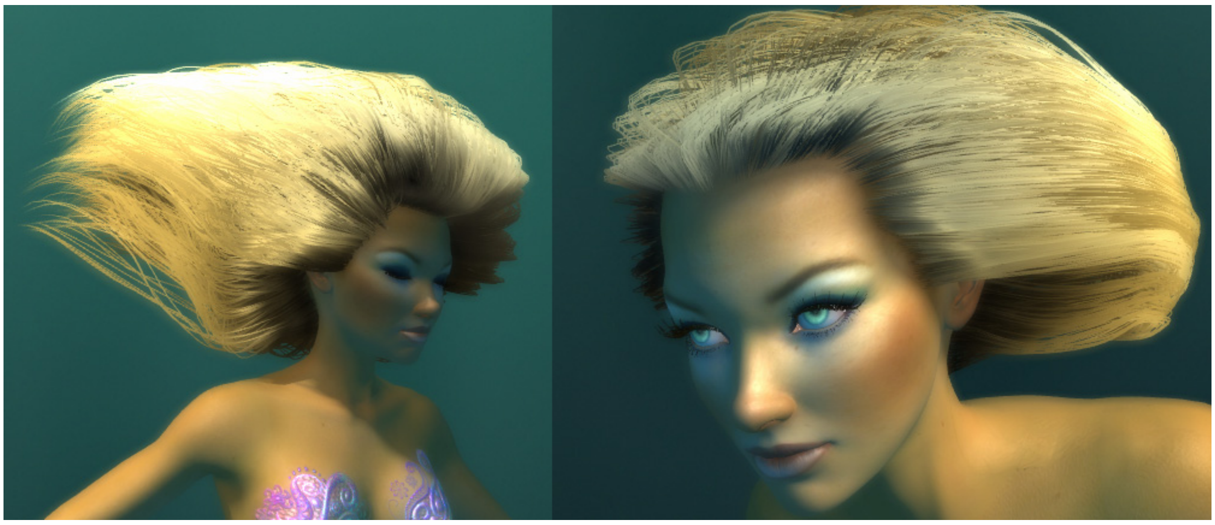


**Figure 2.1.1** *Rendering from NVIDIA's Real-time Nalu Demo. Lines are used as strands. This is a too complex model for games with today's processing power.*

Almost all offline hair rendering methods use lines. Since the method requires many strands it also increases the vertex processing needed. This may be acceptable if you only have to render one hairstyle, but games seldom have only one character at a time on the screen. Because of these limitations line rendering of hair is not suitable for games.

Thorsten Scheuermann and Martin Koster have used a polygonal approach described in [S04]. The same technique is used by Koster et al. [KHS04] and Koh and Huang [KH00]. They model chunks/wisps of hair with polygonal leaves/strips.

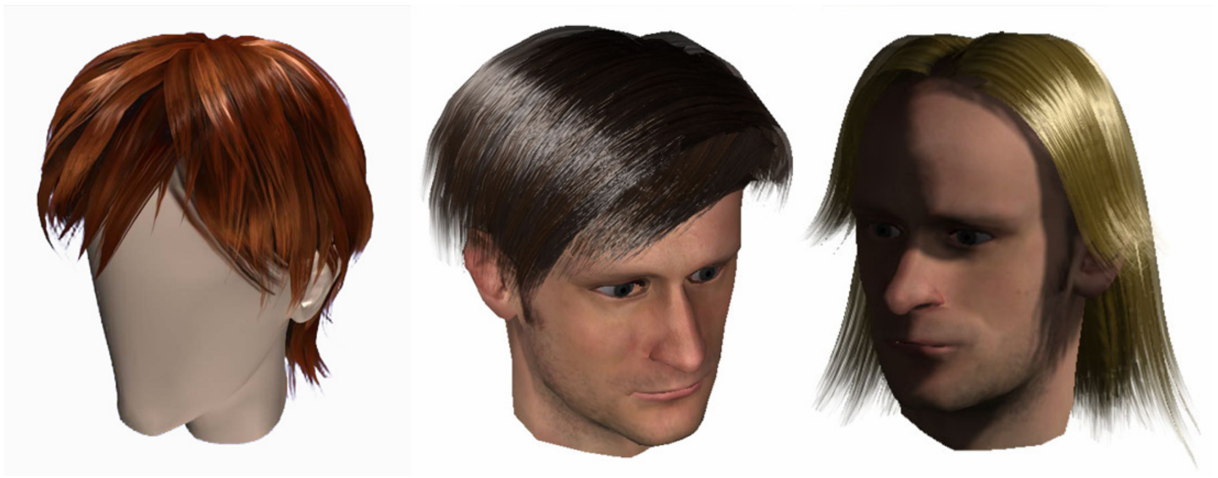


**Figure 2.1.2** *Polygonal models which are more suited for games. Image courtesy of Chuan Koon Koh et al. [KHS04] and ATI.*

This method is better suited for graphics cards; it balances the load on vertex and pixel processors and takes advantage of hardware interpolation and textures to enhance visual quality. It also produces better result at low level of detail which is vital for games. The downside of this technique is that you have to deal with transparency rendering and intersecting polygons.

## 2.2 Wisps

Hair has an observed tendency to form clusters. This phenomenon has been used in several techniques for creating hair. These clusters or wisps as they are called not only produce more realistic results, they also facilitate hair creation and simulation. [KN02] and [CK05] use wisps models with very good results. Both papers describe hair creation for offline rendering, but the concepts can be adjusted to suit real-time game purposes; a wisp of hair strands can simply be modeled by a triangle strip.

Even if the hair creation is reduced to clusters of hair, wisps, it can still be a tedious and time consuming work to manually create them. Wisps are therefore often generated automatically by some user defined input. [KN02] use a scalp surface mesh as a base for a hair style and define larger areas (contours) on the scalp where a number of hair wisps should be generated, and [CK05] define a global number of wisps and use textures to specify the distribution at different parts of the scalp.

Since wisp positions are generated it is important to distribute the wisps evenly over the scalp surface to prevent unwanted chunks and noise patterns. There are several techniques to ensure even distributions, such as Poisson disk distribution or Relaxation methods. [KN02] used a method called position relaxation. For real-time game purposes the number of wisps will be in the order of hundreds, and in addition, wisp generation is a preprocessing step. Since there is no need for a fast algorithm, a simpler but slower version of Poisson disk distribution can be used, called *relaxation dart-throwing* described by Ares Lagae and Philip Dutr´E [LD05]. This is an improved version of dart throwing. Dart throwing generates points randomly with an associated collision radius, but before a new point is accepted it must not collide with the collision radius of existing points. *Relaxation dart-throwing* starts with a large collision radius and adds the criteria that if a large number of failed attempts to create a new point occur, the collision radius is reduced by a fraction. This ensures that a certain number of points will be found on a surface.

## 2.3 Strands

Each wisp consists of several strands which are created from a set of parameters or constraints. Both [LD05] and [KN02] look at a wisp as a generalized cylinder. This facilitates the definition of the wisps 3D space where the strands exist. [LD05] use a model where they use several functions to define the attributes of a wisp, i.e. the strands inside the wisp, called member strands. These functions assume that a master strand has been defined/modeled and attributes such as length, distribution, density and curliness are defined as deviations from the master strand.

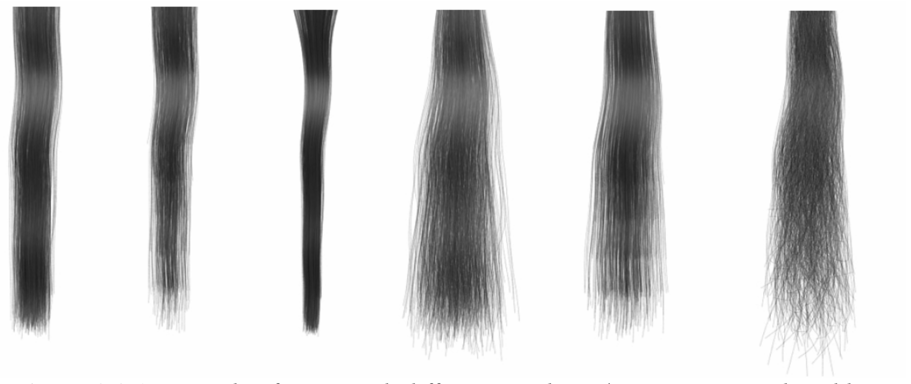


**Figure 2.3.1** *Example of wisps with different attributes/parameters produced by [LD05]. Image courtesy of Choe et al. [LD05]*

As described earlier, it is not suitable to model each strand in games where rendering speed is vital. Instead, it is natural to model a wisp as a polygon strip. Individual strands could then be generated in textures which are mapped onto the polygon wisps. By limiting strands to texture space they are reduced to two-dimensional objects, facilitating simpler strand generation, but reducing realism. Because of this, the shape of the triangle strip has to provide the wisp's significant three-dimensional properties. More subtle details can be modeled with techniques such as normal mapping. Since games often use textures and normal maps to model details the reduced realism will in this case not be noticeable. The downside of this model is that it is not suitable for curly hair because a large amount of polygons would be needed and a lot of artifacts would occur due to intersecting polygons. However this would have been a problem for lines as well considering games, since a lot of line segments are required to produce curly shapes.

## 2.4 Scalability

When viewed at a distance it is hard to see details not only at strand level but also at wisp level. Using triangle strips and textures as wisps facilitate level of detail creation. When viewed from a distance reducing the number of wisps and at the same time increasing their width will produce better rendering performance without compromising visual quality. The graphics cards ability of generating mipmaps will provide level of detail at strand texture level.

## 2.5 Modeling and Updating

As previously mentioned, even if the hair creation is reduced to wisps it is still time consuming to manually create them. Considering that the CPU is often the bottleneck in games, simulation-wise, it would also be of interest to create a higher level description of hair style to reduce the amount of simulation data (if simulation should be done using the CPU at all).

One solution is to create a three-dimensional “field” which defines how hair wisps should grow. There are two major methods; force fields and paths. Yu [Y01] uses a selection of vector field primitives in which the hair can grow. [CK05] use paths to create user-designed vector force fields and at the same time takes other force fields in consideration such as gravity and wind. Both papers produce very good results. However these methods are not suited for fast updates since they require a lot of calculations and can also be hard to simulate movement with. A solution to this could be to let the vector field generate simulation data. Another solution could separate modeling and simulation completely and try to reduce simulation data to avoid simulating every wisp. Simulating every wisp would be computationally too heavy for the CPU and the simulation has to be carried out by the GPU. Some computations can be hard to map onto the GPU and depending on the complexity of the simulation, hair can be one of them.

The second method for defining hair growth is to build a low resolution version of a hair style with paths/strands. [KN02] use this technique and let the user create a path for each contour with good results. This method requires more work by the user (placing 25-50 strands) but in return the same strands can be used for simulation. [ND05] used a low resolution hair style model as simulation data and recreate/update the full hair style from this in real-time. They place their strands at each vertex (forming a triangle space) and interpolate new ones on each triangle using barycentric coordinates, computed entirely on the CPU. [ND05] use a large data set for simulation/modeling; approximately 750 strands with seven points each, which makes it very hard for them to create them manually (they let the simulation sculpt the hair). For games though, 750 strands is too much data to simulate. However the number of strands and strand points can be reduced to gain performance without loosing too much visual quality. Furthermore, as explored in this thesis, the interpolation of new strands can be moved to the GPU to significantly increase performance. This does not only take advantage of the fact that the GPU can handle more floating point operations than a CPU, but also benefits from interpolation which is provided for free since the input data on the GPU is stored as textures.

For the scope of this thesis, it was decided that building a low resolution version of a hair style with paths/strands should be used since Autodesk Maya or similar software can be used to model splines. Force fields or similar solutions would require the creation of separate modeling software. Moreover, a part of this thesis lies in finding a suitable method for hair simulation. Bearing this in mind it would be useful to have a low resolution representation for simulation purposes.

# 3 Visualization

## 3.1 Transparency

Rendering triangle strips (polygons) with textures which are opaque requires use of alpha channels in the texture to mask out the hair strands. A solution that has been used for rendering vegetation, which faces a similar problem as hair rendering do, is alpha testing. Alpha testing tests a pixel's alpha value against a reference value and renders/discards the pixel depending on the result. In real life hair stands are very thin. It would not be a good idea to mask out a single hair strand at a normal viewing distance because it is smaller/thinner than a pixel on the screen. This makes simple alpha testing a bad solution for hair rendering as it works on a per-pixel level. Since hair strands are very thin only a few strands create a transparent effect while many strands create an opaque surface. Therefore transparency can be used to simulate these effects.

### 3.1.1 Blending

Transparency is best handled with blending which mixes the current pixel with the underlying pixel by a certain amount. The draw back is that it is often very slow and rendered objects need to be sorted from back to front. In special cases it is possible to dynamically sort all wisps, but often the movement and shape of the hair makes it impossible. In [S04] they constrain the movement and shape of their hair in such a way that they could use a static sorting and therefore enable blending.

### 3.1.2 Alpha to Coverage

In 2005, NVIDIA published [NV05] a new method describing how to use graphics cards multisampling ability simulate transparency without needing to sort geometry. The method is called *Alpha to Coverage*. It computes how many of a pixels sub pixels (multisamples) that should be filled given an alpha value. Finally all sub pixels are averaged to a final color. The method is proven to produce good results with smaller areas where the alpha goes from opaque to transparent. Rendering large areas of semi-transparent surfaces can produce dithering aliasing. Hair styles are examples of objects that have alpha gradients ranging from opaque to transparent.

## 3.2 Shading Models

Simulating materials in the world requires calculation of how much light bounces/reflects of objects given an amount of incident light. This is called a local reflectance model. Local means that only local surface parameters are known. Hair has an anisotropic material, in contrast to an isotropic material, modeled for example with Phong. An anisotropic material often reflects more light in the direction of fibers, in computer graphics this is referred to as the tangent. Hair reflects more light in the direction of the strand. A model that implements this and has been widely used since it was introduced in Kajiya's reflection model [KK89].

### 3.2.1 The Kajiya Model

Kajiya [KK89] approximate a hair strand with a small smooth cylinder. The model consists of two components; a diffuse and a specular. The diffuse light is computed by integrating a Lambert surface model (cosine falloff function, $n \cdot l$) over the half circumference visible from the light. The circumference normal is expressed using basis vectors $b$ and $l'$, parameterized by an angle $\theta$, which is the angle between $b$ and the circumference normal.

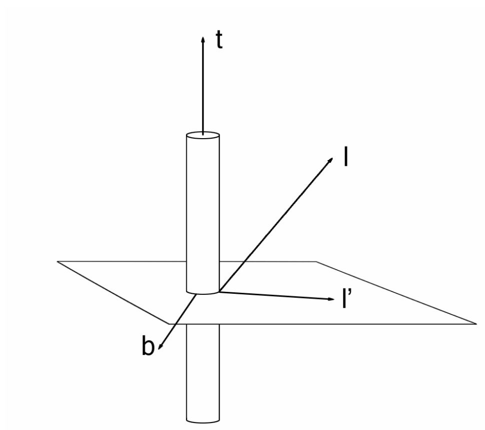


**Figure 3.2.1** *Vectors used by Kajiya-Kay model*

$$n = b\cos(\theta) + l'\sin(\theta)$$

$l'$ is computed by projecting the light vector *l* onto the plane perpendicular to the hair strand.

$$l' = \frac{l - (t \cdot l)t}{\left\| l - (t \cdot l)t \right\|}$$

*b* can then easily be computed by taking the cross product of *t* and $l'$. The integral can then be expressed by:

$$L_{diffuse} = k_d r \int_0^{\pi} (n \cdot l) d\theta = k_d r \int_0^{\pi} ((b \times \cos(\theta) \times l'\sin(\theta)) \cdot l) d\theta$$
$$= k_d r (l \cdot l') \int_0^{\pi} \sin(\theta) d\theta = K_d \left( l \cdot \frac{l - (t \cdot l) * t}{\left\| l - (t \cdot l) * t \right\|} \right) = K_d \sin(t, l)$$

Here $(t, l)$ refers to the angle between *l* and *t*. The specular term is modeled by assuming that incident light scatters at mirrored angles along the tangent, creating a cone of reflected light. Specular highlights are then calculated with Phong dependence between the eye vector and reflection vector.

$$L_{specular} = k_s \cos^p(\theta_r - \theta_e) = k_s (\cos(\theta_r)\cos(\theta_e) + \sin(\theta_r)\sin(\theta_e))^p$$
$$= k_s ((t \cdot l)(t \cdot e) + \sin(t, l)\sin(t, e))^p$$

Final intensity is computed by adding the diffuse and specular terms. This model produces acceptable results for a small amount of calculations, but does not capture the characteristics of human hair. In 2003, Marschner et al. [MJC03] introduced a new model based on a more physically correct light scattering.

## 3.2.2 The Marschner Model

Marschner's [MJC03] model does not limit itself to just the antistrophic reflection (Kajiya [KK89]) but also takes subsurface scattering into account. He models a hair strand as a translucent cylinder and the reflection model consists of three major components/light paths: Reflection (R), Transmission -Transmission (TT) and Transmission-Reflection-Transmission

(TRT), each of them adding a specular component. Each component is a four-dimensional scattering function. Marschner [MJC03] describes how this can be reduced to a multiplication of two two-dimensional functions. This opens up the possibility to store the two functions pre-calculated in textures, Nguyen [ND05]. This approximation can be done because the symmetry of a cylinder. The two-dimensional functions capture the angle dependence perpendicular to and the angle dependence coplanar with the hair strand, denoted *M* and *N*. The scattered light *S* is then expressed by (spherical coordinates):

$$S(\theta_i,\theta_r,\phi_i,\phi_r) = M(\theta_i,\theta_r) \times N(\phi_i,\phi_r) / \cos^2\theta_d$$

$\cos^2\theta_d$ accounts for the projected solid angle, and $\theta_d$ is the difference between $\theta_i$ and $\theta_r$. The angles indexed by $i$ are the incoming light direction and angles indexed $r$ are the outgoing (eye) direction.

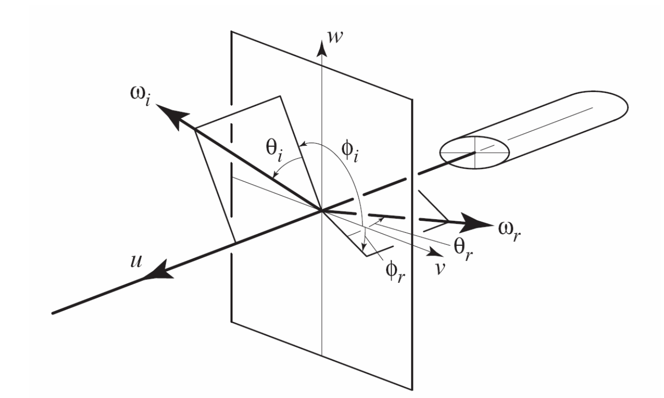


**Figure 3.2.1** *Notation of scattering geometry*
*Image courtesy of Marschner et al. [MJC03]*

Let $M(\theta_i,\theta_r)$ be the longitudinal dependence, coplanar with the strand. Marschner et al. [MJC03] approximate the rough interface of a hair strand, creating random deviation from the predicted direction. The result is a smoothened scattering. Another characteristic of hair that makes them deviate from the cylinder model is its cuticle scales of the silhouette, creating a shift of the scattering lobes, as seen in figure 3.2.2.

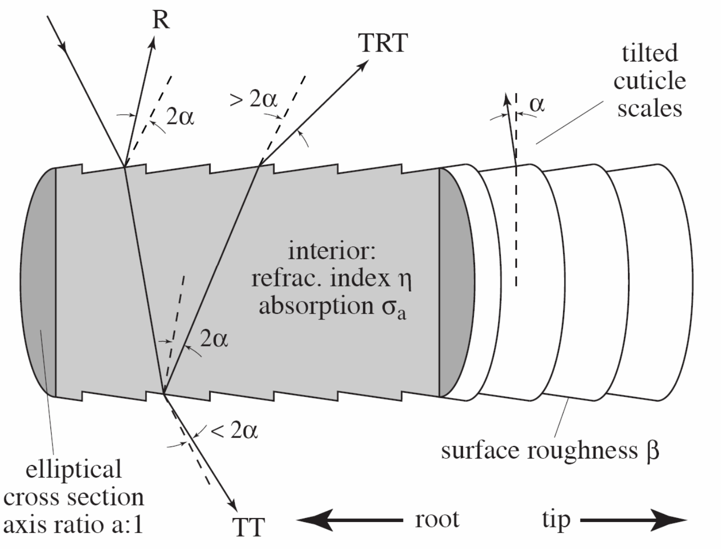

R
TRT
tilted cuticle scales
2α
> 2α
α
interior: refrac. index η absorption σa
2α
< 2α
elliptical cross section axis ratio a:1
surface roughness β
TT
root
tip

**Figure 3.3.2** *Hair strand characteristics affecting the longitudinal dependence (M). Image courtesy of Marschner et al. [MJC03] Light scattering of human hair fibers*

*M* is modeled (empirical results, Marschner et al. [MJC03]) by a Normalized Gaussian function $g(\beta_{stdev}, \theta - \alpha_{shift})$. The standard deviation is altered for the different reflections. The angle-parameter $\theta$ which represents the half angle $(\theta_i + \theta_r)/2$ is shifted to simulate the hairs' cuticle scales. Mean value for the function is zero.

$$M_P(\theta) = \frac{1}{\beta\sqrt{2\pi}} e^{\frac{-(\theta-\alpha)^2}{2\beta^2}} \quad \text{where } P \in \{R, TT, TRT\}$$

*Suggested values of parameters by Marschner et al. [MJC03]:*

| | | |
|---|---|---|
| $\beta_R$ | Standard deviation: R | $-10^\circ$ to $-5^\circ$ |
| $\beta_{TT}$ | Standard deviation: TT | $\beta_R/2$ |
| $\beta_{TRT}$ | Standard deviation: TRT | $2\beta_R$ |
| $\alpha_R$ | Cuticle Shift: R | $10^\circ$ to $5^\circ$ |
| $\alpha_{TT}$ | Cuticle Shift: TT | $-\alpha_R/2$ |
| $\alpha_{TRT}$ | Cuticle Shift: TRT | $-3\alpha_R/2$ |

Let $N(\eta', \phi_i, \phi_r)$ be the latitudinal dependence, perpendicular with the hair strand. This function is modeled by analyzing a two dimensional slice of a hair strand. *N* depends only on the difference of $\phi_i$ and $\phi_r$ and the refraction index $\eta$. The fact that it depends on the difference of $\phi_i$ and $\phi_r$ actually reduces the function to two parameters $N(\eta, \phi_i - \phi_r)$. Due to the two dimensional slice model the refraction index $\eta$ must be recalculated to account for the change of incident vector when projecting it to the slice perpendicular with the hair strand. This is called the effective index of refraction denoted $\eta'(\theta_d)$ and is computed by:

$$\eta'(\theta_d) = \sqrt{\eta^2 - \sin^2\theta_d} \Big/ \cos\theta_d \rightarrow N(\eta', \phi_i - \phi_r)$$

This makes *N* a function of theta and phi, $N(\eta(\theta_d),\phi_i - \phi_r) = N(\theta_d, \phi_d)$. By examining light reflection from a circular cross section of a hair strand, three aspects have been considered by Marschner et al. [MJC03]. The distance light travel inside the hair, absorbing light. The amount of reflected/refracted light at the intersection points, calculated by fresnel equations. Calculation of the incident light paths *h* given an exit angle $\phi$ (difference between light and view angle) and the amount of light divergence/convergence for those paths, denoted $d\phi/dh$.

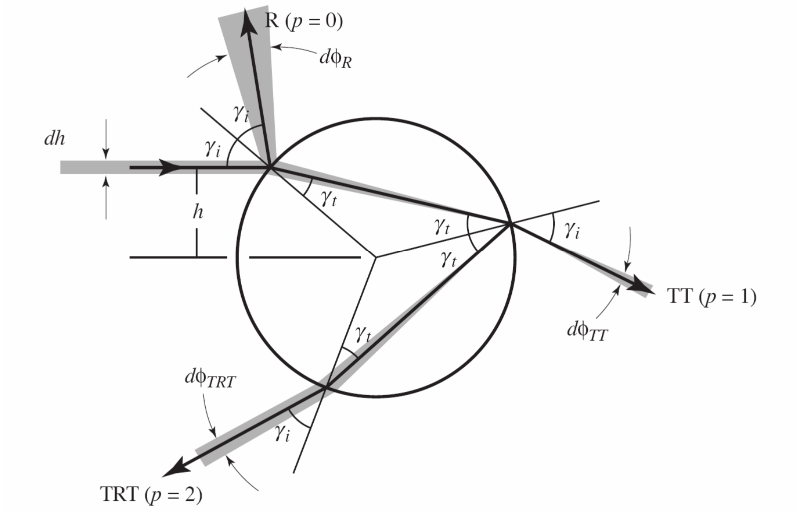


**Figure 3.2.3** *Hair strand characteristics affecting the latitudinal dependence (N). Cross-section of hair fiber and notation for light paths and angles. Image courtesy of Marschner et al. [MJC03]*

When a parallel light beam enters a hair fiber at a specific height from the centre it is reflected/refracted according to Snell's law. As mentioned before this model includes the three first reflections/refractions. These can be expressed by the following formula where *p* denotes the different paths shown in the figure above.

$$\phi(p,h) = 2p\gamma_t - 2\gamma_i + p\pi$$

$$\sin\gamma_i = h\,,\ \eta'\sin\gamma_t = h \text{ and } p \in \{0,1,2\} = \{R, TT, TRT\}$$

$$\phi(0,h) = -2\times\arctan(h)$$

$$\phi(p,h) = 2\times\arctan(h/\eta') - 2\times\arctan(h) + \pi = 2\times\arctan(h/\eta') - 2\times\arctan(h)$$

$$\phi(p,h) = 4\times\arctan(h/\eta') - 2\times\arctan(h) + 2\pi = 4\times\arctan(h/\eta') - 2\times\arctan(h)$$

The incident light paths *h* are found by finding the roots to the equation $\phi(p,h) - \phi_d = 0$ also yielding $\gamma_i$ and $\gamma_t$ which is used for further calculations. Divergence/convergence amount is then computed by differentiating $\phi(p,h)$ with respect to *h* and inserting the root values. The differential describes how much the outgoing light angle changes given a small change in *h*. According to Marschner et al. [MJC03] the radiance emitted from a hair cross section can be expressed by:

$$L(p,h) = A(p,h)\left|2\frac{d\phi}{dh}\right|^{-1} E$$

*E* is curve irradiance, *L* is curve radiance and $A(p,h)$ is the attenuation term for volume absorption and Fresnel reflection. Volume absorption is expressed by:

$$T(\sigma_a, h) = \exp(-\sigma_a(2r\cos(\gamma_t)))$$

$2r\cos(\gamma_t)$ is the internal length of a light beam through a hair strand, $\sigma_a$ is the absorption per unit length and *r* is the radius. The Fresnel term F can be computed with the usual Fresnel equations (polarized equations) as described by Dutré [DBK03]:

$$F_\perp = \frac{\eta_2\cos\gamma_i - \eta_1\cos\gamma_t}{\eta_2\cos\gamma_i + \eta_1\cos\gamma_t} \text{ and } F_\parallel = \frac{\eta_1\cos\gamma_i - \eta_2\cos\gamma_t}{\eta_1\cos\gamma_i + \eta_2\cos\gamma_t} \text{ where } \eta_1 = 1.0$$

$\eta_2$ is replaced with an effective refraction index

$$\eta'(\gamma) = \sqrt{\eta^2 - \sin^2\gamma}/\cos\gamma$$

for $F_\perp$ and

$$\eta''(\gamma) = \eta^2\cos\gamma/\sqrt{\eta^2 - \sin^2\gamma}$$

for $F_\parallel$. For non polarized light the Fresnel term is

$$F = \frac{|F_\perp|^2 + |F_\parallel|^2}{2}.$$

These computations will be denoted $F(\gamma, \eta)$. A general attenuation term for all reflections can be written as:

$$A(p,h) = \begin{cases} F(\gamma,\eta) & p = 0 \\ (1 - F(\gamma_i,\eta))(1 - F(\gamma_i, 1/\eta))F(\gamma_t, 1/\eta)^{p-1}(\exp(-\sigma_a(2r\cos(\gamma_t))))^p & p \in \{1,2\} \end{cases}$$

#### 3.2.2.1 Complete Model

A complete model for Marschner et al. [MJC03] hair shading can be expressed by:

$$S = \frac{\frac{1}{\beta\sqrt{2\pi}} e^{\frac{-(\theta-\alpha)^2}{2\beta^2}} \times \sum_h A(p,h)\left|2\frac{d\phi}{dh}\right|^{-1}}{\cos^2\theta_d} E$$

$$A(p,h) = \begin{cases} F(\gamma,\eta) & p = 0 \\ (1 - F(\gamma_i,\eta))(1 - F(\gamma_i, 1/\eta))F(\gamma_t, 1/\eta)^{p-1}(\exp(-\sigma_a(2r\cos(\gamma_t))))^p & p \in \{1,2\} \end{cases}$$

$$F(\gamma,\eta) = \frac{|F_\perp|^2 + |F_\parallel|^2}{2}; F_\perp = \frac{\eta_2\cos\gamma_i - \eta_1\cos\gamma_t}{\eta_2\cos\gamma_i + \eta_1\cos\gamma_t}, F_\parallel = \frac{\eta_1\cos\gamma_i - \eta_2\cos\gamma_t}{\eta_1\cos\gamma_i + \eta_2\cos\gamma_t}$$

$\gamma_i$ and $\gamma_t$ is calculated by Snell's law and trigonometric equations using *h* solved by $\phi(p,h) - \phi = 0$ where $\phi(p,h) = 2p\gamma_t - 2\gamma_i + p\pi$.

This physically more correct model has been used in several publications published after Marschner et al. [MJC03]. Both NVIDIA and ATI have implemented these phenomena in their latest demos showing that this produces better results than Kajiya [KK89].

## 3.3 Self Shadows

As previously discussed hair is a translucent material, but still it absorbs the light to a certain extent. This prevents the light from lighting underlying hair strands and near the roots, creating a self shadowing effect. This can not be modeled by the shader model due to the limitation of local information. This effect depends on the surrounding (global) hair geometry and light sources. Therefore it is necessary to recalculate the solution every time the hair or the light has moved for a correct representation. This can be very computational intensive since ray tracing each light is needed. Instead a pre-computed diffuse convoluted environments map describing distant light incident from all directions at each point, is used. The technique can be combined with a virtual light source creating a global shadow direction or one or two local light sources creating local shadows or illumination. This setup suggests having a combination of a static and a dynamic solution for self shadowing assuming that hair strands/patches do not move much relative to each other.

### 3.3.1 Static

The static solution is solved by ambient occlusion. But instead of using the traditional approach which only works for patches, Pharr [PF04], casting rays and checking for occlusion a radiosity inspired solution is used. This method, described by Bunnell [B05], casts shadows from each vertex to each other vertex approximating the area of the shadow casting polygon by a disc perpendicular to the surface normal. Our approach would approximate every shadow casting vertex by a transparent cylinder aligned with the hair strand. The radius of the cylinder would represent the width of a hair strand or hair wisp/tri-patch. By using a cylinder the area of the shadow caster can always be expressed as a rectangle $A = 2r \cdot l$ where *r* is the radius of the cylinder and *l* is the length.

The amount of shadow (*S*) cast by a hair section on a vertex is approximated by:

$$S = \sum_{vertices} \frac{\psi_{hairstrand} \cdot \sin\left(T_r, \bar{d}\right) \cdot \sigma}{\psi_{sphere}}, \; \psi_{sphere} = 4\pi$$

The ratio $\psi_{hairstrand} / \psi_{sphere}$ is the solid angle of a shadow casting hair cylinder, at distance *d,* divided by the solid angle of the entire sphere. The term $\sin(T_r, d)$ accounts for the angle between the distance vector $\bar{d}$ and the tangent of the receiving vertex, $T_r$. If the shadow caster is perpendicular to the receiving point on the hair strand the contribution is larger than it would be at small angle. $\sigma$ describes the transparency of a hair strand scaling the amount of shadow casting. The solid angle of a cylinder (rectangle when viewed from the receiver) is approximated by a disc covering the same area as the cylinder would. As described earlier the area is calculated by $A = 2r \cdot l$. However one has to scale this with respect to the angle between the receiving vertex and the tangent of the shadow caster, $T_c$, yielding $A = 2r \cdot l \cdot \sin\left(T_c, \bar{d}\right)$. The solid angle of a disc is, [DBK03]:

$$\psi_{disc} = 2\pi(1 - \cos(\phi))$$

Where $\phi$ is the angle between the centre of the disc and the rim, which can be expressed by the distance between receiver and caster and the radius of the disc, as follows:

$$\phi = \arccos\left(\frac{d}{\sqrt{R^2 + d^2}}\right)$$

$$\psi_{disc} = 2\pi\left(1 - \frac{d}{\sqrt{R^2 + d^2}}\right)$$

Where *R* is the radius of the disc calculated by the visible area of the strand cylinder.

$$R^2 = \frac{A}{\pi} = \frac{2r \cdot l \cdot \sin(T_c, d)}{\pi}$$

Inserting this into the final equation will result in:

$$S = \sum_{verices}\left[\left(1 - \frac{d}{\sqrt{\frac{2r \cdot l \cdot \sin(T_c, d)}{\pi} + d^2}}\right) \cdot \sin(T_r, d) \cdot \sigma \cdot \frac{1}{2}\right]$$

### 3.3.2 Dynamic

The dynamic solution can be solved with several techniques. The general idea is to create a depth function describing the density/transmission throughout the hair volume from the light source. Currently the most interesting solution is opacity shadow maps described by Kim [KN01], and later improved by [ND05] to take advantage of today's graphics hardware.

Opacity shadow maps use a set of parallel uniformly spaced opacity maps perpendicular aligned to the light direction to approximate the amount of light penetrating to depth *z*.

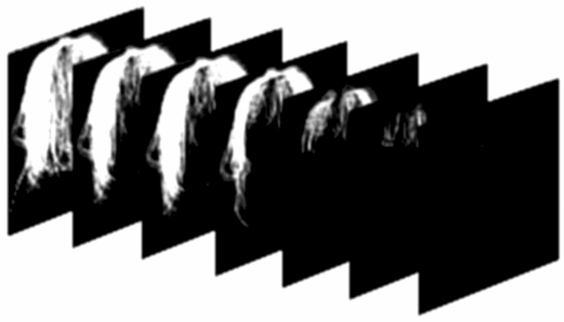


**Figure 3.3.1** *Opacity shadow map layers*

This is expressed by the following function:

$$T(z) = e^{-k\sigma(z)} \quad (3.1)$$

$$\sigma(z) = \int_0^z \alpha(z')dz' \quad (3.2)$$

*T* is the amount of light transmitted, *k* is an extinction coefficient chosen such that when $\sigma = 1$, $T \approx 0$. To evaluate $\sigma(z)$ at depth *z* from the light, discrete sample planes (Opacity shadow maps) of the function are rendered, $\sigma(z_i)$.This is done by rendering the hair opacity with additive blending, clipping the strands further away than depth $z_i$. To evaluate the transmission for a point at an arbitrary depth *z*, interpolation of order *n* is performed using the previously rendered opacity shadow maps. [ND05] showed how this could be implemented on modern graphics cards using four render targets, each rendering to a four-channel texture outputting 16 opacity shadow maps in one pass, using shaders for clipping. The actual interpolated value is then assembled in a shader as well. Furthermore, advancements in next generation hardware and DirectX 10 will make it possible to increase performance of opacity shadow maps greatly by reducing the memory bandwidth of the algorithm.

There has also been an attempt to create a similar depth function but refining the solution creating an adaptive slicing of discrete opacity samples [MKBR04]. This starts by rendering uniform slices (clusters) yielding the mean z-value for each slice representing one iteration in the k-means algorithm [HW79]. This could be refined by further iterations/renderings of k-means, but Mertens et al. [MKBR04] states that one iteration produces sufficiently good results. Now instead of producing opacity shadow maps at the new depth positions the standard deviation of fragments for each cluster is calculated in the second pass. These values are then used to create depth bins indicating the length of density increase which is used to compute the transmission value *T*.

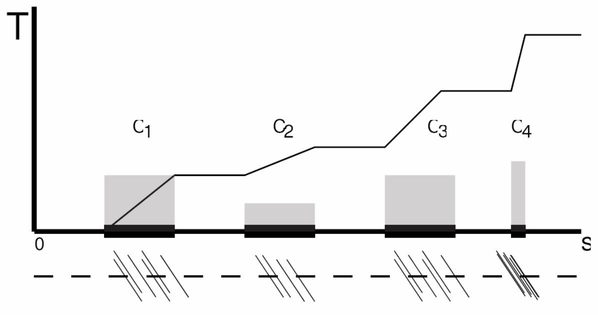


**Figure 3.3.2** *Image courtesy of Mertens [MKBR04]*

This allows a more flexible depth function producing better results with fewer sampling planes. However since it is possible to render all opacity shadow maps in one pass efficiently on next generation hardware, this method is less efficient since it uses multiple passes. Due to the structure of graphics card pipelines it is often a problem, sometimes even impossible, to read and write to same data/texture in the same pass using the GPU. If this were to be changed more flexible and accurate calculations could be used to accumulate a depth function in one pass. One possible solution would be to use shaders for solving and updating a least square problem with QR transformations.

There has also been an attempt to create these depth functions using the CPU [BMC05]. The solution uses a lattice that encompasses the hair at all times. This lattice is always oriented so that it faces the light. The solution is essentially a four pass solution where the passes are:

1. Filling of density map
2. Computation of transmittance

3. Filtering
4. Drawing

In the first step the lattice is filled with values corresponding to the density of hair in each cell. Then, the density values are transformed into transmittance values and a ray is traced through the lattice from the lit side to the back, column wise. To accomplish this, the incoming light rays are assumed to be parallel, which implies a light source at infinite distance. While tracing the transmittance value is decreased for each cell. The result is then filtered to reduce artifacts before drawing. This solution is rather slow even if it can be parallelized to achieve higher frame rate on multi core system. The fact that it is implemented entirely on the CPU does not make it suitable for games since the CPU often is heavily occupied with game play and physics. Typically you would like to use the GPU as much as possible since it has far more processing powers and is extremely parallelized. The second step, which increments a value for each traversed cell, is not immediately suitable for GPU solutions as it requires the previous value computed in the same pass.

Unfortunately the above methods for dynamic shadows can be computationally heavy for games since they derive their depth shadow data from three-dimensional shadow volumes. Both creating and reading from these volumes can be slow given the available computational resources. Since they also require a per object shadow solution (each hair has to have a separate shadow volume) the techniques are limited to self shadowing, including its head/character.

Opacity shadow maps present high quality shadows with reasonable performance. This might be good for character close-ups in cut-scenes or similar situations, but what is often sought is reasonable quality at high performance. Characters in games often move fast and, as we will see in the results chapter, the size of the hair on screen is often fairly small. In settings such as these, the fidelity of opacity shadow maps might not be required. To alleviate some of the performance issues associated with opacity shadow maps, and to give an alternative, a new simple technique is presented. The transmittance function from opacity shadow maps above is adapted to suite the new method.

$$T(z) = e^{-k\Delta z}$$

The hair is rendered as opaque (with alpha test) to one depth buffer from the light. When the geometry is rendered to the screen, the depth of the drawn pixel in light space is compared with the depth in the generated depth buffer. If the depth is greater than the depth in the buffer the difference $\Delta z$ is used as input into the above formula to approximate the transmittance. The simplification made from opacity shadow maps is that the opacity is approximated as a linear function of the distance the light travels through the hair. Most hairstyles, at greater than close distance, can be seen as a uniform convex shape, for which this approximation is intended to work well. It is not as common that some part of the hair is greatly separate from the rest and covering another part with void separating them. At very close distances the latter becomes more apparent though, as the fine detail of hair is revealed.

# 4 Simulation

In computer graphics and in games in particular the desired outcome is most often something that appears real to the beholder. For hair, unless the algorithms are applied in a scope where physical correctness of the simulation is crucial, the end result is something which does not impact anything other than the visual appearance. When the goal is realism only in the visual sense the possibilities of shortcuts are endless. The end result is often a compromise between a visually pleasing result and computational cost; physical correctness does not score high on the list of priorities.

A few basic observations can be made about the characteristics hair has when moving. First, hair has a tendency to return to a position and shape in space. This is particularly important for applications in games and other interactive environments, most game designers will agree they do not want a long hairstyle getting messed up or covering the face after a few moments of intensive movement. However, this is basically only true if the owner is in an upright position. If the hair is at rest but tilted for instance, it would be unnatural for it to return to a predefined shape. Second, when moving freely, hair does not act with noticeable elasticity. It can be assumed that hair strands keep a fixed length. Third, hair has the ability to move relatively freely when under the influence of force or when the object it is connected to moves. As long as it adheres to the first observation this is fine. Finally, hair, of course, has inertia and acts with high friction against airflow with regard to its weight.

Thus, the basic criteria for simulating hair with different hair styles in mind should be (in order of priority):

1. The ability to return to a predefined position at rest
2. Fixed length
3. Free movement
4. Inertia
5. Interaction with outer forces

If no constraints or memory of position are required, simulating a strand is reduced to simulating nodes connected as a chain, with a fixed distance between them. This is easily accomplished using Verlet (or backward Euler) integration [J67], with one or a series of correction iterations to keep the length of the links close to a defined value.

The Verlet integration method is an improvement on Euler integration to allow for better stability. The formula can be derived using an approximation of $P(t+\Delta t)$ and $P(t-\Delta t)$ by the first terms of their Taylor expansion, where $P(t)$ is a position at time *t*:

$$P(t+\Delta t) \approx P(t) + \Delta t P'(t) + \Delta t^2 P''(t)/2$$
$$P(t-\Delta t) \approx P(t) - \Delta t P'(t) + \Delta t^2 P''(t)/2$$

Adding the above equations gives:

$$P(t+\Delta t) = P(t) + k(P(t) - P(t-\Delta t)) + \Delta t^2 F(t)/m \quad (4.1)$$

where $P''(t) = mF(t)$ and a dampening coefficient *k* has been added. This works best with a fixed time step $\Delta t$. The new positions $P(t+\Delta t)$ may be conflicting with the second proposed criteria for hair listed above. To remedy this, one or a series of correction steps are taken. Formally the constraint in question can be described by:

$$\|P_n - P_{n-1}\| = d_n$$

At each correction iteration, every pair of nodes $P_n$ and $P_{n-1}$, which are connected are displaced by

$$a\ (1-\frac{d_n}{\|P_n - P_{n-1}\|})(P_n - P_{n-1}) \quad (4.2)$$

Where *a* is:

| | |
|---|---|
| (1.0, 0.0) | if $P_{n-1}$ is fixed |
| (0.0,-1.0) | if $P_n$ is fixed |
| (0.5,-0.5) | otherwise |

for $P_n$ and $P_{n-1}$ respectively. Done repeatedly, the positions converge to a state where all constraints are met, see [J01] for details.

## *4.1 Adaptive Wisp Tree*

[BKNC03] observe the characteristics of hair motion. Due to friction and self collision hair often groups into clusters and strands in close proximity largely move in the same direction and speed. Furthermore, cluster splitting often occurs from tip to root and merging occurs in the other direction due to increasing constraints towards the scalp. An approach is presented by [BKNC03] aimed at reducing the calculation time while keeping or improving the quality of previous work. A tree structure, *Adaptive Wisp Tree* is presented to adapt the resolution of the simulation locally and globally. The tree is built from a hierarchy of *generalized cylinders (GC)* consisting of one or more parallel segments. The number of segments in a *sub-GC* always is a multiple of the number of segments in a *super-GC*, as shown in figure 4.1.1.

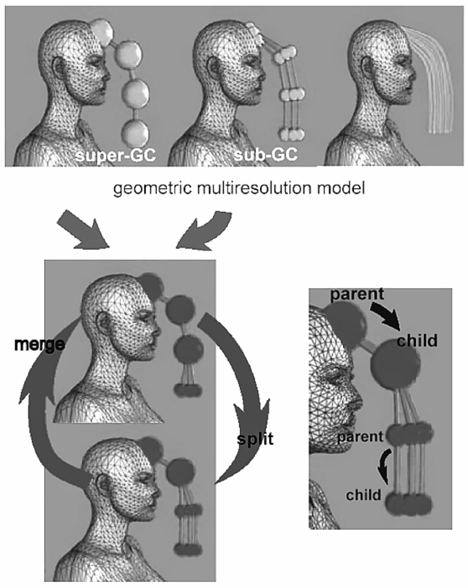


**Figure 4.1.1** *[BKNC03] Visualization of the Adaptive Wisp Tree (AWT) structure.*

A node in the model is defined as the centre of a cross-section, connecting two levels in the tree structure, shown as green and yellow spheres in figure 4.1.1. A parent GC or parent node in the structure is defined as a node or GC higher up (towards the root), and a child as a node or GC lower (closer to the leaves). A GC in the tree may be split into more GCs while keeping the parent GC. This is analogous to the behavior of nodes in the structure. If a GC splits, the corresponding node towards the child end of the GC splits. A split may only occur if child nodes have split into at least the same number of nodes as the impending split (and some additional criteria are met, see below). At creation time each strand of the geometric model is connected to one of the GCs at the finest detail level, this ensures that no simulation artifacts occur when nodes split or merge. Further, at each node a pointer to the nodes in the super-GC and sub-GC at the same level in the tree is stored; these nodes are called super-nodes and sub-nodes respectively. Every node has a radius and a mass based on the number of visual hair strands connected to it. In its relaxed state all cylinders are merged into their super-GCs. An “active” node is a node that is currently part of the simulation, in other words a node that does not rely on its super node for simulation.

A segment in the tree is represented as either a rigid link or a viscous spring, since some hair styles, such as curly hair behave much like springs and others, like straight hair, is better approximated as a chain of rigid links. The observation is made that hair motion becomes more complex where acceleration is great. To increase simulation resolution where more detail is needed, a node splits if the following criteria are met:

1. The acceleration times the radius is larger than a defined value.
2. The node’s child has split.

Merging occurs between nodes with the same parent node if the following criteria are met:

1. The magnitude of the relative velocity of the merging nodes is below a certain value.
2. The merging modes are within the radius of the super node.
3. The acceleration times the radius of the super node is not over the defined value (inverse of splitting condition 1.)

When merging occurs the position of each sub-node is updated as an offset vector to the position of the node. The radius of the super node is set to the smallest radius that contains all merging nodes. The position of the super node is computed as the centre of mass of the merging nodes and the velocity to the mean mass-weighted velocity.

The body connected to the hair is represented as a number of spheres. Since the tree is represented as a set of cylinders collision detection is simple. Hair self collision is reduced to a series of cylinder-cylinder intersection tests. If the closest distance of two lines representing the centers of two cylinders, is smaller than the sum of their radii there is a collision. To further improve performance and to reduce artifacts, collision detection is only done between segments with relative velocity over a certain magnitude. Active segments are stored in a voxel grid. An anisotropic model is used for collision response. Highly viscous friction forces act upon segments with similar orientation that interpenetrate. If $P_1$ and $P_2$ represent the closest points on two lines at the centre of two cylinders, viscous friction forces acting on $P_1$ and $P_2$ are calculated as

$$V_1 = k_f(P_2 - P_1)$$
$$V_2 = -V_1$$

Since the desired effect of the viscous forces is to reduce the relative speed between the two segments, and since the mass of the nodes depend greatly on what level of the GC-hierarchy they are connected to, the friction coefficient has to be adapted depending on the mass of the node. If this is not done the viscous forces may in some cases change the orientation of the segments, which is not desired. To prevent hair strands of different orientation to penetrate, repulsion forces $R_1$ and $R_2$ are added when the angle between two segments is over a defined value

$$R_1 = k_r(r-d)\frac{P_1 - P_2}{\|P_1 - P_2\|}$$
$$R_2 = -R_1$$

The forces needed for further calculations are the ones acting on the nodes rather than the force acting on the point closest to intersection. If $N_p$ and $N_c$ are the positions of the parent and child node connected to the segment, a point on the line segment formed by the two end positions can be expressed as

$$P = uN_P + (1-u)N_C$$

The forces at $N_p$ and $N_c$ are then calculated as

$$F_P = uF$$

and

$$F_C = (1-u)F$$

respectively. The method of [BKNC03] achieves a great improvement in the frame mean time over the simple single-level model. The largest improvement presented is for a long hairstyle with 6065 nodes at the finest level, with three levels of detail, where the mean animation step time was reduced from 7.9 seconds to 0.29 seconds. The smallest improvement presented is for a short hairstyle with 5149 nodes at the finest level and two levels of detail. For this setup the mean animation step time was 3.1 seconds for the single-level approach and 0.32 seconds for the AWT approach. The measurements were made on a computer with a 1.7 GHz Pentium CPU. When many nodes are simulated at the finest level AWT should approach the same amount of calculations as the single-level simulation. For a real-time application this means that the target frame rate has to be achieved for the worst case, which is equivalent to reaching the target frame rate for the single-level simulation. Still, AWT is very scalable and can easily be adapted to use a level of detail that achieves a target frame rate.

## 4.2 An Interactive Forest

For a strand to return to a position defined in space at rest, some form of constraint needs to be introduced. It proves hard to find models that approach a hair this way. However there are similar things in nature that are frequently simulated on computers, such as trees. Much like a strand or a group of strands, trees can move relatively freely while eventually returning to a resting position, where gravity is the only force. Since hair does not branch, the tree model can be simplified by removing branching. [DCF01] present an approach to generating, animating and simulating trees and a forest. Their on-the-fly procedural generation and animation is not relevant in this scope, but the simulation is. Additionally they present methods for simulating wind force on branches.

[DCF01] position a branch relative to its parent using the parameters (h,θ,ϕ), as shown in figure 4.2.1. The reference frame of a branch consists of a z, x and y axis, where the z axis points in the same direction as the branch itself. The reference frame of the child is inherited from the parent, then translated by h along the z axis of the parent and rotated by θ around that same axis. Finally the reference frame is rotated by ϕ around the child's y axis (which has already been translated by h and rotated by θ around z).

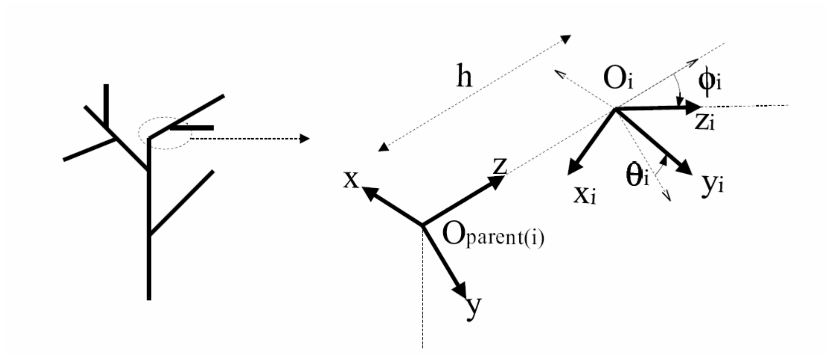


**Figure 4.2.1** [DCF01] A branch positioned relative to its parent.

Arbitrary external forces affecting a branch segment are added to a joint by approximating the torque as the cross product:

$$\tau = Lz \times F \qquad (4.2.1)$$

where L is the length of the branch segment and z its axis. These forces include all external interaction, such as wind and collisions. To approximate inertia when the root has changed direction and speed, forces are added using the same formula, either on every branch segment or alternatively only on the outermost segment. The latter works because of the downward propagation of torque from child to parent as seen in equation 4.2.3.

Joints are modeled as linear dampened angular springs with stiffness *k* and damping *v*. The torque generated by a joint is given by:

$$(k\theta + v\dot{\theta})z_p + (k\phi + v\dot{\phi})y \qquad (4.2.2)$$

Where $z_p$ is the parent z-axis and y is the y-axis of the joint. $\theta$ and $\phi$ describe the current angles relative to the resting position. $\dot{\theta}$ and $\dot{\phi}$ denote angular accelerations approximated in equations 4.2.6. Finally the total torque affecting a branch is the sum of the torque generated by external forces and the joint forces, including the joint forces of all children:

$$\begin{aligned} \tau = & \sum_{i \in external} \tau_i \\ & - (k\theta + v\dot{\theta})z_p - (k\phi + v\dot{\phi})y \qquad (4.2.3) \\ & + \sum_{j \in children} (k_j\theta_j + v_j\dot{\theta}_j)z + (k_j\phi_j + v_j\dot{\phi}_j)y_j \end{aligned}$$

Where axes without subscript denote the axes of the reference frame for which $\tau$ is calculated. The children are of course only the immediate children of the branch, which in the case of a hair strand is only one.

$\Omega$ is defined as angular velocity in world coordinates for a body, $\dot{\Omega}$ its angular acceleration and *J* its inertia matrix. To save computation time [DCF01] make the simplifying assumption $J \approx mI$ (where *I* is the identity matrix). Given this and that only rotations are considered torque can be expressed as:

$$\begin{aligned}\tau &= J\dot{\Omega} + \Omega \times J\Omega \\ &\approx m\dot{\Omega}\end{aligned} \qquad (4.2.4)$$

The relative angular velocity is derived by using the result of 4.2.4 and subtracting the parent's angular velocity. The result is then projected onto the rotational axes to obtain the angular joint accelerations $\ddot{\theta}$ and $\ddot{\phi}$.

$$\begin{aligned}\dot{\Omega}_{rel} &= \frac{1}{m}\tau - \dot{\Omega}_p \\ \ddot{\theta} &= z_p \cdot \dot{\Omega}_{rel} \\ \ddot{\phi} &= y \cdot \dot{\Omega}_{rel}\end{aligned} \qquad (4.2.5)$$

Finally Euler integration is used to evaluate the angles (analogous for $\phi$).

$$\begin{aligned}\dot{\theta}(t+dt) &= \dot{\theta}(t) + \ddot{\theta}(t)dt \\ \theta(t+dt) &= \theta(t) + \dot{\theta}(t)dt\end{aligned} \qquad (4.2.6)$$

As has been shown, this model only deals with angles, not absolute positions. Forces and torque generated by joints are used to directly calculate new angles. An advantage of this is that the length of every segment is always kept constant. Of course, when the geometry is drawn positions of the branches need to be calculated from the angles in some manner, which introduces an additional step to the algorithm.

## 4.3 Cascading Spaces

The tree model described in the previous section was never intended for use with hair, and it was never intended to have a moving connection point to the ground. Mainly the problems boil down to difficulties in simulating the effects of inertia and with adding forces corresponding to collisions, such as collisions with the head to which the hair is connected. Gimbal lock is also a frequent problem, limits can of course be set on the range of angles, but that introduces constraints not agreeable with realistic movement of hair. What we really need is a model that in itself is built to handle inertia and where hard collisions can be handled simply by displacing the colliding nodes. The model must, of course, still handle the first simulation criteria: "The ability to return to a predefined position at rest".

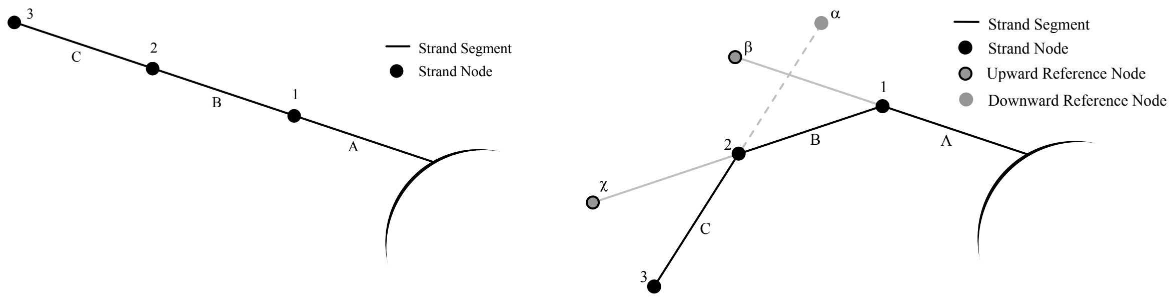


**Figure 4.3.1** *Two drawings of the same strand in different shapes. To the left, the strand is in its reference position, to the right the strand has moved out of the reference position and the reference nodes are visible.*

The model proposed in this chapter consists of a series of spaces, one for each hair strand segment. In each space, positions for two reference nodes are defined, one for the node below the first node of the segment and one for the first node above the second node of the segment. In figure 4.3.1 a strand is seen at the resting position predefined for it and in a bent position. In the bent position segment A and its second node, node 1, is in the resting position. Strand segments B and C however are not in the resting positions. The torsion spring that would be located at node 2 would affect segments B and C and thus nodes 1 and 3. Considering only segment C and the spring at node 2, the position of node 1 should be at $\alpha$, just as the position of node 2 would be at $\beta$. This is analogous for segment B, node 3 and reference node $\chi$. In this manner the torsion springs are simulated by reference nodes which positions are fixed in relation to the strand segment they are connected to, such as segment A with node $\beta$, segment B with node $\chi$ and segment C with node $\alpha$. Of course segment B would have a backward reference node connected to the strand node at the root of the hair, but since this node is fixed to the scalp this node is ignored. If there would have been a segment D however, segment C would have had an upward reference node in addition to the downward reference node $\alpha$. An upward resting position for node 1 is defined in the tangent space at the root node, but is not visible here, since node 1 is in that resting position.

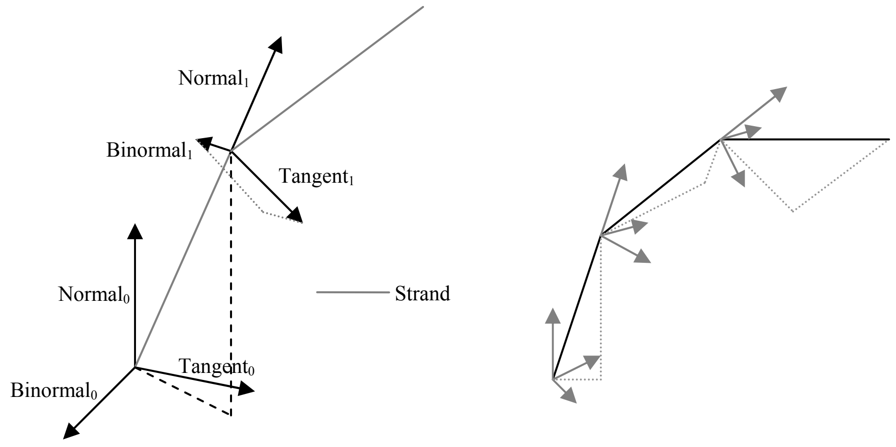


***Figure 4.3.2** (Left) A schematic drawing of the first strand segment with its tangent space.*

The algorithm functions in four steps:

1. Calculate tangent spaces
2. Set reference node positions
3. Simulate strand nodes
4. Correction step(s)

First, the current tangent spaces are calculated from the root up. A tangent space is the space defined by a normal, a binormal and a tangent; all orthogonal to each other. Such a space is defined in a segment between two nodes. In figure 4.3.2 the first segment is the grey line connecting the two tangent spaces. The normal for a segment, such as this, is defined as the directional vector of that segment, from the scalp up. Because we are dealing with line segments, there is no one way of defining the binormal and the tangent. If they were once defined, they can, as the shape rotates around two axes, eventually be any of the possible vectors orthogonal to the normal and to each other. We define first the binormal as the cross product of the current normal and the tangent from the previous segment. The tangent is then defined as the cross product of the normal and the binormal. The first tangent space for the connection point on the scalp is given.

$$N_n = \frac{P_n - P_{n-1}}{|P_n - P_{n-1}|}$$
$$B_n = N_n \times T_{n-1}$$
$$T_n = N_n \times B_n$$

Where $|P_n - P_{n-1}|$ is the magnitude of the vector $P_n - P_{n-1}$.

When the tangent spaces have been calculated, the calculation of the reference nodes becomes an operation of transforming the positions of those nodes in the local tangent spaces to world space. The reference nodes are bound to the corresponding strand nodes with elastic springs. The positions of the reference nodes are fixed during the simulation step. Below P denotes the reference point in tangent space, $O_T$ the origin for that tangent space, which is defined as the position of the second node of the segment that defines the normal of that tangent space. M is a matrix consisting of the binormal, normal and tangent as column vectors from left to right.

$$P_{world} = MP + O_T$$

$$P_{ref} = \begin{pmatrix} B_x & N_x & T_x \\ B_y & N_y & T_y \\ B_z & N_z & T_z \end{pmatrix} \begin{pmatrix} x_{ref} \\ y_{ref} \\ z_{ref} \end{pmatrix} + O_T$$

The simulation step consists of two main parts. First, the forces with which the springs exert on the strand nodes are calculated. This is a simple matter of applying the spring equation:

$$F_S = k(L - L_0)$$

We define length of the spring at rest to be zero, which reduces the above equation to

$$F_S = kL$$

The Verlet formula (4.1) is then applied to every node using the force $F_s$ as input. The resulting new positions of the nodes may be conflicting with the second criteria at the introduction to this chapter; fixed length. To remedy this one or a number of correction steps are taken as described with formula (4.2).

To make the reference position of a strand stable the reference nodes may be displaced upward to remedy the influence of gravity. This implies that $F_s$ above should be equal to the weight of a node $mg$ at the resting position, which gives that the reference node, in this state, should be displaced by

$$L = mg / k$$

If there are two reference nodes, one *upward* and one *downward* connected to a simulated strand node both of them may be displaced half the distance to introduce stability.

## 4.4 Spring Model

This model is an attempt to create the simplest possible model that would satisfy the five basic criteria that were listed in the beginning of this chapter. It is simply a series of nodes connected with fixed-length segments. Additionally each node is connected via a spring to one reference node.

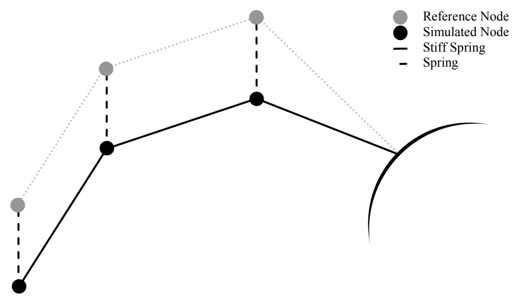


***Figure 4.4.1*** *A strand with reference positions*

Each reference node is displaced vertically from the real reference position to compensate for gravity, in a similar way to that in the previous chapter. This displaced position is then kept static in relation to the base throughout the simulation. If the resting length of the springs connecting the simulated strand nodes and the reference nodes is zero, the length of the spring at rest is given by:

$$L = mg / k$$

, which is how far the reference nodes are displaced vertically. The algorithm functions in three steps:

1. Update reference positions
2. Simulate strand nodes
3. Correction step(s)

As mentioned the reference positions are relative to the base, which is usually the head. This entails that the first step consists of a transformation from object space to world space of the reference nodes.

The simulation and the correction steps are identical to those in section 4.3.

As with the other models gravity is of course a factor in the equations. However, the spring strength and other parameters required for a natural look may reduce the influence of gravity. Since the model is designed to maintain the modelled position, at rest, this is not apparent when the head is only rotated and translated, but may be when it's tilted. To remedy this, a modification to the spring strength is done so to make it none static. As the head is tilted the spring strength is reduced by a degree dependant on the angle of the tilt.

$$k_m = \frac{z \cdot z_h + 1}{2}(k - \alpha) + \alpha$$

Here $z$ is the up vector, $z_h$ the up vector of the head. $\alpha$ is the minimum spring strength, which is the result when $z_k = -z$, this is introduced to retain some strength when the head is upside down.

# 5 Implementation

This chapter describes in more detail how the implementation of the previously mentioned ideas/techniques was made and what simplifications/modification was done. The purpose of the implementation was to create a "Hair Application" that could render a hairstyle created with an external program. The following features were implemented.

#### 5.1.1 Feature List

- **Hair Model**
  - Triangle strips represent wisps
  - Triangle strips are generated/updated by a set of simulation strands
  - Individual strands are represented by textures
  - Simulation is performed on the CPU using the simulation strands
  - The GPU is used to update and render the polygonal data
- **Modeling Pipeline**
  - Control of hair generation/simulation
  - Set of tools to facilitate modeling
  - Using textures to control hair modeling
  - Adjusting global variables
- **Simulation models**
  - Euler Angle model
  - Simple Spring model
  - Local Coordinate System Spring model
- **Physics Interaction**
  - Sphere Collisions
  - Wind forces
- **Shading Models**
  - Kajiya-Kay
  - Marschner
- **Shadowing techniques**
  - Ambient Occlusion
  - Opacity Shadow Maps
  - Modified Shadow Maps
- **Transparency**
  - Alpha to Coverage
  - Blending with sorted hair geometry

## *5.2 Developing Environment*

Given the target area for this thesis, games, the implementation was made with the most common platform/language/API used for game creation. The main program (viewer) was written in C++ with Microsoft Visual Studio. DirectX and HLSL were used as graphics and shader APIs. For hairstyle creation Autodesk Maya and MEL Scripts were used because of its simplicity and the fact that you can use many of the modeling features that Autodesk Maya has to offer.

## *5.3 Maya Pipeline*

As mentioned earlier given the chosen method of simulation it was natural to create the content in MAYA. The idea was to let the user create the low resolution model/simulation data (strand paths) and then export this to the viewer. Ideally one would write a plug-in for MAYA where the user not only can create the low resolution model, but also see the result of it while modeling as well as the possibility of tweaking the shading. For the purpose and time frame of this thesis it was decided to use MEL Scripting and write a simpler tool. I addition to the paths, custom hair creation features, such as density textures and collision spheres etc., can also be created easily with MEL Scripting. The easiest way of exporting the hairstyle to a viewer was to write a custom text based file format.

### 5.3.1 Toolset

The following tools were implemented as a shelf in MAYA, described in the same order as in the figure below.

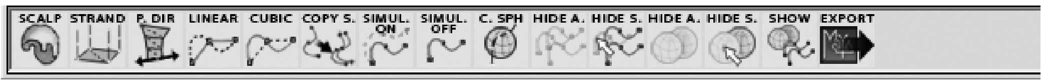
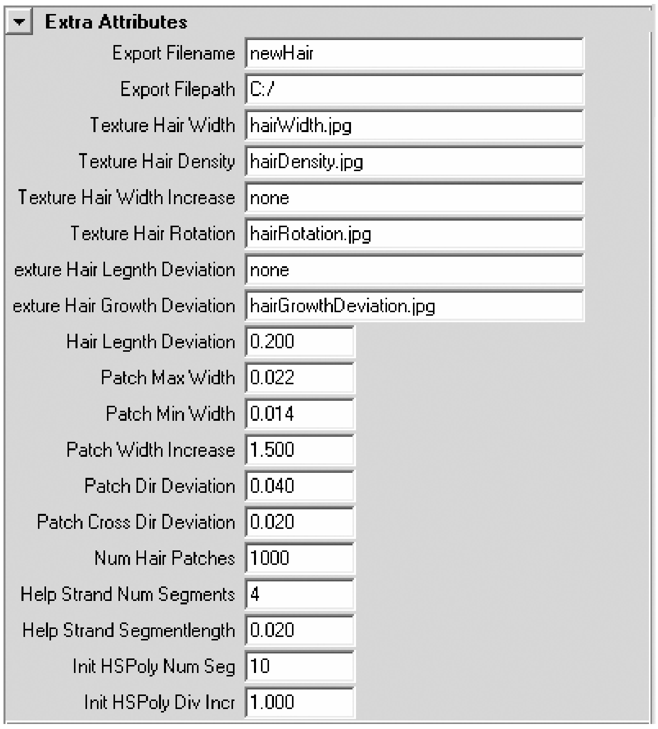


**Figure 5.3.1** *Maya Shelf containing toolset for hair creation*

- **Create scalp**
  Makes a polygonal object into a hair scalp and assigns hair specific attributes to it.
- **Create strand paths**
  Generates strand paths and their attributes in the direction of scalp vertex normals given the number of nodes in a path. In this way the user only has to edit the strand paths.
- **Assign patch directions**
  Building patches from strands requires extra information, i.e. rotation, which paths do not contain. This tool lets the user modify the scalp normals to define path rotation.
- **Cubic and linear**
  Lets the user switch degree of interpolation for the paths.
- **Copy**
  Facilitates creation of paths since strands close to each other often are very similar.
- **Simulation on and off**
  Controls if the path is going to be simulated.
- **Create collision sphere**
  Creates a collision sphere which is exported to the viewer.
- **Hide and Show**
  Makes it easy to hide and show paths so that the working area does not become confusing.
- **Export**
  Exports the hair data.

To increase control of hair generation a set of global attributes were added to a hair scalp. A variable controls how many wisps that should be generated, a texture defines the density at different areas. Wisp width is then controlled by a texture along with a minimum and maximum value. The width can also change within a wisp, increasing or decreasing to a fraction/multiple of the root width. The wisps length is defined by the paths and modified by a texture and a maximum value. To further avoid a uniform look wisp rotation can be altered to create variance and the growing path of the wisps can be modified to give a less well combed look.



| | |
|---|---|
| Extra Attributes | |
| Export Filename | newHair |
| Export Filepath | C:/ |
| Texture Hair Width | hairWidth.jpg |
| Texture Hair Density | hairDensity.jpg |
| Texture Hair Width Increase | none |
| Texture Hair Rotation | hairRotation.jpg |
| exture Hair Legnth Deviation | none |
| exture Hair Growth Deviation | hairGrowthDeviation.jpg |
| Hair Legnth Deviation | 0.200 |
| Patch Max Width | 0.022 |
| Patch Min Width | 0.014 |
| Patch Width Increase | 1.500 |
| Patch Dir Deviation | 0.040 |
| Patch Cross Dir Deviation | 0.020 |
| Num Hair Patches | 1000 |
| Help Strand Num Segments | 4 |
| Help Strand Segmentlength | 0.020 |
| Init HSPoly Num Seg | 10 |
| Init HSPoly Div Incr | 1.000 |

**Figure 5.3.2** *Attributes of a hair scalp in MAYA*

Each path also has two attributes; one to control the number of polygon segments each wisp should have, generated by the path, and one to flag the path for simulation.
Preferably strand textures should be automatically generated given some user input variables. Considering the time frame of this thesis, it was decided that strand textures would be drawn by hand. The figure below shows a modelled hair style and its collision spheres with MAYA.

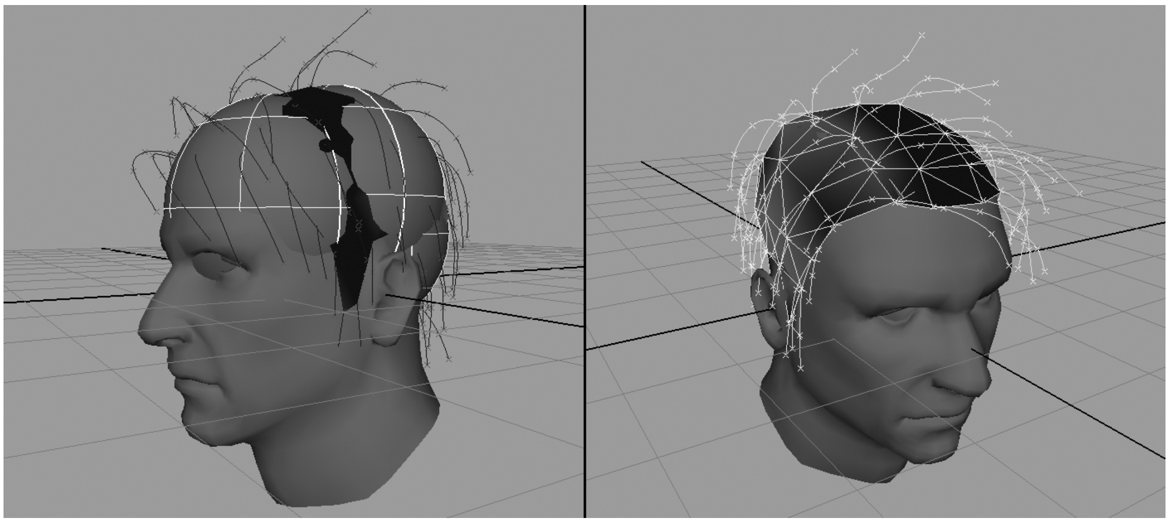


**Figure 5.3.3** *Hair style paths and collision spheres in MAYA*

## 5.4 Generating Hair Data

Graphics cards today often have certain latency for each draw call depending on shader constants and other cache related latency. Therefore it is important to keep the number of draw calls to a minimum. Our implementation uses one buffer for the entire hair model.

Because a hair style consists of a number of triangle strips (wisps) it is also easy to arrange vertices to avoid cache misses causing latency during rendering.

To distribute wisps evenly over a hair scalp, each scalp polygon was assigned a number of wisps which are scattered with *relaxation dart-throwing*. This is both easier and faster than scattering wisps in UV-space over the entire scalp simultaneously, mainly because *relaxation dart-throwing* uses an $O(n^2)$ algorithm. Assigning wisps to polygons was done by computing the area of each polygon and building an area table. The areas in the table, and therefore also the total area, are modified (multiplied) by the scalp texture defining wisp density. A Random number between zero and the total scalp area was then generated for each wisp which could be hashed to a specific polygon. Since the number of wisp for each polygon can be very low depending on the area of a polygon, an additional criterion had to be included for *relaxation dart-throwing* to prevent patterns.

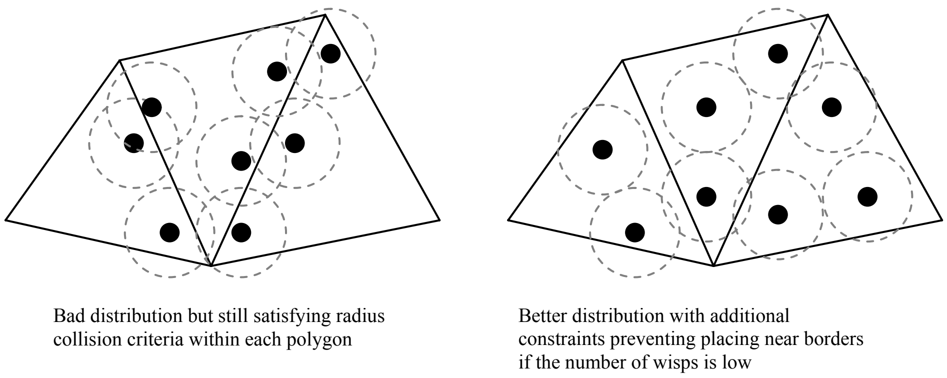

Bad distribution but still satisfying radius collision criteria within each polygon
Better distribution with additional constraints preventing placing near borders if the number of wisps is low

**Figure 5.4.1** *Unwanted distribution patterns*

A circle with its origin in the centre of a polygon defining where wisps can be placed was added. The circle's radius is based on the number of wisps assigned to a polygon. This assures that if the number of assigned wisps is low, they are generated near the centre of the polygon. Finally, before a point is accepted it has to pass a probability test with the density texture. Each generated point is expressed by barycentric coordinates.

Before building the triangle strip a rotation is needed. This was interpolated from the predefined patch directions in MAYA. To allow some variance a random deviation angle (0-45 degrees), modulated with a texture, is added. From one generated point, two new points at a certain distance from the original point controlled by a minimum and maximum value along with a scalp texture can now be created representing the base for the polygon strip. Barycentric coordinates are calculated for these points as well to identify their position relative to the polygons triangle. These three coordinate points can be used to create the full polygons strip by reading at an increasing height from the paths which form a triangular space. The two base points are used to calculate the current patch direction. Patch vertices can then be created at an offset from the original coordinate point in that direction. The offset is modified depending on desired patch shape.

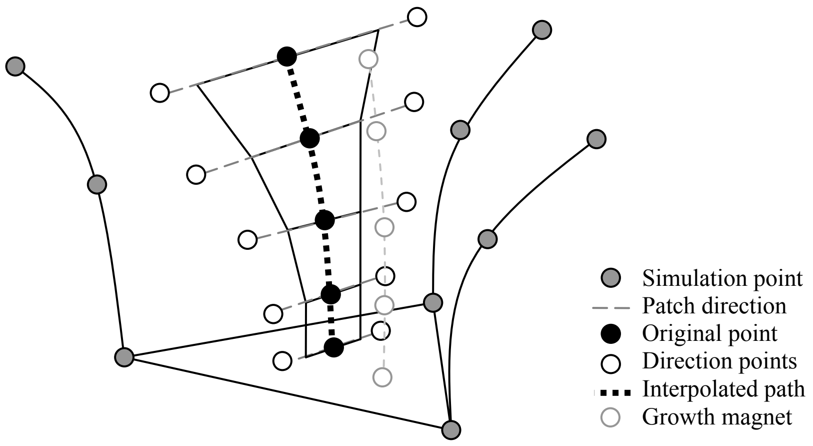


**Figure 5.4.2** *Patch generation*

Using three coordinate points avoids artifacts and strange patch-shapes/widths due to poor triangle spaces compared to just using two fixed coordinate points. Poor triangle spaces are a frequent appearing problem using this model.

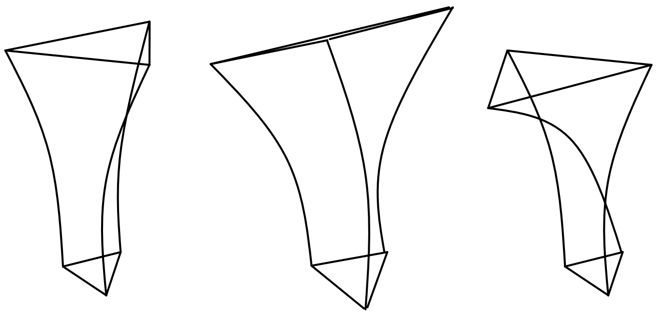


**Figure 5.4.3** *Triangle spaces. Left: A normal triangle space*
*Centre: A triangle space formed into a degenerated triangle*
*Right: A triangle space switching its topology*

The figure above shows two examples of poor triangle spaces resulting in misshaped triangle patches if two fixed points is used. To allow for more variation in growth direction, a second "original" point was generated which could influence wisp growth by acting as a "growth magnet" causing the wisp to bend towards it.

An additional problem that occurred during patch creation was the discretizations of the wisp, a smooth curved surface, to a triangle strip.

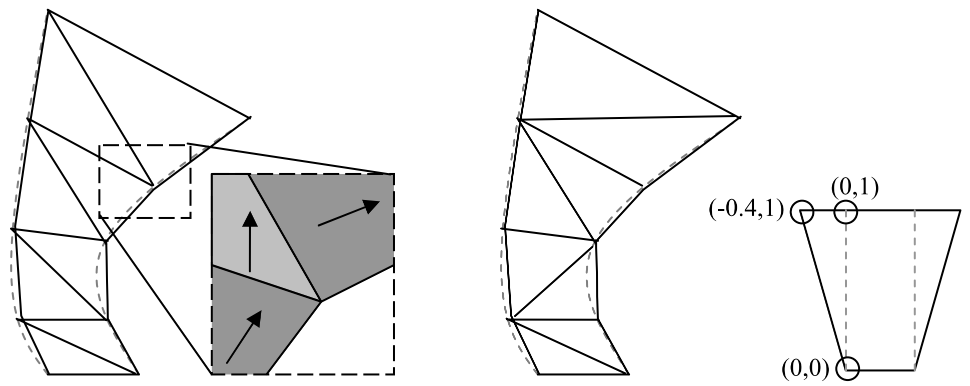


**Figure 5.4.4** *Triangle spaces. Left: A triangle strip causing large variations in plane angles(3D space) causing artifacts for texturing. Centre: A triangle strip with switched polygon splits to reduce variations in plane angles. Right: UV coordinates were adjusted given the increase in polygon width to reduce texture artifacts.*

Large variations in slope for adjacent polygons were causing texturing artifacts. This was reduced by changing the topology of the triangle strip and adjusting UV coordinates to depend on the patch width.

To be able to update the hair patches given the movement of the simulation points each vertex has to have its coordinates in its triangle space and pointers to the paths forming it. Triangle space coordinates are defined by a barycentric coordinate along with tree height values defining where along the paths to read positions from. During hair creation the paths are calculated by spline interpolation but only linear interpolation is available for hardware computation. This will result in reading different position values during updating than creation. The problem is avoided by recalculating the patch vertices' barycentric coordinates with respect to linear interpolation of hair paths.

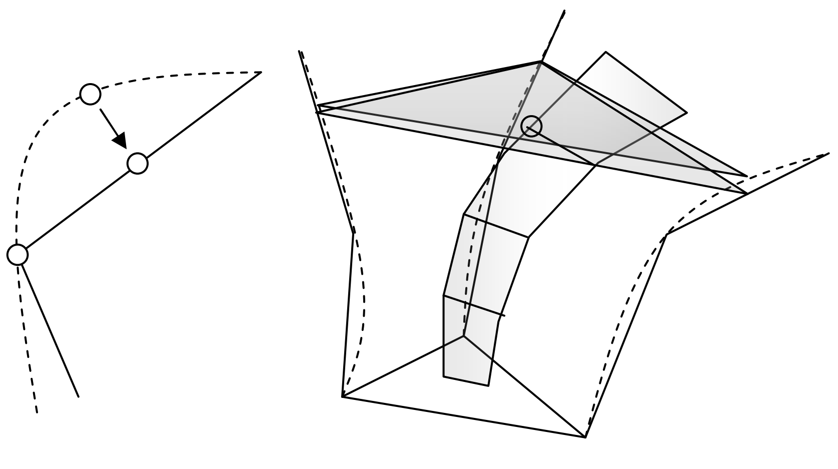


**Figure 5.4.5** *Triangle spaces. Left: Projecting the spline interpolation onto linear interpolation resulting in different position. Right: Reading a different positions will result in a slightly different triangle plane. The original vertex is therefore projected onto the new plane and its Barycentric coordinate is recalculated.*

Hair shading models depends only on the direction of the strands, tangents in this case. To be able to perturb the tangent with textures, comparable with normal mapping, it was also necessary to have polygon normals. Tangents and normals can be calculated easily during patch creation. However these are only valid in a static case. Instead a tangent is stored along with every simulation point, which is updated during simulation. It is now possible to interpolate a tangent in the same way as positions are updated. The interpolated tangent shows the direction of the triangle space which is not always the same as the direction (tangent) of the hair patch. A solution to this is to store and use the relative position of the actual patch tangent compared to the direction of the triangle space. A local coordinate system is computed using the vertex position, triangle space direction an a third interpolation computing the neighboring vertex position. The real tangent and normal can now be expressed in this coordinate space and recomputed during updating with three simple dot products.

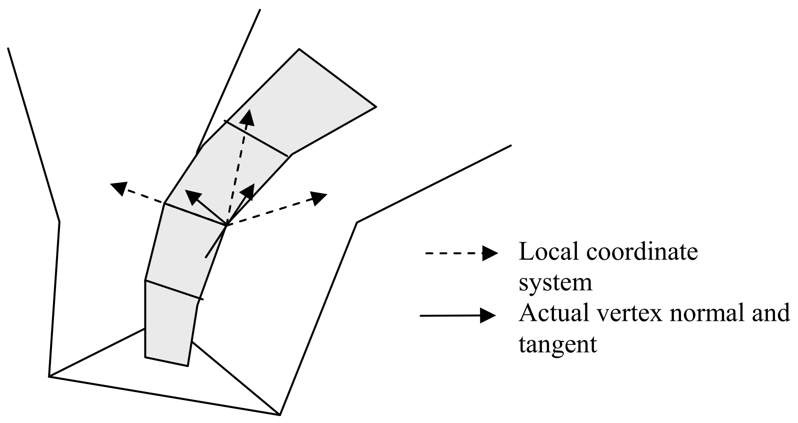


**Figure 5.4.6** *Local coordinate system of a vertex computed by the vertex position triangle space direction and the neighboring vertex position.*

## 5.5 GPU Updating

An important requirement was that the application had to work on graphic cards from both ATI and NVIDIA.

For various reasons discussed the hair is simulated on the CPU. Since simulation occurs in software a method for interpolating the positions of what is drawn is needed. Additionally, for the lighting models to work, a tangent and a normal of the patch are needed. The algorithm used to update the positions, tangents and normals of the drawn geometry with guide strands as input can be divided into three steps:

1. Upload guide strand positions and tangents to the graphics card
2. Draw new patch positions, normals and tangents to textures
3. Draw to screen using updated textures

First, the positions and tangents of all nodes on all guide strands are written to two textures and uploaded to the GPU. The number of guide strands can of course vary greatly, and so can the number of nodes on every strand. However, anything above 64 strands and 8 nodes per strand should be considered extremely high for most hairstyles. Even with these values we get a fairly small texture size of 64 times 8 pixels, which is sent to the GPU every frame. The data is stored so that nodes are indexed along the width of the texture and strands are indexed along the height. This results in some unused space since the texture has to be wide enough to hold the maximum number of nodes any strand has, and all strands do not have the same number of nodes. What is gained is simplicity in the lookup in step 2, and the possibility to utilize interpolation in texture lookups. As mentioned in the previous section the height of a vertex in a strip does not need to be aligned with the nodes of the three simulation strands that defines its position. Therefore, in most cases, the position of a vertex is defined by two nodes on each guide strand. To avoid doing two texture lookups for every guide strand we utilize linear interpolation in the texture lookup. This can be done since the nodes are aligned along the width of the texture; a lookup is done at a position somewhere between the centers of two pixels length wise, corresponding to the position of the vertex at creation time in relation to the two closest nodes on the strand. This way no interpolation is done height wise (between strands) in the texture, but utilized to avoid two lookups length wise (between nodes). This is exploited in the second step of the algorithm. Current graphics hardware does not support interpolation with floating point 32bit or 16bit textures. Because of this, and the need for interpolation, the guide strand positions texture uses 16 bit integer textures. The positions are scaled to fit within a bounding box containing the hair before being written to the texture. When the texture is used the inverse of the scaling operations is performed. Since hair usually does not occupy a large area in space and we have no need for microscopic resolution 16 bits is more than enough. The same is true for tangents. For the example above with 64 times 8 pixels the data sent to the graphics card for the positions texture and the tangents texture every frame is about 16 kilobyte.

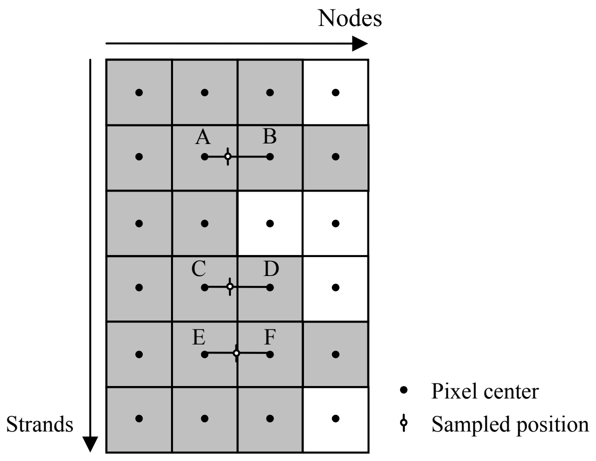


**Figure 5.5.1** *An example of the positions texture. Every square is a pixel; if they are grey they contain position data. In the figure a vertex takes a sample from the second strand from the top. The position of the vertex has been determined at creation of the strand to be at a height on that strand that corresponds to the sampling point, in relation to nodes (or pixels) A and B.*

When the positions texture and the tangents texture have been uploaded a rendering pass is executed where the positions, normals and tangents of all vertices of the drawn geometry are drawn to three textures. To accomplish this four static textures are uploaded once and stored on the graphics card containing the following data:

- Indices of the three guide strands
- Heights on guide strands
- Weights of guide strand values
- Weights of guide strand values of the neighboring vertex

This pass is done in a one to one mapping of pixels by drawing a quad to the screen and reading the values (from the four textures above) from the same UV coordinates that are rendered to. Since positions, normals and tangents are defined by three scalars each (in 3D), they can be stored in the R, G and B channels of the texture. One texture read for each of the four textures above is performed per-pixel. The index texture contains the indices of the three guide strands influencing the position of the vertex and the height texture holds the heights on the guide strands where the position should be read. These two values for every guide strand together make up the UV coordinate to be read from the texture containing the positions of the guide strands. As mentioned briefly in chapter 5.4, tangents and normals are defined in a tangent space at each vertex. This tangent space is defined by the weighted tangents of the three guide strands and by a binormal defined as the direction from the vertex in question to its neighboring vertex. All of this results in three reads for the tangents and none for the bi-normal. The latter is possible since the neighboring vertex is at the same height and uses the same guide strands as the current vertex; the only difference is the weights used. A normal is produced by the cross-product of the tangent and the binormal. A new binormal is then produced by the same operation on the new normal and the tangent. This way an orthogonal coordinate system is defined and the final normal and tangent can be calculated by three dot products each. All of this is similar to the approach used in the cascading spaces chapter,

$$\begin{aligned} B_0 &= P_1 - P_0 \\ N &= B_0 \times T \\ B &= N \times T \end{aligned}$$

where $P_0$ denotes the position of the current vertex and $P_1$ its neighbor. The final tangent $T_v$ is then calculated as

$$T_V = \begin{pmatrix} B_x & N_x & T_x \\ B_y & N_y & T_y \\ B_z & N_z & T_z \end{pmatrix} \begin{pmatrix} T_{xL} \\ T_{yL} \\ T_{zL} \end{pmatrix}$$

where $T_L$ is the tangent as defined in the space defined by N, T and B. Analogous for $N_v$.

Finally, the positions and tangents of the drawn vertices need to be set to the values written to the textures. Currently two different approaches are used by the two main graphics card producers ATI and NVIDIA. Current NVIDIA cards support texture fetching in the vertex shader which is defined by shader model 3.0. ATI’s approach is similar to what has been possible in OpenGL. Their technique, dubbed *Render to Vertex Buffer*, or *R2VB* for short, allows for a render target to be used as a vertex stream. This, of course, does not allow for random access of the texture in the vertex shader, but it provides exactly what we want, a means of getting the serial data from the position and tangent textures to the vertex shader.

Two other obvious approaches for the whole process would be to do all of the simulation on the GPU, or to write all of the data to the vertex buffer on the CPU and upload it to the GPU. The first approach would require persistence for positions and momentum, something which current graphics cards are not designed for. It is possible to solve in a number of ways, such as alternating the roles of two textures between render target and data storage. Additionally, since tangents and normals are needed for the lighting model to work, the current positions of neighboring vertices are needed, which would introduce an extra pass similar to the one described in this chapter. As will be presented in the results the simulation does not require much CPU time. The bottleneck, at least at close ranges, is the final pixel shader, which motivates the current CPU based simulation. This might also be the way to go with current and future hardware moving towards multiple cores. As for writing to the vertex buffer on the CPU, it is very slow; results of this are discussed in the results section. The obvious advantage of the approach described in this chapter is that only the positions and tangents of the guide strands, which are not very numerous, are sent to the GPU every frame. All of the data needed for the GPU update are static and kept on the GPU throughout the program execution and the three textures containing the results of the operation are never sent to the GPU. An alternate method on NVIDIA cards could be to skip the pass updating the positions texture and perform those operations directly in the vertex shader when drawing the geometry. This would require six texture fetches in the vertex shader as opposed to three. The latency of texture fetch in the vertex shader with current hardware is high compared to texture fetch in the pixel shader. This might make the alternate approach slower.

## 5.6 Transparency

Alpha to coverage, described earlier, which was used for handling transparency possesses a problem that needs to be addressed to avoid artifacts. When the texture describing wisp transparency is rendered from a distance and higher mipmaps are queried, the texture becomes more and more blurred until it reaches its mean value at the highest mipmap. This results in larger semi-transparent areas which are not handled very well with alpha to coverage. To avoid this the mipmap level is calculated in the pixel shader using the intrinsic derivative functions, *ddx* and *ddy*, and used to sharpen the texture.

As an experiment, an attempt to sort all hair patches individually and use blending was implemented. Sorting was done by preprocessing all wisps (once) and storing their root, tip and middle position. Since the hair model is constructed from fewer strands, which in most cases do not move very much relative to each other, the assumption that the same applies to the wisps was made. These points could later be used to dynamically compute the closest point on each wisp to the camera. As the wisps are stored in coherent chunks of memory in the index buffer the distances were stored in a structure along with pointers to index buffer memory. This was sorted according to distance and used to rearrange a dynamic index buffer. However this did not manage to solve the transparency issues. One would need to sort each polygon individually, which would be too demanding of resources.

Even though a rough sorting is not good enough for blending, it can improve performance when using alpha testing due to increased rejections of z-tested pixels. For the latter purpose the geometry has to be sorted front to back instead of the normal sorting order of back to front.

## 5.7 Shadows

Opacity shadow maps were implemented to allow for 16 opacity layers to be rendered in one pass. Most recent graphics cards support four render targets. Each render target can hold four channels and since the opacity value is a scalar a total of 16 opacity values can be stored per-pixel. This single pass solution, as described in [ND05], is fast enough to be suitable for real-time applications and games. Formula 3.2 in chapter 3.3.2 and its description defines the opacity thickness function $\sigma(z)$ as strictly increasing with depth and occlusion from the light source. Opacity shadow maps is a discretization of this value into a set of z values $z_0$ to $z_n$. When retrieving an opacity value for a depth z, linear interpolation is done between the two closest layers in z space. Given this, further improvements are presented by [ND05] for the lookup. For a given pixel (x,y) the opacity thickness at z, $\sigma(z)$, can be expressed as

$$\sigma(z) = \sum_{i=0}^{n-1} (\sigma(z_i) \times \max(0, 1 - \frac{|z - z_i|}{dz})) \quad (5.1)$$

This is in other words is an expression for the linear interpolation mentioned above, which is suitable for shading languages. For mathematical correctness this would be calculated in the pixel shader. However, to save computation time, a part of this expression can be computed in the vertex shader without noticeable loss of quality. The values computed can be seen as weights which are to be multiplied with the values received from all opacity layers. These weights are defined as

$$w_i = \frac{|z - z_i|}{dz}$$

The computation of formula 5.1 is then reduced to a computation of the sum

$$\sigma(z) = \sum_{i=0}^{15} w_i \sigma(z_i) .$$

If the weights $w_i$ are calculated and stored in quintuplets, the calculation of four of the elements in the sum above can be done with a single dot product. This is further simplified since the opacity thickness is already stored in quintuplets.

$$\sigma(z) = \sum_{k=0}^{3} w_k \cdot \sigma(z_k)$$

Here $w_k = \{w_{4k+0}, w_{4k+1}, w_{4k+2}, w_{4k+3}\}$ and analogous for $\sigma(z_k)$. The final shadow value which is to be multiplied with the color of the hair is then computed with formula 3.1 in chapter 3.3.2. If the graphics card in question benefits from branching texture lookups, this can be implemented as well. In most cases, only one texture lookup is required with a maximum of two, if the opacity planes from which the interpolation is done are part of different textures.

To maximize the quality of the available opacity maps and their resolution, the hair is fitted into a bounding box. The far and near plane of the light are set to fit the model in its current position. The field of view is then set so that it would fit the extremes of the bounding box from any angle. The latter is a simplification to avoid extra calculation, the values for this was already available from the GPU updating algorithm. An optimal solution would require the projection of all simulated points onto the light projection plane.

Opacity shadow maps work very well for highly transparent objects. However, there is an inherent problem with the algorithm when rendering opaque objects or objects with low opacity. The problem occurs when an opaque surface is rendered which is at an angle from the direction of the light.

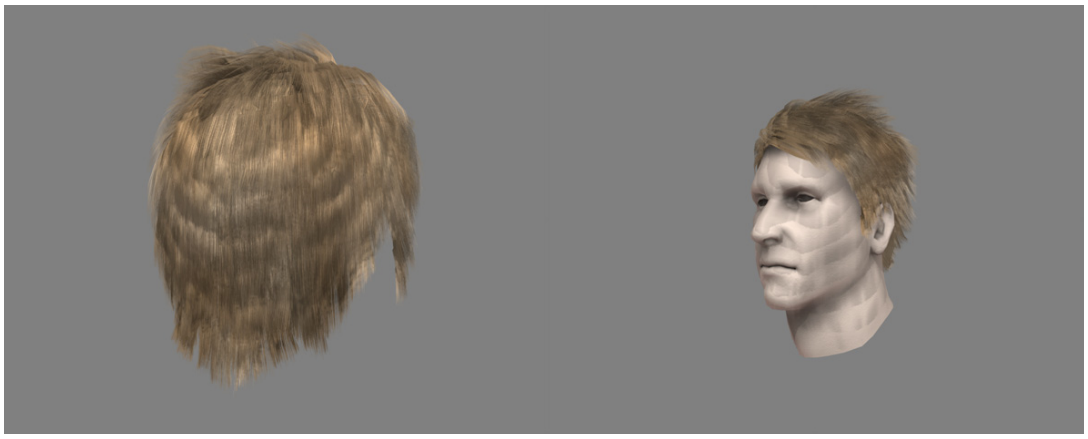
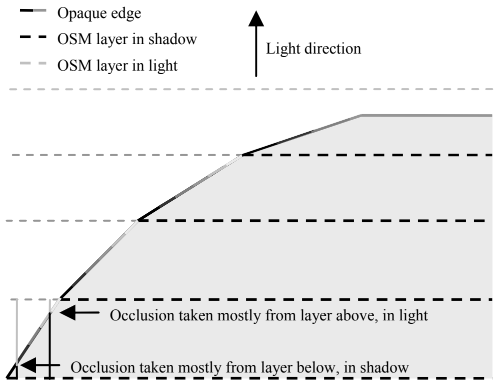


**Figure 5.7.1** *Two screenshots illustrating the problem of applying opacity shadow maps to highly opaque objects*

All parts of the layers in the opacity map which are occluded by the object will have an opacity value of zero. When the object is rendered and a lookup in the opacity map is made, the opacity value is interpolated between the closest two layers. As figure 5.7.1 illustrates, when an opaque object which is not in shadow is rendered and the edge of the object is oriented so that it is not parallel to the layers of the opacity map, a lookup is always made in one layer in shadow and one in light. The final opacity value will vary depending on the distance from these layers, giving rise to the effect seen in figure 5.7.1.


Opaque edge
OSM layer in shadow
OSM layer in light
Light direction
Occlusion taken mostly from layer above, in light
Occlusion taken mostly from layer below, in shadow
**Figure 5.7.2** *Problems with opacity shadow maps*

There is no good solution for this. Opacity shadow maps are not meant to be used with opaque objects as both occluders and receivers. A large bias can be added either in the lookup or when rendering to the opacity map, effectively translating the layers in the map a distance away from the light. This does not solve the problem, but can limit it to appear for angles above a certain value. For hair this is usually not a problem, the opacity value used in figure 5.7.1 is much higher than usual. A normal opacity value and a small bias eliminate most of the artifacts. As for rendering opaque objects with shadows, a regular shadow map is more suitable. When rendering hair a lookup can be made in that shadow map as well as in the opacity shadow map used for the hair. When rendering a skull a lookup can still be made in the opacity shadow map, regardless if the skull was rendered to it or not.

The latter shadowing method described in chapter 3.3.4 is very straight-forward to implement. A regular depth pass is rendered from the light. In this pass the depth of the alpha tested hair is rendered. When the geometry is rendered to the viewport the position is transformed in the light's projection space. The difference between the value written in the depth pass and the value of the current geometry's depth from the light is used as input to the transmittance function. The approximation being that the occluder closest to the light determines the transmittance for all pixels in its shadow, regardless of the thickness of in-between geometry. This works fairly well with most hairstyles at distances, where the hair and head can be seen as a uniform convex shape. In the figure below we clearly see the gradient produced by the transmittance function to the right as compared to the more correct transmittance produced by opacity shadow maps.

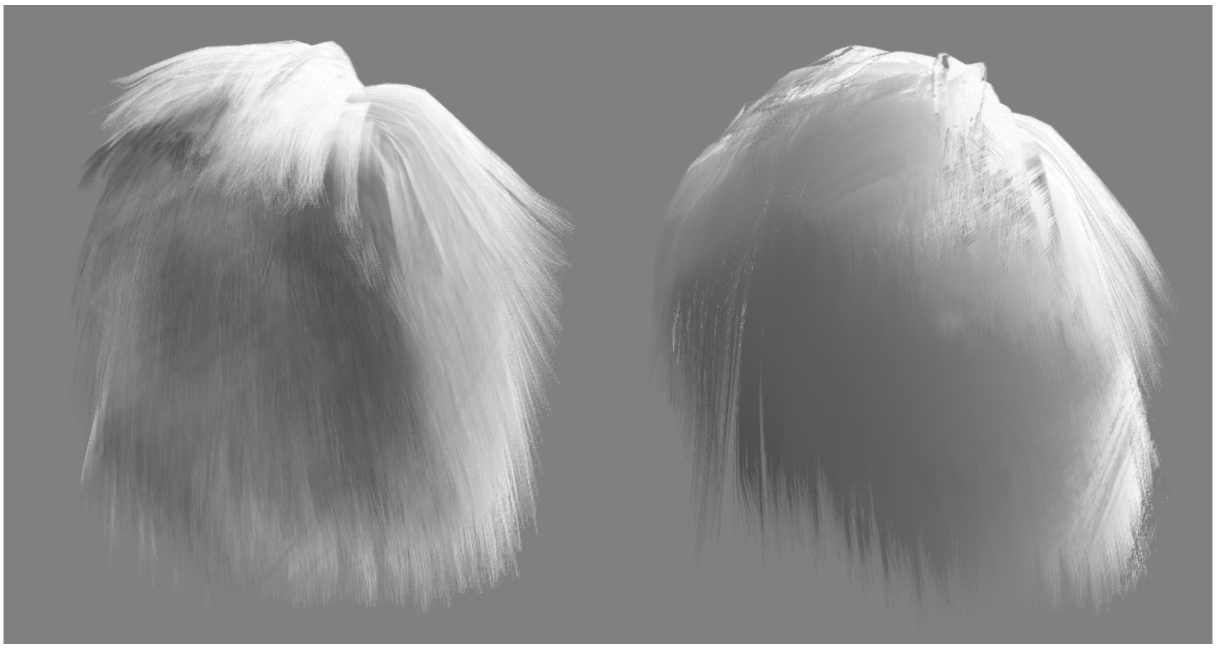


**Figure 5.7.3** *Difference between shadow techniques*
*Left: Opacity shadow maps, Right: Modified Shadow map*

For static self shadowing, a per-vertex approximating solution as presented in chapter 3.3.1 was implemented. Each hair vertex is approximated by a cylinder and each scalp vertex by an oriented disc. Since the solution casts self shadows based on distance between vertices, artifacts often appears due to discretization (vertex count is preferably held low for games). Artifacts caused by hair vertices casting shadows onto hair vertices or hair scalp are often not immediately visible since their shadow is modulated by an opacity value and therefore the artifacts are ignored in this thesis. However when scalp vertices cast opaque shadows onto hair vertices they can occasionally cast to much or too little shadow depending on the distance. For example, a patch's base vertices can be located at different distances for scalp vertices and therefore receive different amount of shadow when they should have the same. This is solved by always adding a constant shadow value at the hair base vertices. Furthermore if a vertex can be projected onto its shadow casting oriented scalp disc and the projected distance is smaller than the distance between the vertices this is used instead.

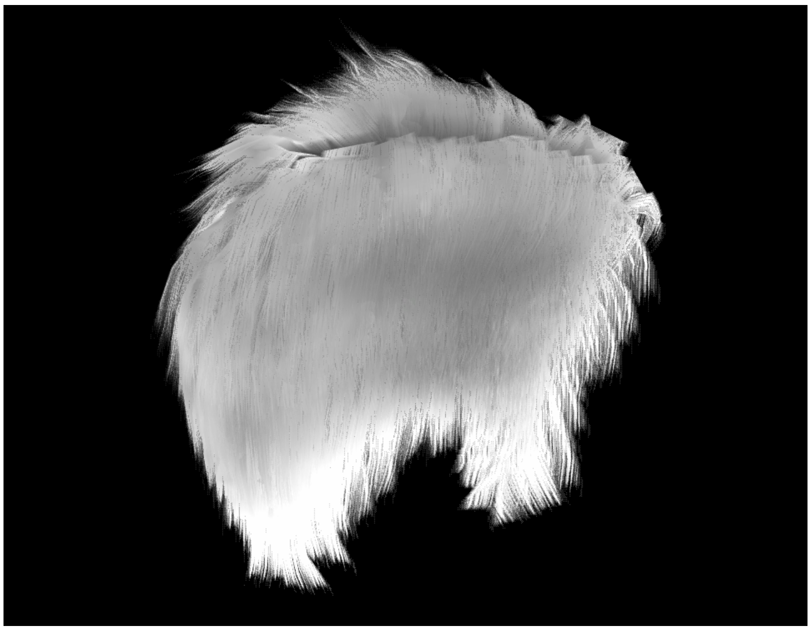


**Figure 5.7.4** *Result of ambient occlusion calculation.*

## 5.8 Shading

For the possibility of comparison, both the Kajiya-Kay and the Marschner models were implemented as pixel shaders. Kajiya-Kay requires only a few calculations and is easily adapted for real-time computer graphics. This makes it easy to implement the entire model in a pixel shader. Marschner is a much more complex model and is not suited for real-time computer graphics. It requires too many and computationally heavy calculations for it to fit in a pixel shader. Therefore a similar solution to [ND05] was implemented. Assuming that the internally reflective properties such as hair strand width, cuticle shifts, absorption and index of refraction are the same over the entire hair style. It is then possible to compute lookup tables for $M(\theta_i, \theta_r)$ and $N(\phi_i, \phi_r)$ in a preprocessing step.

For this thesis it was chosen to use the same absorption for all colors and force $N(\phi_i, \phi_r)$ to output a single channel per reflection/refraction and thus reduce storage. Color absorption is modeled by multiplying the reflected intensity with the hair color in TT and TRT while using a separate specular color in the R case, see section 3.2.2 for details.

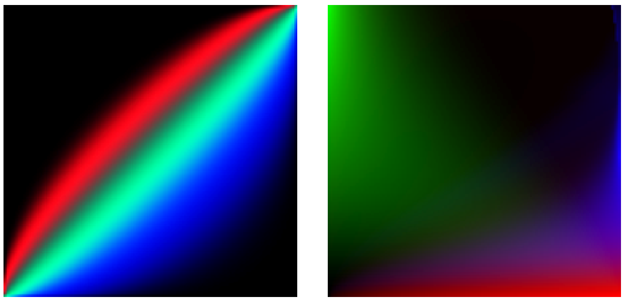


**Figure 5.8.1** *An example of lookup tables for the Marschner shading model. Left: M-function. Right: N-function*

To handle caustic effects due to singularities it was decided to use a simplified version compared to the one proposed by Marschner and simply clamp values above an allowed threshold. Marschner only describes the specular reflectance model. Therefore the diffuse term of Kajiya-Kay was added to Marschner's model as well.

The functions $M(\theta_i,\theta_r)$ and $N(\phi_i,\phi_r)$ have angles as input parameters. This is not very computationally friendly since the pixel shader only has positional vectors and computing angles from these would involve trigonometric functions such as arcsin, arcos or arctan, which are very expensive. Instead, the same technique used by [ND05] was implemented where the lookup tables were calculated to take $\sin(\theta_i),\sin(\theta_r)$ instead of $\theta_i,\theta_r$ and $\cos(\phi_i),\cos(\phi_r)$ instead of $\phi_i,\phi_r$ as input parameters. These values can be computed by the following formulas.

$$\begin{aligned}\sin(\theta_i) &= lightvector \cdot T \\ \sin(\theta_r) &= eyevector \cdot T\end{aligned} \quad \text{where } T \text{ is the tangent}$$

$$\cos(\phi_i) = (eyePerp \cdot lightPerp) \times ((eyePerp \cdot eyePerp) \times (lightPerp \cdot lightPerp))^{-0.5}$$

where *eyePerp* and *lightPerp* are the light and eye vectors projected perpendicular to the hair.

$$\begin{aligned}eyePerp &= eye - (eye \cdot T) \times T \\ lightPerp &= light - (light \cdot T) \times T\end{aligned}$$

Since $\cos(\phi_r)$ is a function of $\theta_i,\theta_r$ we can use the alpha channel of the first lookup table to store this and save some computations.

Due to the limitation of a small number of polygons it can be hard to create the feeling of a thick hair style. Especially when a shorter hairstyle is created and one can often see the hair scalp between the polygon wisps. To avoid this problem we apply the same shader to the hair scalp and replace its normals with the base tangent of its accompanying hair strand, using these for shading along with a texture that perturbs the tangent according to the direction of the hair. This makes the blend between the scalp shading and hair strands almost unnoticeable.

To avoid latency in pixel shader execution, texture lookups were distributed over the code and variables and operations were packed in 4D vectors to optimize performance.
In an attempt to reduce the number of pixel evaluations caused by heavy overdraw due to transparency (alpha to coverage) and a fairly heavy pixel shader, a z-pass was rendered, writing the depth value of the final rendering to the z-buffer. After that the hair was rendered with z-function equal to try to prevent the graphics card from evaluating the pixel shader in a pixel that ultimately will fail the per pixel z-test. However these attempts were only successful on the ATI graphics card since this functionality was not available on NVIDIA cards. A small optimization that did work for both graphics card was to render the head (opaque) before the hair and gain some performance by hierarchical z-cull rejection (gross rejection of triangles, no z-buffer comparison). It was also discussed to split the hair into several chunks (draw calls) and render invisible occluders in between them to further take advantage of hierarchical z-cull. This was not implemented since it would increase the number of draw calls and the overall gain was presumed to be too small given the amount of work it would require.

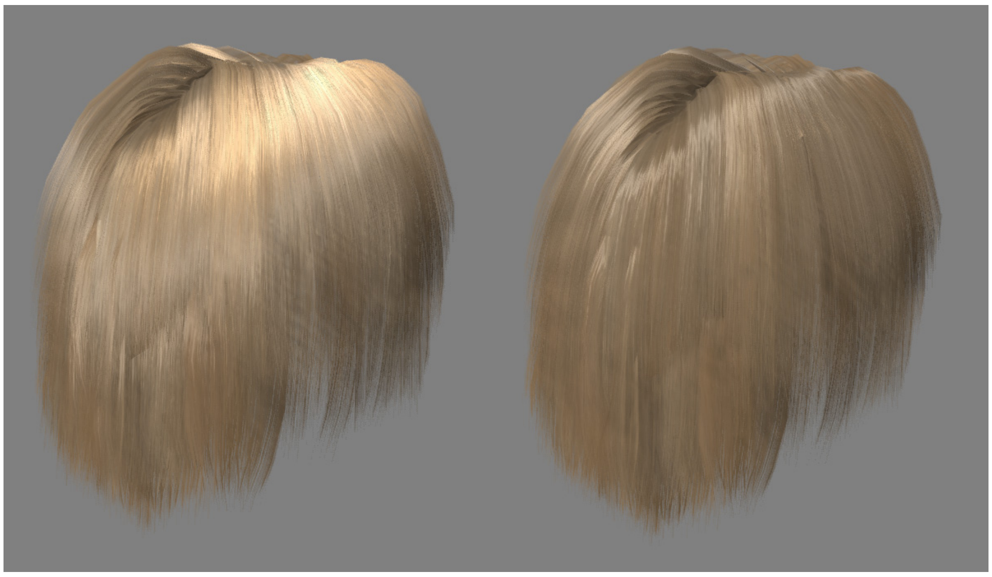


**Figure 5.8.2** *Left: Marschner model. Right: Kajiya-Kay model.*

## 5.9 Simulation

We cannot simulate a full head of hair exactly. Even in offline simulation compromises are made and shortcuts are taken, it's not plausible to simulate every strand with collisions and friction. The way a strand acts in an area shared with thousands of other strands is very different from the way it would act on its own. Not only would an exact simulation need to model every hair as a continuous curve in space with collisions, there are other factors that would usually be ignored, such as the increased cohesiveness and friction close to the scalp, due to the increased density caused by young hair and a small amount of grease that is present even in very clean hair. As with most graphics and physics in games, the goal is not something that simulates reality, but something that looks and behaves like reality. Ultimately what is real is what passes as real to the eye of the beholder.

Three models were implemented, the Euler angle model, the local coordinate system model and the simple spring model. For the Euler angle model an observation was made that the system defined by [DCF01] and described in section 4.2 is close to Gimbal lock in the initial position for most trees and for straight hair strands. For a strand such as this, the position of the strand is along the z-axis, which is the primary rotation axis. This does not matter for the definition of the position of the strand, since the secondary rotation axis, y, is rotated with the strand in the z-axis rotation. What does matter though is how the system behaves when moving and when affected by external forces. If a close to straight strand is affected by a force parallel to the y-axes of that strand, the response is stiffer than if the force would have been orthogonal to the y-axes. The torque

$$\tau = Lz \times F$$

generated by such a force F would be orthogonal to both the z and the y axis and so would the resulting angular acceleration

$$\dot{\Omega}_{rel} = \frac{1}{m}\tau$$

As shown in chapter 4.2, the angular joint accelerations $\ddot{\theta}$ and $\ddot{\phi}$ are calculated by projecting the angular acceleration on the z-axis and the y-axis. Since these axes are orthogonal to the angular acceleration there is no change in the angles of the strand, it's completely stiff to such forces. This occurs because an arbitrary force is projected onto two axes. One way to ease these problems is to change the model defined in [DCF01] to use the x-axis of the parent instead of the z-axis as the primary rotation axis. The changes to the formulas listed in chapter 6.2 consist of this change only, and need not be listed here. The torque, of course, is still calculated using the z-axis as shown above, since this axis is the strand segment itself, which the force acts on. The problem does not disappear if the strand's orientation is close to either of the rotational axes, since a force acting on it orthogonal to these axes will result in the same unwanted behavior. However, as mentioned, strands (and trees) are often closer to straight than bending ninety degrees, which makes the algorithm with the proposed changes of axes more suitable.

For all three models, a decreasing spring strength with distance from the root of the strand was used. Partly due to the reasons given in the first paragraph of this chapter, but mostly because this is what looked most natural. A quadratic reduction was usually a good compromise between stiffness and natural flow, but a linear or cubic reduction is also usable. With some initial strength *k* at the root, the strength $k_L$ at a distance *L* from the root would be simply

$$k_L = \frac{k}{(Lc)^a}$$

Where *a* is 1 for linear, 2 for quadratic and 3 for a cubic reduction. *c* is a constant scaling of the length.

All of the implemented models (and almost all models in existence) simulate strands with a number of nodes connected one after another. With the addition of the first of the criteria listed in chapter 4, the problem is similar to having a pendulum with a large number of joints return to a pre-defined position at rest. This is accomplished by combining dampening and spring forces which make for a natural appearance and also accepting arbitrary influence from head movement and external forces. Additionally the model needs to be relatively stable, which is not a problem if it does not need to be pleasing to the eye. As mentioned in the opening paragraph of this chapter, we cannot expect to model hair exactly, and modeling a few simulated strands as close to reality as possible would not represent a full head of hair. A few isolated strands do not act like 40000 which are closely interacting. What we attempt is to make a few strands act in a way to represent those 40000 of a full head of hair. The implemented models accomplish this with varying degree of stability. The most stable of the models is not surprisingly the most simple; the spring model described in chapter 4.4. Instability only appears in the form of wobbling where the spring forces greatly exceed the influence of dampening. The two other models, however, may suffer from more severe instability when forces are applied in a manner not suited for them. As mentioned as a reason for modifying the model, the interactive forest model has issues with Gimbal lock. This becomes very apparent when external forces are influencing nodes which are close to entering that state. This can be redeemed by limiting the angles in such a way that Gimbal lock never occurs, but that easily results in an unnatural appearance. The instability problems occurring in the cascading coordinate systems model are mostly caused by high stiffness in the springs connecting the downward reference nodes to their strand nodes. With longer strands and in states where equilibrium is not possible this causes a struggle between the upward and downward reference nodes. A solution which works fairly well is to reduce the stiffness of the downward reference nodes to a fraction of the stiffness of their upward counterparts, giving the latter priority. To sum up; the spring model is relatively easy to handle. For the two other models great care needs to be taken to avoid instable behavior.

# 6 Results

The performance of our method is described in detail in section 6.1. Numbers and pictures do not describe the physical behavior or other characteristics of a dynamic hair. Therefore we present an evaluation in section 6.2 created with the help of artists. Finally thoughts on improvements and future work are presented in section 6.3.

## *6.1 Performance*

We present various performance measurements illustrating the benefits and pitfalls of our method. We will see that the model fits both high-definition cut scenes and in-game integration made possible by high its performance.

### 6.1.1 Test Setup

All measurements were done on a 3.6 GHz Intel Pentium 4 with 2 Mb L2 cache and 2 Gb 532 MHz RAM. Two different graphics cards were used, both with 256 Mb of memory, one ATI X1800XL and one NVIDIA Geforce 7800 XT. The ATI card was used for all measurements except where times for individual draw calls are presented or where the measurement is specified to be on NVIDIA. The screen resolution used was 1024x768, with four antialiasing samples on NVIDIA, six on ATI where no comparison with NVIDIA is made, and four on ATI otherwise.

### 6.1.2 CPU Simulation

Since the number of simulation strands and nodes needed for a visually pleasing result is small the CPU time needed for simulation is rarely a bottleneck. At 30 fps the simulation time is typically less than half a percent of the CPU time.

| | 9 strands | 25 strands | 49 strands | 121 strands |
|---|---|---|---|---|
| Static | 32.10 | 86.44 | 162.60 | 394.38 |
| Forest | 46.78 | 122.54 | 1460.50 | 9546.57 |
| Cascade | 44.12 | 110.05 | 203.48 | 553.60 |

| | 9 strands | 25 strands | 49 strands | 121 strands |
|---|---|---|---|---|
| Static | 21.12 | 56.03 | 103.40 | 259.98 |
| Forest | 25.21 | 72.90 | 690.40 | 4353.57 |
| Cascade | 21.53 | 61.47 | 115.41 | 277.08 |

***Figure 6.1.1*** *The tables show computation times per frame for simulation with different strand configurations (9, 25, 49 and 121 strands) and simulation methods (Static, Forest, Cascade). In the right table four nodes per strand were used, in the left table, eight nodes per strand were used. All values are in microseconds.*

Immediately obvious is that the forest model is slower than the other two. While the simple spring model and the cascading spaced model scales in a linear fashion, the forest model does not. The implementation of the latter is more complex and requires handling of more data to a point which might not suite the processor. The simple spring model is slightly faster than the cascading spaces model for all numbers of strands and nodes. Doubling the number of nodes per strand does not double the simulation time. For the spring model a doubling of nodes results in an increase in iteration time by about 55 percent. For the other simulation models the increase varies between 70 and 120 percent with a general rising trend with increasing strand count.

### 6.1.3 GPU Simulation

As shown above, the simulation is not very demanding. The problem lies in updating the positions of the drawn vertices in some way according to the positions of the simulation strands affecting those vertices. This was done in three ways; a software update, and two different hardware update methods depending on the hardware support. The CPU update method was done by writing directly to a vertex buffer and sending the updated vertices to the GPU. Both of the hardware methods use the algorithm described in 5.5 . The difference is in how the vertices are updated from the texture containing the positions of the updated simulation nodes. On NVIDIA cards the final positions are fetched from the positions texture in the vertex shader. On ATI, R2VB is used to reinterpret the positions texture to a vertex buffer directly.

In the measurements below in figure 6.1.2 the total frame time was measured for the three different methods. The time needed to write to the vertex buffer or to the texture containing the positions of the simulated nodes could of course be measured. Some parts of these methods can not be measured accurately however. When the texture or vertex buffer is actually sent to the GPU is up to the driver. The extra time required for the different paths of NVIDIA and ATI would also be impossible to measure. The result of this is that the time measured is not directly representative of the method used, but rather a comparison between method and hardware.

| polygons | 2000 | 4000 | 10000 | 15000 |
|---|---|---|---|---|
| ATI | 0.820 | 0.808 | 1.058 | 1.650 |
| NVIDIA | 1.129 | 1.393 | 1.855 | 2.415 |
| SW | 2.959 | 4.975 | 12.346 | 17.544 |

***Figure 6.1.2*** *Total frame time in milliseconds using the three updating methods and four different polygon counts.*

As expected the GPU methods are greatly outperforming the CPU method. Since we usually deal with large numbers of vertices, uploading the vertex streams every frame is not a good idea. Doing the required floating point calculations in the updating pass on the GPU suits its capabilities in this area very well. However, for small numbers of vertices updating on the CPU is still viable. Especially if integrating an extra pass for the update, with custom solutions for different hardware, becomes an expensive task. The similar values for 2000 and 4000 polygons on both of the hardware methods, but especially on ATI, are explained by the power-of-two nature of the updating texture. The large difference between the ATI and NVIDIA paths should be expected. The NVIDIA path requires texture lookups in the vertex shader where the ATI path uses R2VB to interpret the texture as a vertex stream.

| Polygons | 2000 | 4000 | 10000 | 15000 |
|---|---|---|---|---|
| Time | 88 | 287 | 287 | 360 |

***Figure 6.1.3*** *Time taken, in microseconds, for the hardware update pass with varying polygon count.*

At 30 fps and with 4000 (or 10000) polygons the time taken for the updating draw call is less than one percent of the frame time. The implementation creates square power-of-two textures to fit the data. The size of the textures should be the major factor for these passes. This is corroborated by the identical times for 4000 and 10000 polygons.

### 6.1.4 Shadows

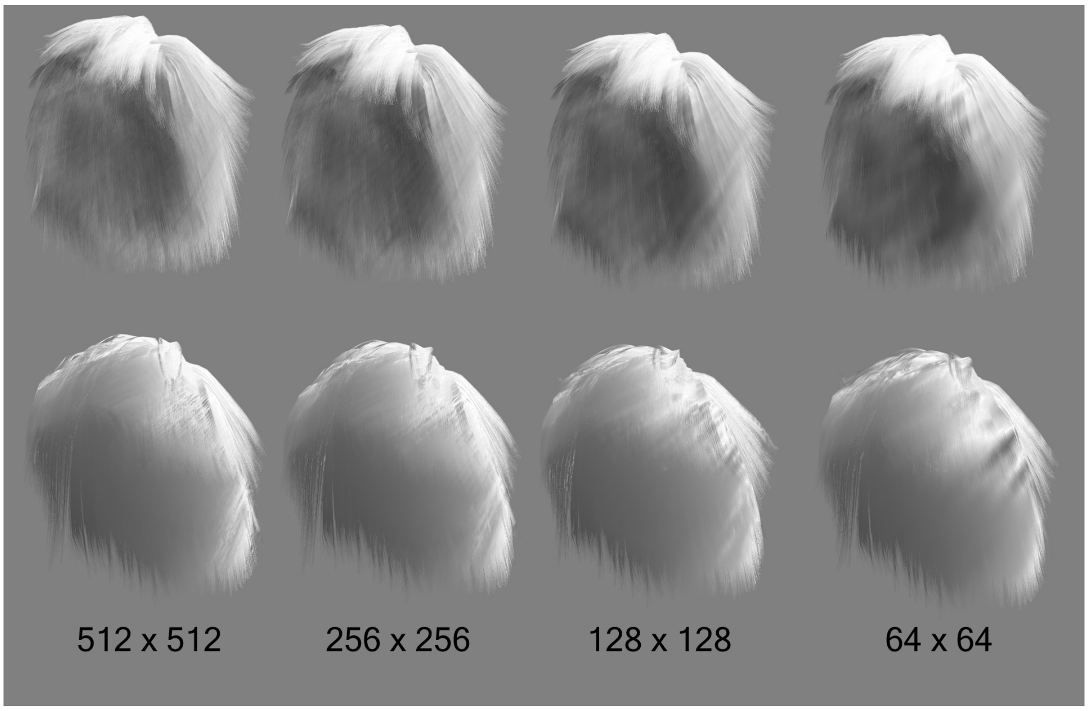


***Figure 6.1.4*** *Pictures showing shadowing with different resolutions on opacity shadow maps (top) and modified shadow maps (bottom).*

The figure above gives an impression of the visual quality difference between shadowing methods and fidelity of the method used. Obviously higher resolution looks better, but a resolution of 128x128 does well against 512x512, a resolution containing 16 times the number of pixels. As with all things in real-time computer graphics, there is a trade off between quality and speed.

| Resolution | 64x64 | 128x128 | 256x256 | 512x512 |
|---|---|---|---|---|
| Opacity | 220 | 235 | 640 | 2010 |
| Modified | 108 | 118 | 272 | 791 |

***Figure 6.1.5*** *Time taken, in microseconds, for the pass updating the shadow maps, with 4000 polygons.*

| Resolution | 64x64 | 128x128 | 256x256 | 512x512 |
|---|---|---|---|---|
| Opacity | 280 | 490 | 1380 | 3208 |
| Modified | 266 | 277 | 483 | 1511 |

***Figure 6.1.6*** *Time taken, in microseconds, for the pass updating the shadow maps, with 10000 polygons.*

Immediately obvious in the measurements above is that the modified shadow map method is much faster. This is expected since it uses one render target as opposed to four in Opacity Shadow Maps. In addition to this the Modified Shadow Map method renders the alpha tested hair as opaque, where Opacity Shadow Maps uses alpha blending. The similar values for 64x64 and 128x128 presented in 6.1.5 and the small difference when using 64x64 (in the 4000 and 10000 polygon setups) implies that there is an overhead at work. Excluding the lowest resolution, a doubling of the width and height, giving a pixel count four times higher, roughly triples measured time in all instances. For resolutions above 64x64, changing the polygon count from 4000 to 10000 roughly doubles the draw call time. However, the time taken for the updating draw call is not completely indicative of the difference between the two methods. The lookup using opacity shadow maps is much more demanding than the calculations done for the modified shadow map method.

| | far | medium | close |
|---|---|---|---|
| shadow map | 13,4 | 1,651 | 0,589 |
| opacity maps | 19,4 | 2,289 | 0,814 |

***Figure 6.1.7** Main draw call time for 4000 polygons using shadow maps and opacity maps with a resolution of 256x256 pixels.*

As expected the opacity map draw calls are more demanding of the GPU than the modified shadow map draw calls. Not only does the opacity method require more texture lookups, but also more instructions overall and more interpolators.

## 6.1.5 Rendering

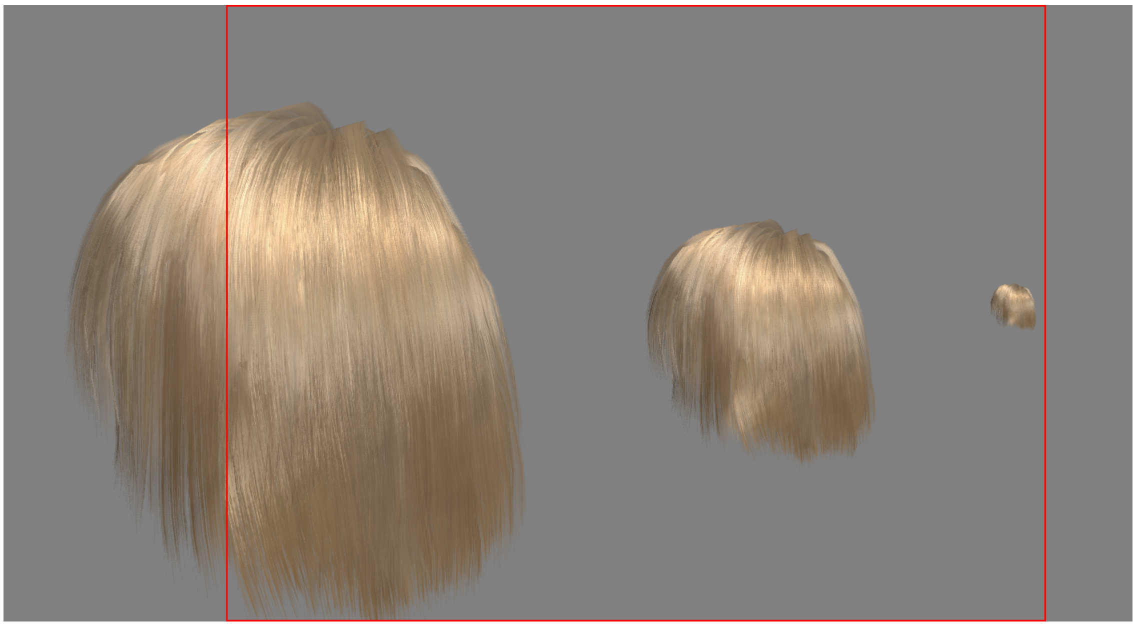


**Figure 6.1.8** *Hair at different distances from the viewer. The red square illustrates the screen size*

The picture above attempts to illustrate the visibility of hair in a scene from different distances. This is of course dependant on a number of factors in a game, like field of view, the size of the hair and how close the player gets to the hair. The pictures above are taken at a field of view of 45, with the size of the viewport illustrated as a red border. We use these distances in the measurements in this chapter. The closest hair is referred to as being at a distance of 70 cm from the viewer, the middle hairstyle at 1.4 m and the furthest at 7 meters. This reduces the requirements on the level of detail needed for a state of the art look. The figure below illustrates the same hairstyle with different polygon counts.

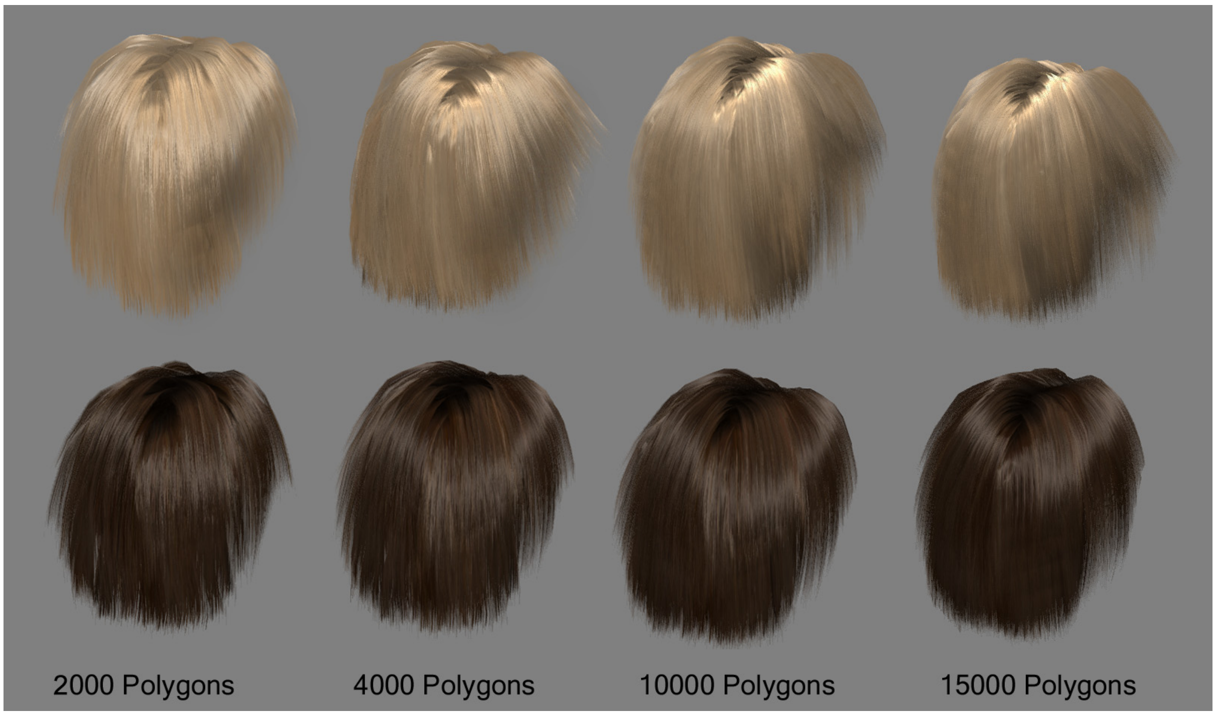


**Figure 6.1.9** *Illustration of two hairstyles using different polygon counts*

Comparing these pictures with the pictures of the same hair at different distances gives us an impression of how many polygons would be required. Subjectively even 4000 polygons would rarely be needed for anything but extreme close-ups and cut-scenes; in most situations 2000 polygons would probably be very sufficient. With today's hardware the main bottleneck of the technique presented in this document is the pixel shader. The measurements below illustrate this. They show times for the main draw call for the same hairstyle at different distances with two different polygon counts.

| | 70cm | 1.4m | 7.0m | | 70cm | 1.4m | 7.0m |
|---|---|---|---|---|---|---|---|
| Marschner | 21,7 | 6,1 | 0,8 | | 10 | 3,2 | 0,39 |
| Kajiya | 17,8 | 5,1 | 0,79 | | 9,1 | 2,5 | 0,38 |

***Figure 6.1.10*** *The draw call time of two different polygon counts were measured at distances roughly equivalent to real-life 70cm 1.4m and 7.0m at a field of view of 45. The figures to the left are for 10000 polygons, the ones to the right are for 4000 polygons. All times are in milliseconds.*

The measurements above clearly illustrate that there is a dramatic decrease in drawing time for the same geometry at different distances from the camera. This, together with indications from various diagnostics tolls lead us to the conclusion that the single most determining factor of draw call time is the amount of screen space the hair covers, or the number of times the pixel shader is called. We can safely say that the algorithm is pixel shader, or fill rate bound. We also see that a reduction of polygon count from 10000 to 4000 roughly halves the measured times. Due to the transparent nature of most of the content, super sampling, and the way the hair is built, with high depth complexity the increased polygon count directly influences the total pixel shader load.

Recent hardware has functionality that allows the pixel shader to be skipped completely, even with super sampling. This allows for the geometry to be rendered in a depth only pass, with transparency super-sampling. The depth test is then changed to equal and depth writes disabled for the main draw call. This allows a very simple shader to be run in the pre-pass. When the geometry is drawn to the screen only the final sub-pixels execute the pixel shader. Since the technique presented is clearly pixel shader bound a *pre-z* pass can significantly reduce the time taken to draw the hair.

| distance | far | medium | close |
|---|---|---|---|
| pre-z | 0,923 | 1,764 | 9,434 |
| no pre-z | 1,143 | 2,611 | 17,241 |

***Figure 6.1.11*** *Hairstyles with 4000 polygons drawn at varying distances, with and without a pre-z pass.*

The measurements above were done with the ATI setup. At the time of writing there was no way to accurately measure the draw call time on ATI hardware, and NVIDIA hardware did not support early-z required for the technique described in the above paragraph. Because of this the total frame time is presented instead, which includes some overhead. Despite of this, a rough forty percent performance increase can be seen for the closer distances when using a pre-z pass. This is a huge performance gain and again confirms that the technique is pixel shader bound. This is true even for larger distances, but to a lesser degree. Also, a low frame time of 0.923 milliseconds can be seen for furthest distance. If we turn the simulation off at the furthest distance, which also disables the simulation update pass, a total frame time of 0.417 milliseconds is achieved.

### 6.1.6 Totals

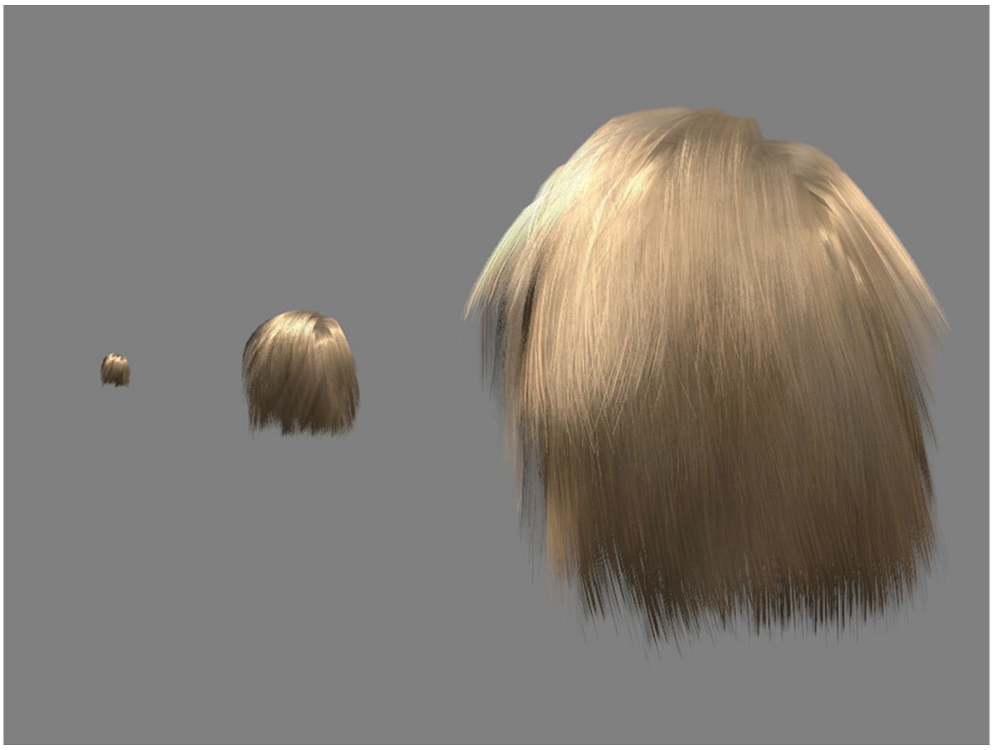


***Figure 6.1.12*** *A hairstyle with rendering and simulation level of detail adapted to fit the requirements of the distance*

For total time measurements in realistic circumstances, three different setups were introduced that fits the three measurement distances presented above. For the closest distance, a configuration which would fit a cut scene was used. As has been shown the medium distance is actually close for in-game purposes, and the furthest is a relatively normal in-game distance. The polygon count, shadowing technique and resolution, sorting and simulation were adapted to fit the requirements of the distance. A pre-z pass was used in all measurements

| | Polygons | Shadows | Sorting | Simulation | Frame Time (s) |
|---|---|---|---|---|---|
| Close | 7800 | Opacity 128x128 | On | On | 0,013889 |
| Medium | 2000 | Map 64x64 | Off | On | 0,001068 |
| Far | 1250 | Map 32x32 | Off | Off | 0,000408 |

***Figure 6.1.13*** *Measurements with input adapted to the requirements of the distances*

## 6.2 Evaluation

By looking at the presented techniques, a short evaluation was performed in collaboration with graphics/character artists to evaluate what is useable and/or visually acceptable.

### 6.2.1 Modeling

The hair creation process is greatly simplified by only creating and sculpting guide strands instead of creating all geometry manually, and working with textures and MAYA Paint gives an easy and intuitive way of changing the hair style over the scalp. Although the modeling is simplified it is still tedious work. More tools such as a comb, scissors and/or a solver for computing an initial hair style (for guide strands) given some force fields (for example gravity) would be preferable. To further increase control one would also enable the possibility of editing generated geometry and manually add new geometry to refine areas where it can be difficult to generate a perfect result from an artistic perspective. Such areas can be the edges of the hair and the fringe.

### 6.2.2 Rendering

From a visual quality perspective the demand for detail is heavily dependant of how large the rendering is, which of course depends on how far away the hair is rendered and the screen resolution. The image below shows three renderings of the same hair at different distances.

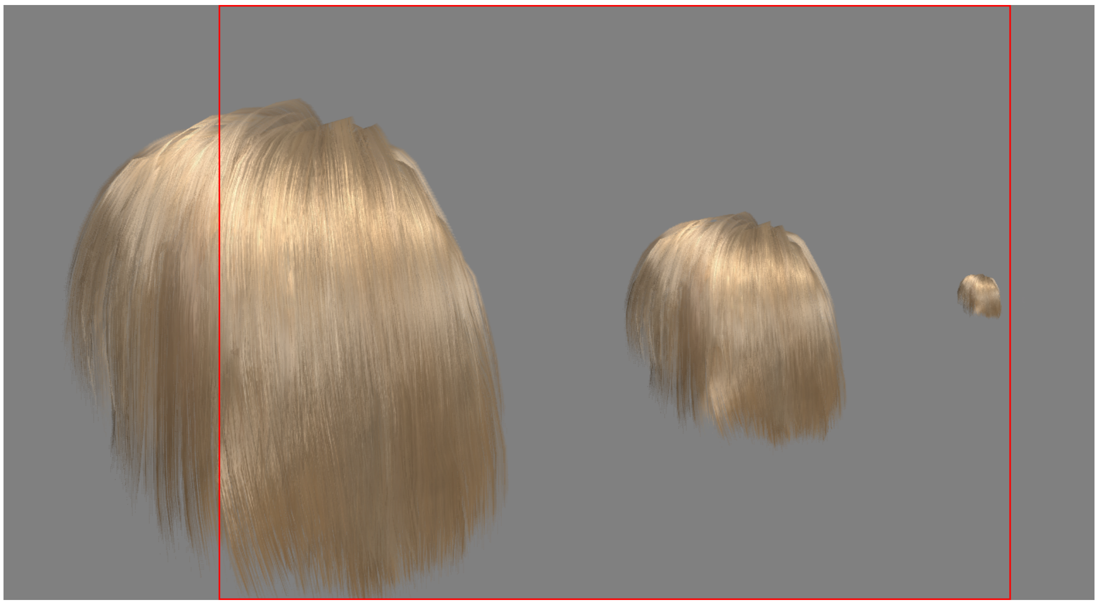


***Figure 6.2.1*** *Hair renderings from different distances. The red border indicates the viewport size.*

The left and middle renderings represent close distances while the right rendering represent a more realistic distance for a typical in-game scene. By looking at the renderings it is clear that the polygon count can be reduced substantially when viewed from a typical in-game distance. A polygon count somewhere between 1000 and 2000 should produce an acceptable result at such a distance. For close distances where the hair covers large parts of the screen a higher polygon count, 4000-8000, would have to be used to avoid loosing detail. An example of such a case would be a cut-scene.

Comparing the lighting models it is obvious that the Marschner model produces a more vivid result, which more closely resembles human hair than the Kajiya-Kay model which gives a more fur-like look. Even if the difference is more subtle at long distance one can still tell the techniques from each other. Since the rendering performance for the different lighting models is practically the same at long distances, there is no performance benefit for switching shading techniques if Marschner is the visually preferred one at close distance.

We found that dynamics is a very subjective thing to grade. As discussed in the previous simulation chapters, real-time simulation of a full head of hair is reduced to approximations. In addition to being three completely different approximations of reality, the three techniques used can be tweaked in different ways. This being said there are some traits that clearly differ regardless of parameters. The Forest model is stiffer than the other two; it has problems simulating inertia and other direct outer forces such as collisions. It does however act very well when affected by wind; this was a big part of the original paper presented by [DCF01], although for trees in a forest. The Cascading Spaces model is generally considered the most realistic looking model without requiring tweaking or exceptions for special cases. It is the more difficult of the models to force into a stiff appearance without introducing instability, so it is often more loose looking. The Simple Spring Model is as the name implies a very simple model. It could just as easily be said to resemble a bridge or linked chain. It requires customization to resemble moving hair, and once the parameters are good enough it may not act like hair in all situations, such as an upside down position. The special cases can be fixed though, as mentioned in chapter 4.4, and once they are, this simple model looks very good which the feedback reflected. It is also very stable.

Comparing the result from the shadow techniques by only looking at a shadow intensity rendering, there is a distinct difference, especially at close range. However when the lighting model is added to the rendering the difference becomes more subtle. There are still some visible artifacts but from a games perspective where artifacts are more accepted and one almost never see the character at such close range, the difference becomes practically unnoticeable. High definition shadows could be useful for cut scenes and other similar cases when the player is rendered at a very close range.

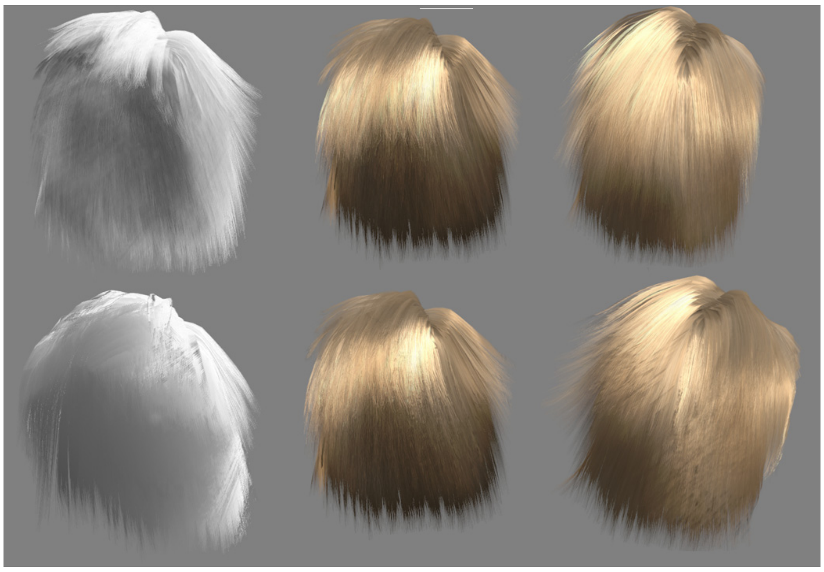


***Figure 6.2.2*** *Result of different shadowing techniques. The top row illustrates Opacity Shadow Maps, the bottom row Modified Shadow Maps*

## 6.3 Future Improvements

Beyond spring forces designed to keep a reference position, the implementation of the simulation presented in this thesis have constraints to the degree that individual strands and parts of strands can be declared as static. This has the effect that they always stay in the same position relative to the skull. One obvious improvement on this would be to introduce the ability to define constraints between simulated strands, allowing for more complex hairstyles. An example of such a hairstyle would be a pony-tail. This is possible to achieve with our simple constraints, but would introduce serious limits.

Inter-strand constraints could also be used to group wisps together in hairstyles which are less uniform. On a related note collisions between strands are ignored. Beyond simple proximity collisions between all neighboring strands, this could be improved in a number of ways. The Adaptive Wisp Tree model [BKNC03], described in previous chapters, assigns spheres of varying size, depending on the number of strands they represent, to their nodes. These spheres collide and act with viscous forces.

In addition to their shadowing algorithm, [BMC05] present a method for using the density in their voxel grid as input to a function that forces strands in high density voxels to separate to voxels with less density.

As mentioned a number of times in this thesis, none of the simulation methods are suitable for an implementation completely on hardware; the number of rendering passes required would be too great. However, other simulation methods may prove more suitable.

The fidelity of the opacity shadows described in this thesis is dependant on the number of render targets available. Every render target represents four depth layers. With current hardware it is possible to output to four render targets in one render pass. It is possible to use more targets with multiple passes, but this would have a serious impact on performance. DirectX 10 hardware will support eight render targets, thus doubling the number of depth layers that can be rendered in the same pass. This would still be slower than using four render targets and the lookup would be more complex. With the current optimizations eight interpolators would have to be used, but new hardware will provide new ways to optimize the lookup. One of the new features that will help in this regard is texture arrays. The reason for the optimizations in our implementation is interpolation between the two closest layers to the drawn pixel. This is exactly one of the benefits of texture arrays, an interpolated lookup can be made between two textures. In addition to eliminating at least two texture lookups this would eliminate the need for the use of four interpolators and four dot products. Branching some of the texture lookups could be done with today's hardware, but the performance impact of this has not been measured. The opacity shadow map algorithm increases the occlusion value in all layers behind the occluder, writing to the layers in front of the occluder is not necessary. However, today's hardware does not allow for branching writes to render targets, but future hardware could benefit from this. If this was the case, an adapted opacity shadow map algorithm could be conceived where the occlusion value is only written to the closest layer and the lookup reads from all layers in front of the light, shifting some of the computational weight to the lookup. This is generally a bad idea, since less pixels are usually rendered in the generation of the maps than in the lookup, but might provide benefits in some situations.

The ambient occlusion algorithm used was meant for a hardware implementation by the authors in [B05]. Our software implementation is much too slow for real-time calculations. The complexity of the algorithm is $O(n^2)$, but a way of organizing the data is presented in [B05] which reduces this to $O(n \log n)$. With the optimization it is claimed that dedicated hardware can process 500,000 vertices per second, this on older hardware. If the performance requirement is half a percent of the total frame time at 30 fps this would allow for about 800 vertices, which is not enough for our needs. In a couple of years time it could be possible to achieve the required level of performance for maybe 2000 vertices. The visual gain from using dynamic ambient occlusion would probably not be worth it for the distances where 2000 polygons is a useful complexity. A voxel grid technique might be more suitable, similar to the one presented in [BMC05]. This could be adapted to use the density value of nearby voxels as a basis for approximating ambient occlusion. To avoid a large number of lookups for each voxel, a multipass algorithm could be conceived where previously computed nearby densities could be used. If this could be used in combination with texture arrays, or if the voxel data could be spatially coherent this might allow for a real-time approximation.

As mentioned in the previous section, hair style creation needs to have a fast workflow. In addition to our implementation it could be very useful to implement tools for speeding up guide path modeling. Such tools could be implemented with the use of force fields. An example of this could be computing initial strand positions from gravity and self collisions or implement a comb with force fields.

Evaluation of the modeling system also showed that generating hair patches seldom gives a perfect result from an artistic point of view. The best solution to this would be to make the generated geometry editable and let the artist modify it or manually create new. Possibly one could use similar modeling tools as for guide paths.

The current implementation uses three different alpha maps (one texture with RGB-channels) for strand textures. These are selected by a color texture on the scalp. Ideally one would like to build a tool which bakes different strand textures to one large texture and re-computes uv-coordinates accordingly.

Tested hardware, NVIDIA 7800GT and ATI X1800, could only use alpha-to-coverage with respectively four and six samples. This only enables five and seven levels of opacity. More samples would produce smoother gradients. Furthermore, alpha-to-coverage does not currently work well with many semitransparent surfaces positioned on top of each other. Renderings of each surface will never take underlying (previously rendered objects) into account, which of course makes perfect sense since alpha-to-coverage aims at producing the same result regardless of rendering order. However this produces an artifact; many semi-transparent objects rendered on top of each other would never fill an entire pixel, which they would have done with blending. If future hardware would allow a mixture between alpha-to-coverage and blending functionality one could eliminate this artifact and produce better transparency renderings. If this will not be the case, there could instead be savings in performance if the graphics cards could have the capacity of rejecting individual pixels (without evaluating the pixel shader) depending on a z-buffer which has the same resolution as the screen buffer (with alpha test enabled). Currently a lot of pixels are not rejected until after the pixel shader is evaluated since the pre-pixel-shader z-test functionality only works with a “low resolution z-buffer”, (hi-z memory) when alpha testing is enabled. If a high resolution z-testing was available before evaluating the pixel shader a lot of the overhead from overdraw would disappear assuming that the rendered geometry is sorted front-to-back.

A number of features in upcoming hardware could possibly improve on some of the bottlenecks in our algorithm. In our case the balance between pixel shader and vertex shader is not ideal. By having a unified shader structure, used with new ATI graphics cards supporting DirectX 10 and XBOX360 graphics, where the pipelines are allocated by demand the performance would increase. Geometry shaders could allow for many new techniques for generating geometry and for handling level-of-detail scaling. The most mundane use would be to generate a higher tessellation of the individual strands. Other techniques may be developed where complete strands could be generated in-between other strands on demand. It might be possible to move the hardware update pass completely to a combination of vertex and geometry shader. The generation may require some processing power though and the buffer allocated must always be able to hold the highest polygon count.

Finally, a line drawing solution could be used in combination with the hardware update pass presented in this thesis. This might allow performance for line drawing closer to what is required for an implementation in a non-dedicated interactive environment than presented in previous work. NVIDIA uses 150000 strands in their Nalu demo. Assuming our hardware update scales linearly, this pass alone would take about 4 milliseconds for that vertex count, which is more than 10 percent of a 30 FPS frame. The Nalu hairstyle is relatively long though; a shorter hairstyle with say 75000 strands might work better, but would still be pushing the limit with today’s hardware in an environment which requires 95 precent of the time for other things.

## Acknowledgements

First of all, we would like to thank our advisors at Digital Illusions, Torbjörn Söderman and Jonas Kjellström for their invaluable advice throughout the thesis work.
We would like to thank the following people for their contributions and inspiration. Johan Andersson at Digital Illusions,, Petrik Clarberg, and Thomas Akenine-Möller at the Lund Institute of Technology, Ken Museth at Linköping University, Nicolas Thibieroz at ATI and Mark Harris from NVIDIA.

## References

[PF04] Matt Pharr, R. Fernando Ambient occlusion. *GPU Gems, 2004 p.667–692*

[B05] Michael Bunnell Dynamic Ambient occlusion and Indirect Lighting. *GPU Gems 2, 2005 p.223–233*

[KK89] James Kajiya, Timothy Kay Rendering Fur with Three Dimensional Textures. *Proceedings of ACM SIGGRAPH 1989.*

[MJC03] Stephen Marschner, Henrik Wann Jensen, Mike Cammarano Light Scattering from Human Hair Fibres. *Proceedings of SIGGRAPH 2003.*

[ND05] Hubert Nguyen, William Donnely Hair Animation and Rendering in the Nalu Demo. *GPU Gems 2, 2005 p.361–380*

[DBK03] Philip Dutré, Philippe Bekaert, Kavita Bala Advanced Global Illumination, *2003 AK Peters*

[KN01] Tae-Yong Kim, Ulrich Neumann Opacity Shadow Maps. *Proceedings of SIGGRAPH 2003*

[MKBR04] Tom Mertens, Jan Kautz, Philippe Bekaert, Frank Van Reeth A Self-Shadow Algorithm for Dynamic Hair using Density Clustering. *Eurographics Symposium on Rendering 2004*

[HW79] J. A. Hartigan, M. A. Wong K-Means Clustering Algorithm*, Applied Statistics Volume 28, No. 1, 1979 p. 100-108*

[BMC05] Florence Bertails, Clément Ménier, Marie-Paule Cani A Practical Self-Shadowing Algorithm for Interactive Hair Animation*. Graphics Interface May 2005, p. 71-78*

[S04] Thorsten Scheuermann Practical Real-Time Hair Rendering and Shading. *Proceedings of SIGGRAPH 2004*

[KHS04] Martin Koster, Jorg Haber, Hans-Peter Seidel Real-Time Rendering of Human Hair using Programmable Graphics Hardware. *Proceedings of Computer Graphics International 2004*

[CK05] Byoungwon Choe and Hyeong-Seok Ko A Statistical Wisp Model and Pseudophysical Approaches for Interactive Hairstyle Generation. *IEEE Transactions on Visualization and Computer Graphics, Volume 11, Issue 2, 2005 p. 160-170*

[KN02] Tae-Yong Kim and Ulrich Neumann Interactive Multiresolution Hair Modeling and Editing. *Proceedings of SIGGRAPH 2002*

[LD05] Ares Lagae and Philip Dutr´E A Procedural Object Distribution Function. *ACM Transactions on Graphics, Volume 24, Issue 4, 2005 p. 1442-1462*

[NV05] NVIDIA Corporation Technical Report: Antialiasing with Transparency, *2005 http://http.download.nvidia.com/developer/SDK/Individual_Samples/samples.html#AntiAliasingWithTransparency (Accessed 18/3 2007)*

[Y01] Yizhou Yu 2001 Modeling Realistic Virtual Hairstyles. *Proceedings of the 9th Pacific Conference on Computer Graphics and Applications 2001, p. 295*

[KH00] Chuan Koon Koh and Zhiyong Huang Modeling and Animation of Human Hair in Strips Computer. *Animation and Simulation, 2000 Springer-Verlag p.101-110.*

[WLLFM03] Kelly Ward, Ming C. Lin, Joohi Lee, Susan Fisher, Dean Macri Modeling Hair Using Level-of-Detail Representations. *Proceedings of the 16th International Conference on Computer Animation and Social Agents 2003*

[DCF01] Thomas Di Giacomo, Stéphane Capo, François Faure An interactive forest. *Eurographics Workshop on Computer Animation and Simulation sept. 2001, p. 65-74*

[BKNC03] Florence Bertails, Tae-Yong Kim, Marie-Paule Cani, Ulrich Neumann, Adaptive Wisp Tree - a multiresolution control structure for simulating dynamic clustering in hair motion. *ACM-SIGGRAPH/EG Symposium on Computer Animation 2003, p. 207-213*

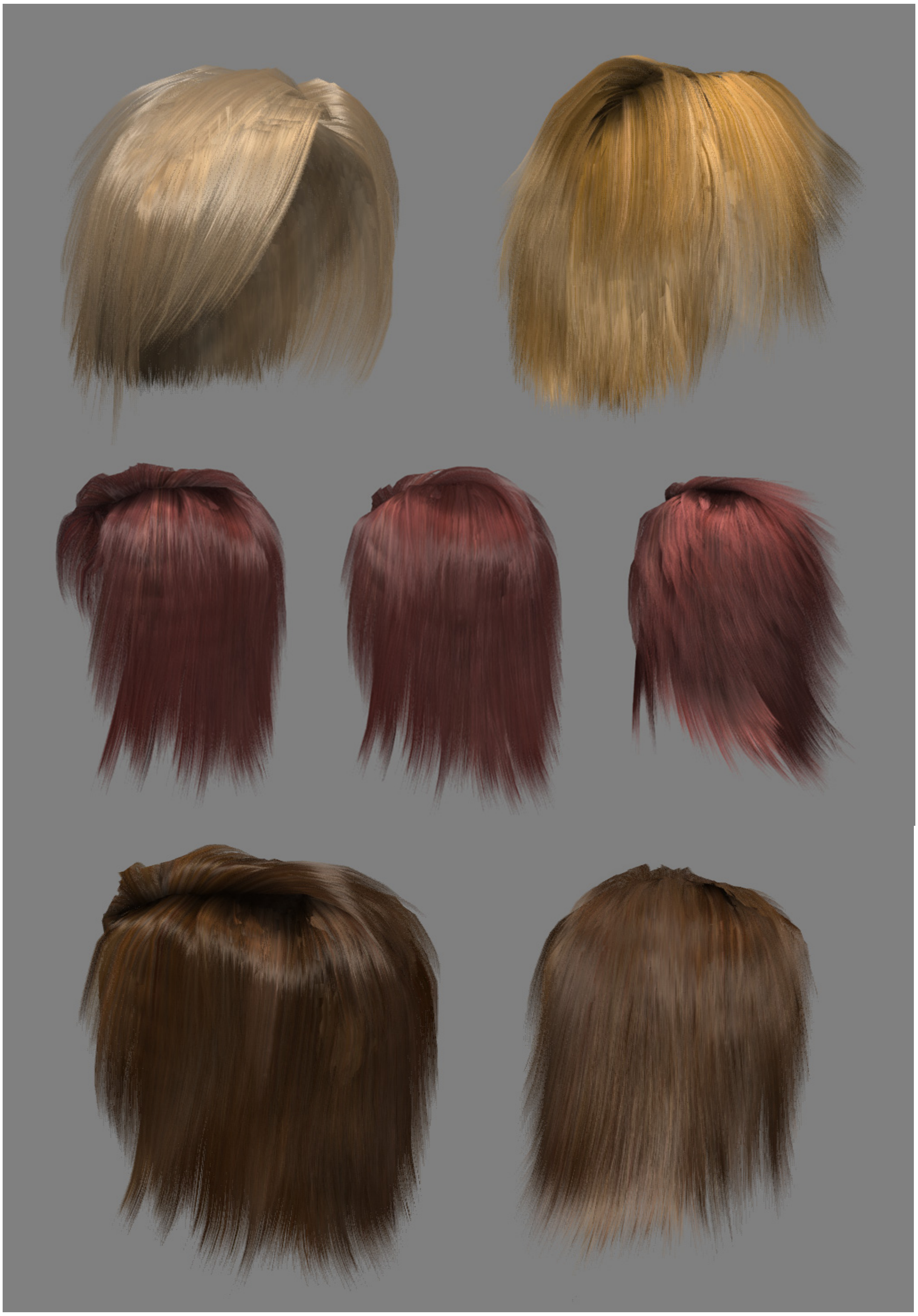
[VMT04] Pascal Volino, Nadia Magnenat-Thalmann Animating Complex Hairstyles in Real-Time . *Proceedings of the ACM symposium on Virtual reality software and technology 2004, p.41-48,*

[CJY02] Johnny T. Chang, Jingyi Jin, Yizhou Yu A Practical Model for Hair Mutual Interactions. *ACM-SIGGRAPH/EG Symposium on Computer Animation 2002, p. 73-80*

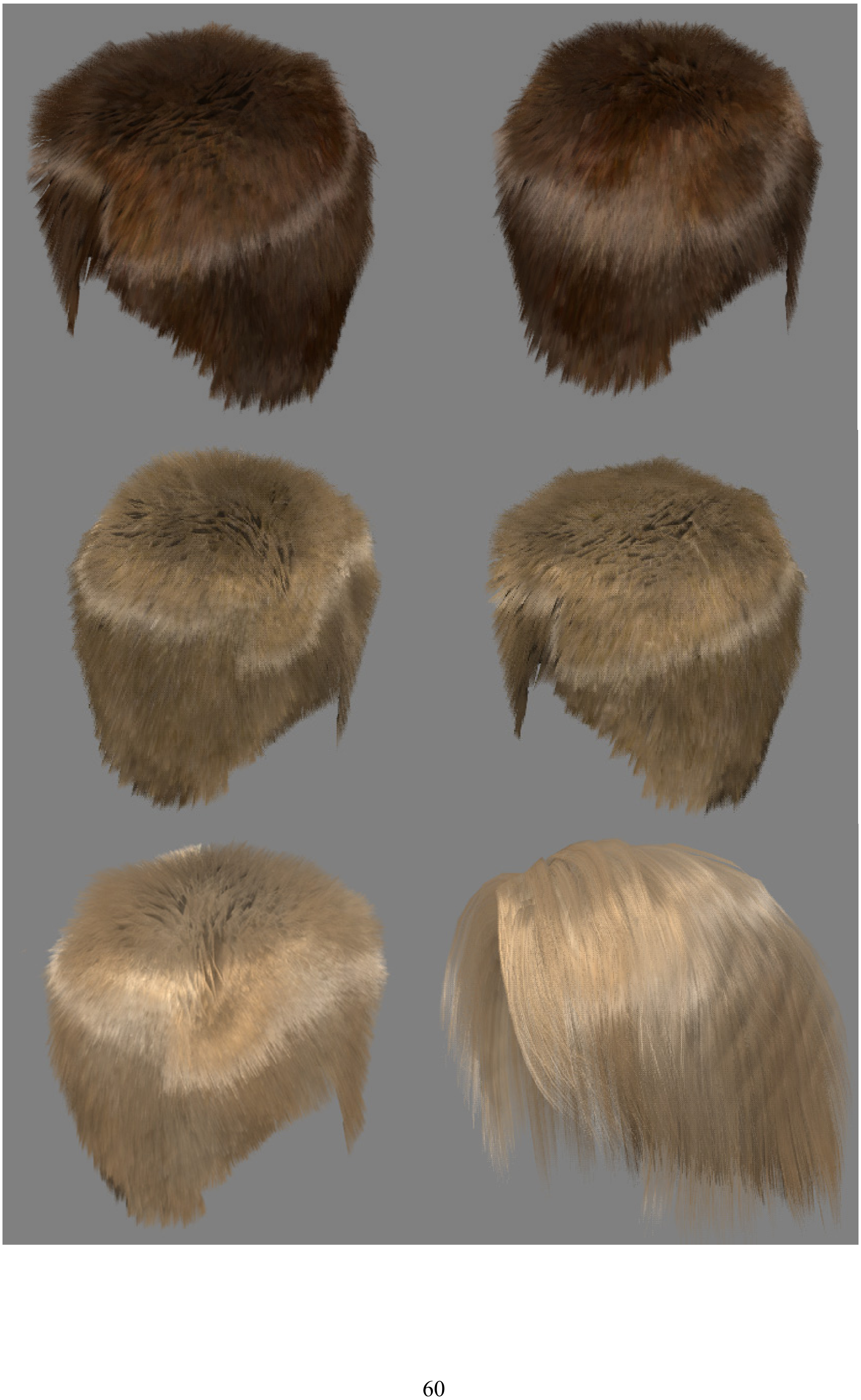
[J01] Thomas Jakobsen Advanced Character Physics. *Proceedings of GDC 2001, Lecture*

[V67] Verlet, L. Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules. *Physics Review 159, 1967 p. 98-103*

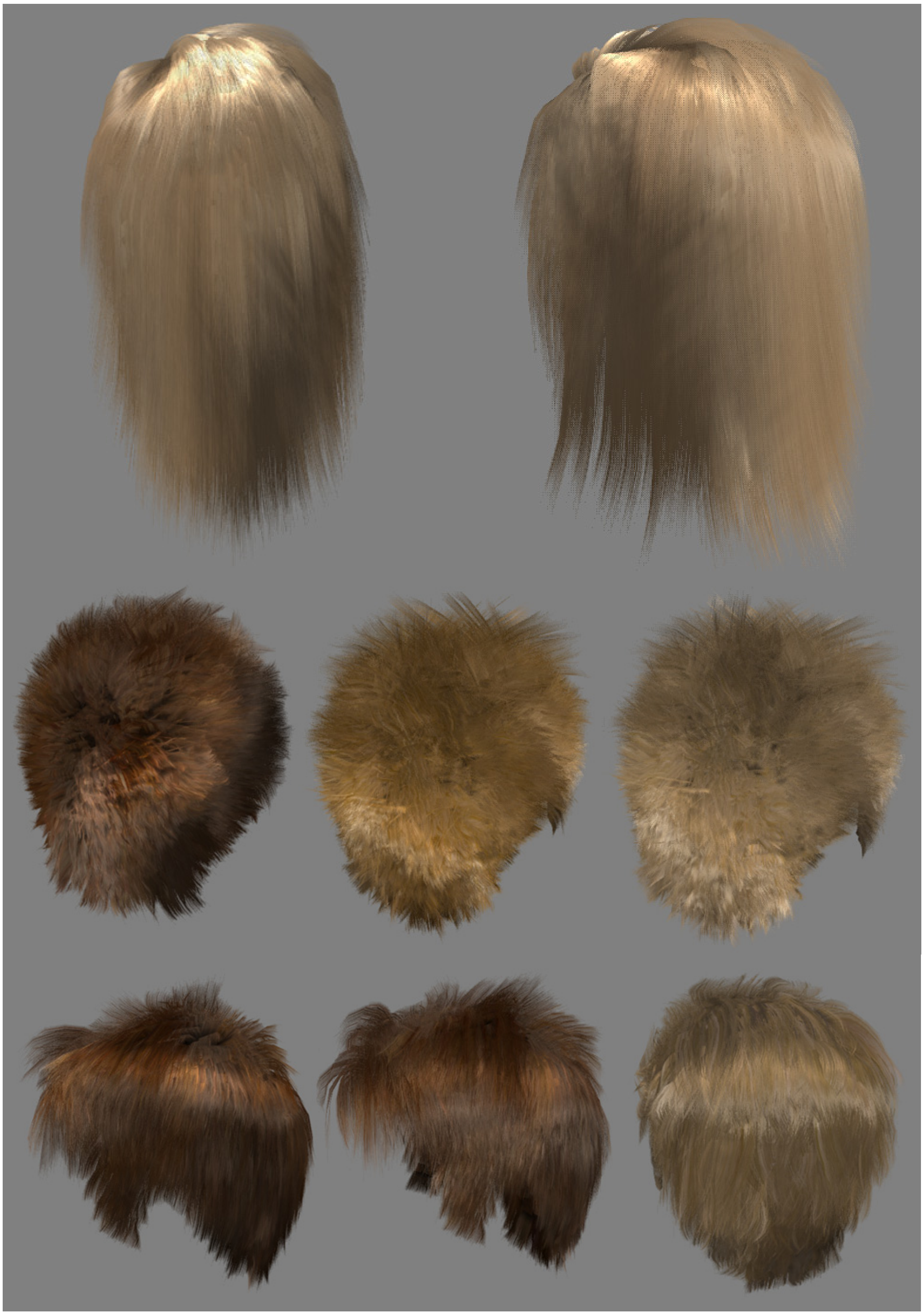
## Appendix – Gallery



Henrik Halén, LTH, Sweden
Martin Wester LiTH, Sweden



Henrik Halén, LTH, Sweden
Martin Wester LiTH, Sweden



Henrik Halén, LTH, Sweden
Martin Wester LiTH, Sweden