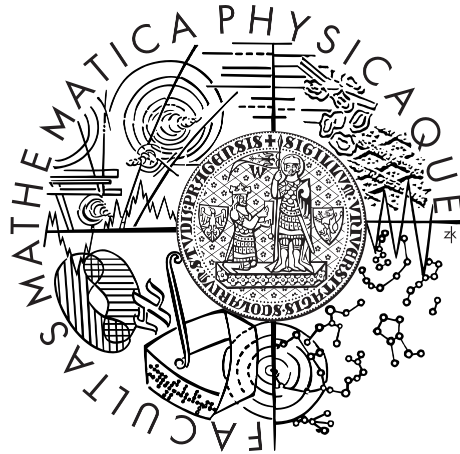
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Tomáš Svoboda

Realistic hair rendering in Autodesk Maya

Department of Software and Computer Science Education

Supervisor of the master thesis:	Ing. Jaroslav Křivánek, Ph.D.
Study programme:	informatics
Specialization:	software systems

Prague 2012

I would hereby like to thank my supervisor Jaroslav Křivánek for his supervision and counselling which guided me throughout my work.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In .... date .... signature of the author

Název práce: Realistic hair rendering in Autodesk Maya
Autor: Tomáš Svoboda
Katedra: Kabinet software a výuky informatiky
Vedoucí diplomové práce: Ing. Jaroslav Křivánek, Ph.D., Kabinet software a výuky informatiky
Abstrakt: Tato diplomová práce popisuje real-time zobrazování vlasů v 3D modelovacím programu *Autodesk Maya*. Zobrazovací modul je součástí projektu *Stubble* – zásuvného modulu do programu *Maya*, který slouží k modelovaní vlasů. Prezentovaný algoritmus poskytuje vysoce kvalitní interaktivní náhled, pomocí kterého je možné modelovat vlasy bez nutnosti zdlouhavého vytváření náhledů v externím programu. Cílem je vytvořit takový náhled, který se bude co nejvíce podobat obrázkům, které produkuje *3Delight* - zásuvný modul pro program Maya, který implementuje standardy zobrazovacího rozhraní *RenderMan*.
Klíčová slova: počítačová grafika, 3D, vlasy, real-time renderování

Title: Realistic hair rendering in Autodesk Maya
Author: Tomáš Svoboda
Department: Department of Software and Computer Science Education
Supervisor: Ing. Jaroslav Křivánek, Ph.D., Department of Software and Computer Science Education
Abstract: This thesis describes a real-time hair rendering in 3D animation and modeling software *Autodesk Maya*. The renderer is part of the *Stubble* project – a *Maya* plug-in for hair modeling. The presented renderer provides a high-quality interactive preview that allows fast hair modeling without the need for rendering in slow off-line renderers. The goal of this work is to create a renderer that can generate images in real-time that are as close as possible to the output of the *3Delight* renderer - a plug-in for *Maya* that is based on *RenderMan* standards.
Keywords: computer graphics, 3D, hair, real-time rendering

Contents

Introduction **3**

1 Theory **5**

1.1 Overview . . . 5
1.2 Hair fiber graphical representation . . . 5
1.2.1 Problem overview . . . 5
1.2.2 Drawing primitives . . . 5
1.2.3 Line width computation and anti-aliasing . . . 7
1.3 Alpha blending . . . 11
1.3.1 Problem overview . . . 11
1.3.2 Exact solution . . . 11
1.3.3 Sorting approximation . . . 13
1.3.4 Order independent transparency . . . 15
1.3.5 Discussion . . . 16
1.4 Light-scattering model of a hair fiber . . . 16
1.4.1 Kajiya-Kay model . . . 16
1.4.2 Marschner model . . . 18
1.4.3 Multi-scattering models . . . 18
1.4.4 Discussion . . . 19
1.5 Self-shadowing in hair . . . 19
1.5.1 Depth-based shadow mapping . . . 19
1.5.2 Deep Shadow Maps . . . 20
1.5.3 Opacity Shadow Maps . . . 20
1.5.4 Deep Opacity Maps . . . 22
1.5.5 Occupancy Maps . . . 23
1.5.6 Discussion . . . 26

2 Implementation **27**

2.1 Autodesk Maya . . . 27
2.1.1 Viewport 2.0 . . . 27
2.2 Stubble . . . 28
2.2.1 Implementation of Stubble in Maya . . . 28
2.2.2 Hair generation in Stubble . . . 28
2.2.3 Default interactive renderer in Stubble . . . 30
2.2.4 RenderMan and Stubble . . . 30
2.3 Architecture of the presented renderer . . . 31
2.3.1 Rendering in Viewport 2.0 . . . 31
2.3.2 Problems with Viewport 2.0 . . . 31
2.3.3 Shader programs . . . 32
2.3.4 HairDrawOverride . . . 33
2.3.5 HairModel . . . 34
2.3.6 Lights . . . 36
2.4 Implementation of hair fiber representation . . . 37
2.5 Implementation of hair alpha blending . . . 39
2.5.1 Hair sorting . . . 39

2.5.2 Pre-muliplied alpha 40
2.5.3 Multiple lights 42
2.6 Implementation of light model 43
2.6.1 Light model in 3Delight 43
2.7 Implementation of Deep Opacity Maps 43
2.7.1 Setting camera matrices 44
2.7.2 Rendering maps 44
2.7.3 Final hair rendering pass 45

3 Results **47**
3.1 Testing environment 47
3.2 Performance 47
3.2.1 Hair fiber representation 47
3.2.2 Hair segment sorting 49
3.2.3 Self-shadowing 50
3.2.4 Multiple lights 52
3.3 Visual results 53
3.3.1 Differences from 3Delight 53
3.3.2 Final hair rendering 55

Conclusion **58**

Bibliography **60**

A Attached CD's content **62**

Introduction

Hair is a very important visual aspect of human head and represents arguably the most challenging problem during the rendering of a human body. Our eyes are very sensitive to any inaccuracies in hair appearance. Hair geometry is very complex as it contains up to 150.000 very thin hair strands with complicated light-scattering properties. Rendering of animals is not any easier as the animal fur can contain millions of strands. As such, the rendering of convincing images of hair or fur remains a very challenging problem, especially in real-time graphics. Conventional rendering techniques often fail due to poor performance and aliasing problems. This thesis presents techniques for rendering images of hair at interactive time, that are very similar to those rendered in high-fidelity off-line renderers, such as *3Delight*[1]. See Figure 1 for illustration.

Figure 1: 100k hairs rendered by the *3Delight* (on the right) and by the presented renderer in *Autodesk Maya 2012*.

The presented renderer is implemented in the environment of *Autodesk Maya*[2] – a widely used tool for modeling and animating 3D scenes. The hair geometry is generated in the *Stubble* project, which is a plug-in for *Maya* for modeling hair.

In order to render a plausible looking image of hair, there are several issues that need to be solved:

Hair fiber graphical representation. The hair fiber resembles a very thin cylinder but using cylinders for drawing hair is slow and therefore not suitable for real-time rendering. Another issue here is aliasing. Normal hair strand is much thinner than the pixel width and even with multi-sampling, current hardware

[1] `http://www.3delight.com/`
[2] `http://usa.autodesk.com/maya/`

cannot capture the high-frequency geometry of hair. Alpha blending is normally applied to approximate anti-aliasing but it raises other problems.

Alpha blending. As stated above, alpha blending is required to eliminate severe aliasing. Also, blond or light-colored hair fibers are semi-transparent, which in itself requires alpha blending. Proper usage of alpha blending to achieve transparency effects usually requires that the fragments are drawn in back-to-front order, which is not easily realizable. Several ways to address this problem is presented.

Light-scattering model of a hair-fiber. The light-scattering, especially in blond hair, can be complex as it produces both reflection and refraction of incoming light. Fast implementation of a plausible light-scattering model is essential for interactive hair rendering.

Self-shadowing. Hair fibers cast shadows onto each other, as well as receive and cast shadows from/to other objects in the scene. Self-shadows are very important for recognizing the shape of hair. It is crucial to implement hair shadowing to provide usable preview during hair modeling. Several methods for hair self-shadowing are presented in this thesis.

Thesis overview

This thesis consists of following chapters:

- **Theory**: This chapter discusses possible algorithms for different aspects of hair rendering. Related work is presented individually in each section of this chapter.
- **Implementation**: Here is presented the actual implementation of the renderer in the environment of *Autodesk Maya* and *Stubble*.
- **Results**: This chapter shows visual results and performance of the final hair renderer in comparison with off-line renderer *3Delight*.
- **Conclusion**: The conclusion of my thesis and possible future work.

1. Theory

1.1 Overview

There are several sections in this chapter that describe the theory required for solving separate aspects of the hair rendering. All of those sections show methods presented by other authors except Section 1.2.3, which is my own work.

1.2 Hair fiber graphical representation

1.2.1 Problem overview

The hair geometry is very complex. There is about 100 - 150 thousand hairs on a human scalp and animal fur can consist of millions of strands. Even the geometry of a single hair fiber is not trivial. A long curled hair is geometrically complex on its own. If each hair was drawn using 100 triangles, it would cause a serious slowdown even on modern hardware.

Besides that, the hair strand is very thin in diameter (under 0.1 mm) and can be very long. This causes an under-sampling problem since pixels are typically much thicker. Hardware multi-sampling does solve this issue. Super-sampling by rendering to an off-screen buffer would help but it would significantly increase the number of generated fragments, and since fragment shaders for hair are normally quite time-consuming, this solution would bring severe performance issues.

1.2.2 Drawing primitives

A hair fiber is a curved cylinder, see Figure 1.1. Tessellated version of a cylinder is a valid representation of a hair but has its drawbacks. Such cylinder would consist of many triangles and rendering many thousands of cylinders would be too slow. Also, the problem with aliasing would lead to a low-quality result. Tessellated cylinders would be a good approach for rendering a limited number of thick fibers, or when rendering a detail of a human scalp.

Another approach is to approximate a hair as a flat ribbon. It is a good approximation but requires the ribbon to be facing the camera. This could be done in hardware shader but the aliasing problem still exists. Non-rotating ribbons are used as a graphical representation in default *Stubble* renderer and are also available in my renderer, because they can preserve hair thickness even if the width is different at the hair root and the hair tip. Results of rendering typical hair with this representation are not very good though, see Figure 1.2.

Further simplification leads to a line strip. Its primitive count is halved compared to the ribbon representation. A line mathematically does not have any width, but in the case of hardware rendering, the line width is non-zero and relative to a pixel width. This causes an inconsistency between the hair's width in 3D world space and the projected line's width in 2D screen space. The line's projected width is constant with distance from the camera plane (plane perpendicular to the camera direction containing the camera position) and with camera's field of view. It is, on the other hand, not constant with the viewport resolution. This

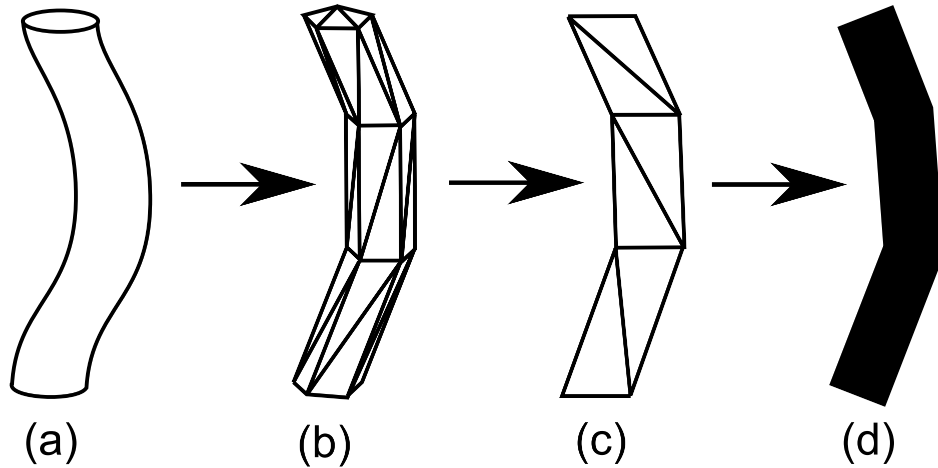


Figure 1.1: Possible hair graphic representations. On the left is the exact hair shape (a), simplified by tessellated cylinder (b), flat ribbon (c), and the line strip (d).

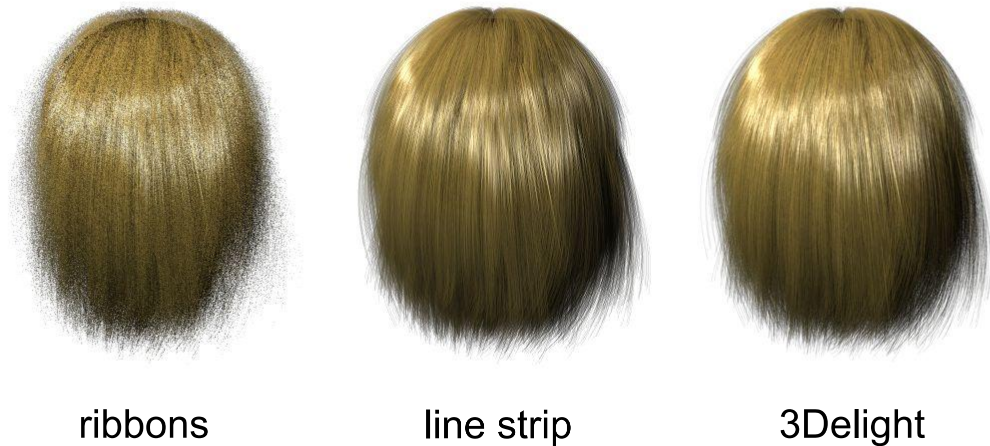


Figure 1.2: Image of hair rendered with the ribbon and line strip representation compared to reference rendered in 3Delight.

means that if the resolution is higher, the line of a given width in terms of pixels appears thinner on the screen. See Figure 1.3 for illustration. It is also not possible to represent different width of hair tip and hair root with a line strip because the line width is an attribute of the graphics pipeline that cannot be efficiently changed on a per-primitive basis.

Graphics hardware can do fast line anti-aliasing. It allows fairly smooth drawing of lines with various widths based on the pixel width. However, it puts some conditions on the rendering process. It requires lines to be drawn from back to front with specific blending parameters. Those parameters are not, unfortunately, suitable in some cases. Details about this issue are discussed in section Section 2.5.2. Also, the capabilities of the hardware line anti-aliasing are very dependent on the actual hardware. Normally, drawing lines that are thinner than a pixel, is not supported. Since thin hairs are often projected to a screen area that is smaller than a pixel, the hardware line anti-aliasing does not solve the problem of the hair width inconsistency.

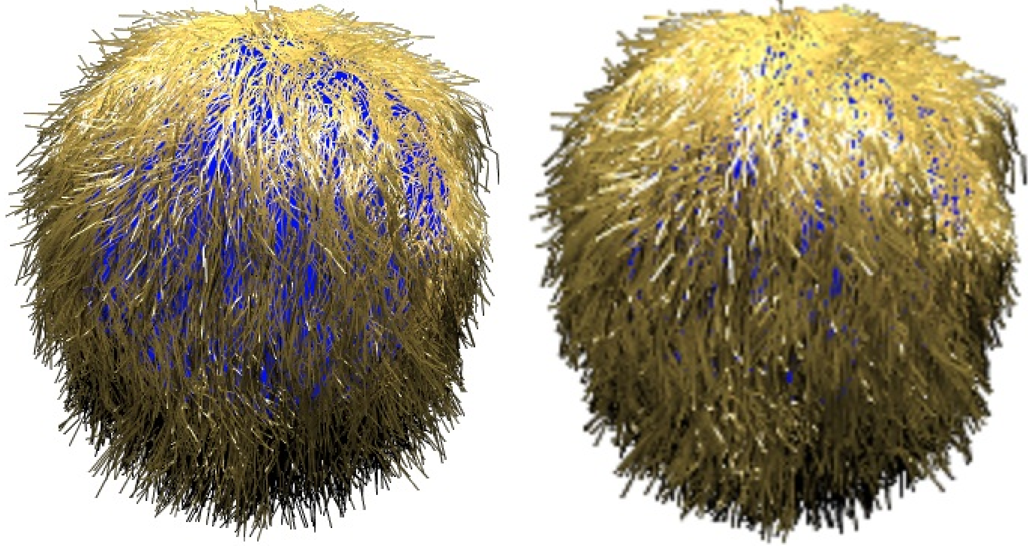


Figure 1.3: The difference between line width based on the resolution. The image on the left is rendered at a resolution of 640×480. The image on the right is rendered at a resolution of 320×240

In the next section, I described an algorithm I developed in order to cope with the problem of the hair width inconsistency and aliasing.

1.2.3 Line width computation and anti-aliasing

As shown in Figure 1.3, the inconsistency of projected hair width can totally disrupt the hair volume. It would not be acceptable if the image looked completely different based on its resolution.

In order to draw hair fibers with correct width, it is necessary to find the screen area, to which the fiber is projected. With that, it is possible to compute the intersection between the projected fiber and each pixel the fiber projects to. The final color addition of the fiber to the pixel can be computed as:

$$C_a = I_s/P_s \cdot C_f$$

where I_s is the size of the intersecting area between the projected line and the pixel, P_s is the size of the pixel, and C_f is the color of the hair fiber. See Figure 1.4.

The implementation on CPU of the intersecting area computation would not, however, be very effective. The hardware line anti-aliasing does this but with some restrictions as mentioned above. By a considerate approximation, it is possible to achieve a plausible and fast solution.

Dealing with sub-pixel width

If the projected width of a hair fiber is smaller than the pixel width, I propose to approximate the intersection of the projected fiber with the pixel simply by the fiber's projected width. The equation for the hair fiber addition to the pixel color is changed to:

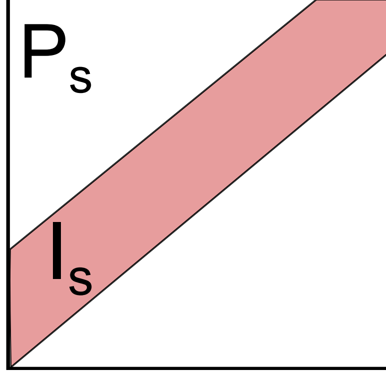


Figure 1.4: Projected line going through a pixel. P_s is the pixel size, I_s is the size of the intersected area.

$$C_a = \frac{F_{pw}}{P_w} \cdot C_f$$

where P_w is the pixel width and F_{pw} is the projected width of a hair fiber. This equation could be modified to:

$$C_a = \frac{F_w}{P_{3dw} \cdot d} \cdot C_f$$

where P_{3dw} is the width of a square at distance 1 from the camera plane, that is projected on the screen to an area that exactly corresponds to one pixel. d is the distance of the hair fiber from the camera plane.

The P_{3dw} is computed from the viewport frustum and resolution. The viewport frustum is a rectangular pyramid that is cut by two planes (near plane and far plane), which are perpendicular to the view direction, see Figure 1.5.

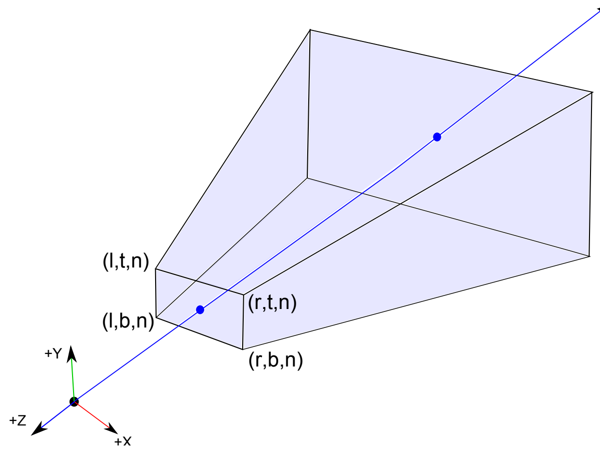


Figure 1.5: Viewport frustum.

The width of a square at distance 1 from the camera plane is given by:

$$P_{3dw} = \frac{(r - l)/n}{Res}$$

where r and l are the right and left boundaries of the near plane, respectively, n is the distance from near plane to the camera plane and Res is the horizontal viewport resolution.

This approximation assumes that the line is either horizontal or vertical. If the line is diagonal, the final color addition is underrated. That is because the intersection area of a horizontal/vertical line and the pixel is smaller than the intersection area of a diagonal line. Let us assume that the line goes through the center of a pixel as shown in Figure 1.6. The relative intersection area of the horizontal line is:

$$I_{ha} = \frac{F_w}{P_{3dw} \cdot d}$$

The relative intersection area of the diagonal line would be:

$$I_{da} = \frac{\sqrt{2} \cdot F_w}{P_{3dw} \cdot d} - \frac{F_w^2}{2 \cdot P_{3dw}^2 \cdot d^2}$$

So the ratio of I_{da} and I_{ha} is:

$$\frac{I_{da}}{I_{ha}} = \sqrt{2} - \frac{F_w}{2 \cdot P_{3dw} \cdot d}$$

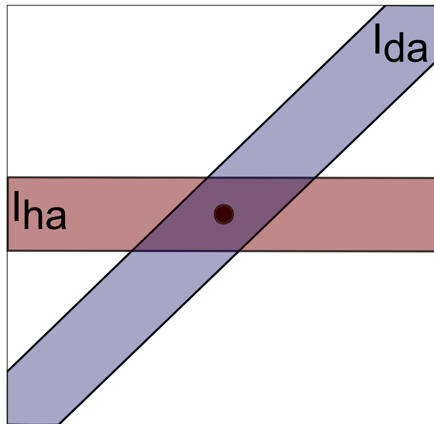


Figure 1.6: Horizontal and diagonal line going through the pixel center.

In practice, an accurate computation of the ratio that is based on the line's angle is not necessary. I suggest that the ratio is approximated as a constant, independent of the actual angle as follow:

$$Rat = \frac{I_{da} + I_{ha}}{2 \cdot I_{ha}}$$

This formula gives a correct result on average and the actual error is very small.

The final equation of the color addition of the fiber to the pixel is computed as:

$$C_a = \frac{F_w}{P_{3dw} \cdot d} \cdot Rat \cdot C_f \tag{1.1}$$

P_{3dw} must be computed only when the viewport is changed. The rest of the equation changes for every fragment. By exploiting the fact that the C_a changes linearly through the line segment, it is possible to resolve the equation just in the line vertices and linearly interpolate for every fragment of the line. Therefore, it can be easily implemented in the hardware vertex shader. For further details on implementation, see Section 2.4.

Setting correct line width

Above presented equations work only for fibers that have the projected width smaller than the pixel width. If the projected width is larger than the pixel width, it is necessary to set the line's pixel width before rendering. The equation for line width is very similar to equation 1.1:

$$L_w = \frac{F_w}{P_{3dw} \cdot d}$$

Getting the distance between the hair fiber and the camera plane (d) is not so easy here. The L_w value is required before the rendering, thus it cannot be computed in the vertex shader. The hair sorting algorithm presented in Section 1.3.3, which is used to approximate alpha blending, helps with that. It divides the hair geometry into slices, based on the distance from the camera plane. The slices are then drawn in back to front order. This can be easily exploited and the line width can be set before each slice is rendered. See Section 2.4 for details.

Using a line width other than 1 has different effects, depending on whether the hardware line anti-aliasing is enabled. If the line anti-aliasing is disabled, the line width is rounded to the nearest positive integer.

Computing the line width correction is also important when rendering shadow maps, discussed in Figure 1.5. Without the corrections, the shadow maps with different resolution would produce different self-shadowing, see Figure 1.7.

Figure 1.7: Hair shadowed by shadow maps with different resolutions and with and without line width correction.

Anti-aliasing

The computation of hair fragment addition does not, by itself, entirely solve the problem of aliasing. However, it allows using the line strip as geometry primitive for fibers with sub-pixel projected width. The line strip does always rasterize into a fragment, to which is the hair fiber projected, as opposed to other representations, see Figure 1.8 (a) and (d). This alone significantly reduces

the aliasing in the hair geometry, see Figure 1.2. Proper anti-aliasing can be achieved by combination with hardware anti-aliasing. It works well with both line anti-aliasing and multi-sampling techniques, see Figure 1.8.

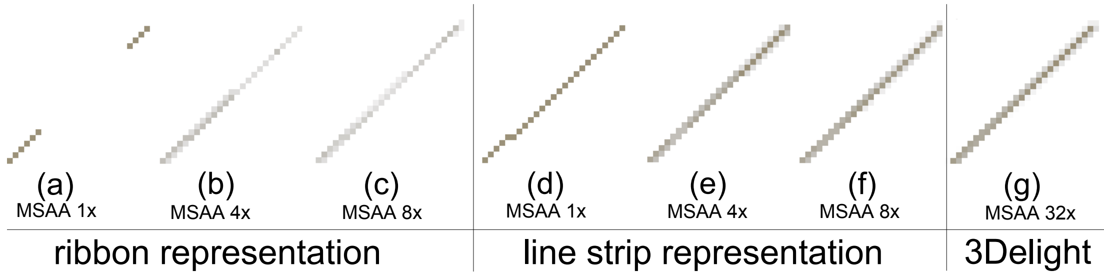


Figure 1.8: Zoomed image of a thin hair rendered with different hair representations and different multi-sampling settings (MSAA). Note that ribbon without multi-sampling (a) cannot capture the thin hair and even 8x MSAA does not give accurate result. On the other hand, the line strip is plausible without MSAA and with 4× and 8× MSAA gives a fairly accurate result.

Discussion

The main contribution of the method proposed in this section lies in the approximation of the conversion between a screen space width and world space width, which allows rendering sub-pixel geometry, such as hair fibers, with a line strip. Together with hardware anti-aliasing, it produces very accurate results. A small error is produced by the approximation of *Rat* ratio, but the *Rat* is correct on average. Results are illustrated in Figure 1.9.

1.3 Alpha blending

1.3.1 Problem overview

Alpha blending is the process of combining a translucent foreground color with an opaque background color, which produces a new blended color. It is necessary for rendering light-colored semi-transparent hairs. As stated in section Section 1.2, it also helps in anti-aliasing thin hair fibers. Modern graphics hardware can deal very efficiently with alpha blending, but usually requires translucent fragments to be drawn in the order from the farthest to the nearest to the camera plane, see Figure 1.10. When considering hundreds of thousand of line segments needed for representing full human hair, it is clear that interactive rendering requires a fast solution to this problem. Note that translucency techniques that do not require fragment sorting, exist, see Section 1.3.4.

1.3.2 Exact solution

To ensure that all fragments are drawn in a back to front order, a technique called Depth-peeling by Everitt (2001) [1] is commonly used. This method draws the geometry multiple times with depth test set to pass fragments with larger or equal z value than the one in the depth buffer. The next render pass uses

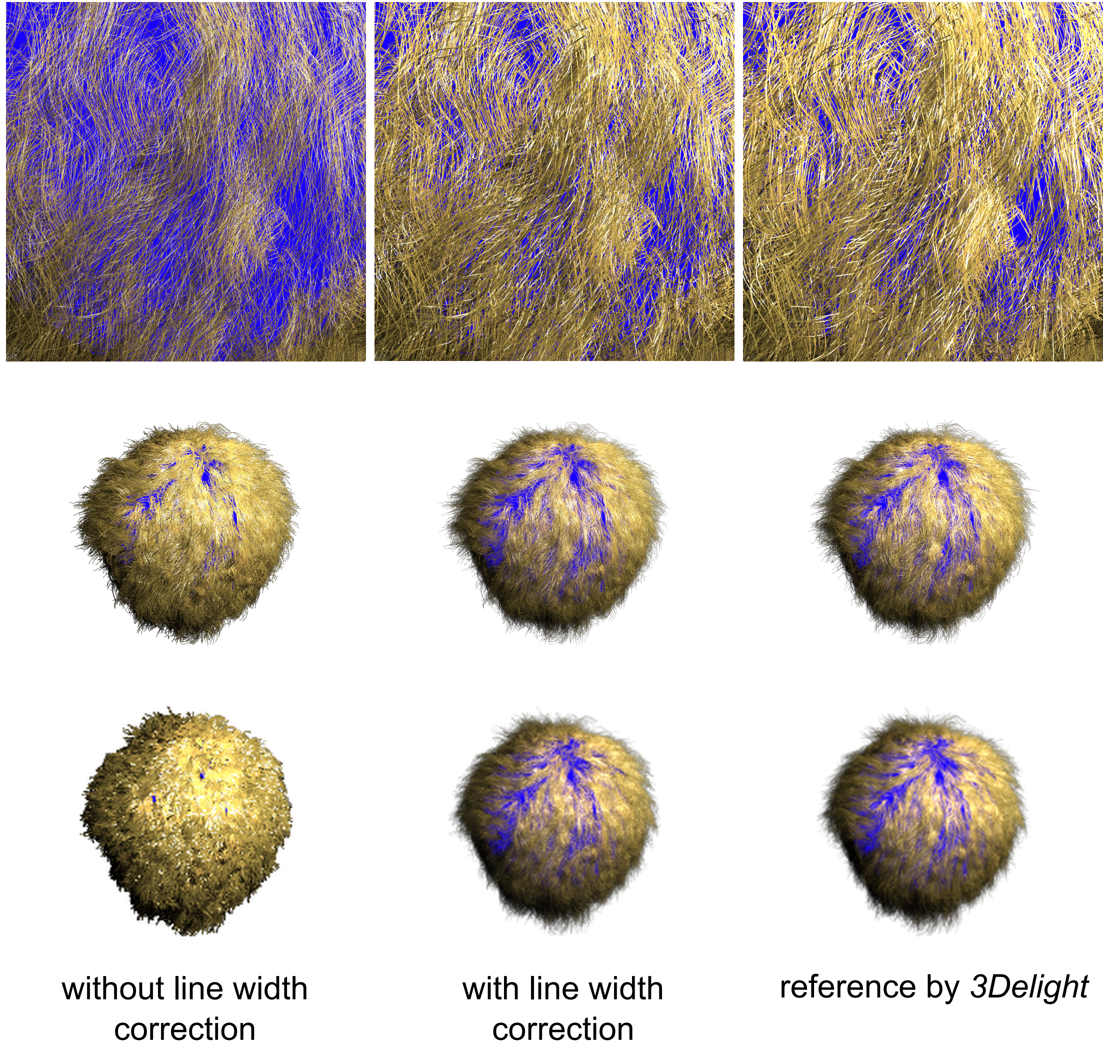


Figure 1.9: Images of hair taken from different distances. Top images show a close-up view, middle images show middle distance and bottom images are rendered from large distance and then zoomed in. Note how the images rendered without the line width correction does not maintain hair width with different distances.

the depth buffer of the previous pass. This way, each render pass peels off one layer. However, the number of render passes needed for rendering thousands of hairs can easily reach hundreds, making this technique unusable for interactive rendering.

Several attempts to speed up the depth-peeling were made. Liu et al. (2006) [2] proposed to use multiple render targets for sorting the fragments in the fragment shader. The theoretical 8 times speed-up is rarely achieved in practice due to the possibility of read-modify-write hazards.

[3] suggested a the technique that peels off the front-most and further-most layer. It doubles the performance of the original depth-peeling algorithm, but when rendering geometry with high depth-complexity such as hair, the speed is not sufficient for interactive use.

Figure 1.10: Hair geometry rendered without sorting produces very bad results. Approximate line segment sorting into 256 bins shows a result very similar to the reference rendered by 3Delight.

A faster solution would be to explicitly sort all the geometry segments by distance from the camera plane, but sorting the sheer number of hairs is not very fast either. On top of that, it is not always possible to figure out the rendering order of segments in a way that it gives a correct result. See Figure 1.11.

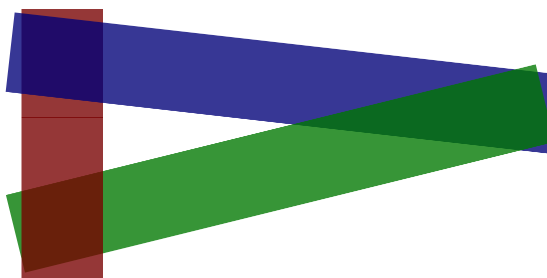


Figure 1.11: Three lines, each of which is both covered by and covering other line. There is no correct rendering order for those lines.

It turns out, however, that exact sorting is not necessary in order to achieve a convincing image of hair. If we put up with approximate hair ordering, it is possible to attain a fairly fast sorting algorithm.

1.3.3 Sorting approximation

As presented by Kim (2003) [4], one way of approximating hair sorting is that the hair geometry is divided by planes perpendicular to the camera direction. Each bin, a volume bounded by a pair of adjacent planes, stores indices of hair segments whose farthest end point is contained by the bin, see Figure 1.12.

The index of the bin enclosing each point is found by

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

where i is the index of the bin and N is the total number of bins. D is the distance from a hair point (corresponds to a vertex in the line strip) to the camera

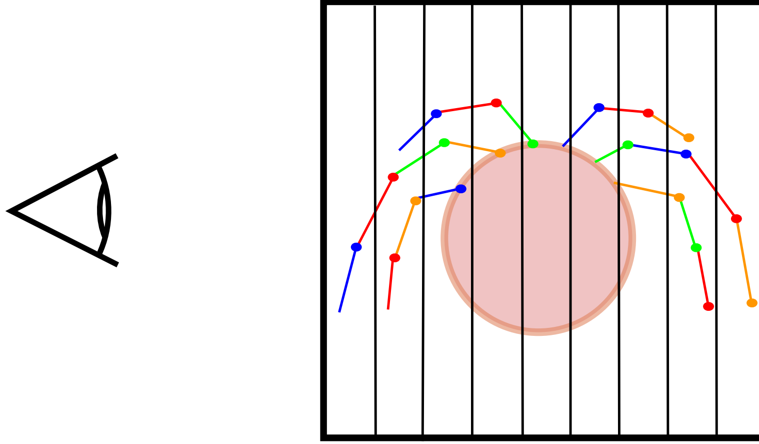


Figure 1.12: Hair volume divided into bins. Each hair segments is assigned by its farthest point.

plane. D_{min} is the distance between the camera plane and the first bin and D_{max} is the distance to the last bin.

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

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

When hair segments are nearly parallel to the image plane, the number of incorrectly ordered segments is small if the slices are close together. However, when segments extend over many bins, the visibility order cannot be determined either by maximum depth or minimum depth of the segment. On the other hand, the pixel coverage of such a segment is usually very small. For example, when a line is perpendicular to the image plane, the pixel coverage of the line is a single pixel. In practice, using maximum depth for every segment produces good results. Generally, the shorter the segments are and the smaller the distance between slices is, the more accurate result is produced.

The algorithm has a time complexity of $O(n)$, where n is the number of hair segments. It is invariant to the number of bins, so the slices could be dense enough (however, high numbers of bins can slow down the rendering). 256 bins seems to be sufficient in most cases, see Figure 1.10. Rendering of hair is done by drawing the hair segments contained in bins. First goes the farthest bin and last the nearest bin.

Writing to the depth buffer should be disabled when rendering hair as it produces much better results. Let u imagine two overlapping lines that are drawn in wrong order, as illustrated in Figure 1.13. The correct result of blending those two lines would be:

$$C_1A_1 + (C_2A_2 + B(1 - A_2))(1 - A_1)$$

where C_i is the color of the line i, A_i is the opacity of the line i and B is the color of the background. When writing to the depth buffer is disabled, the result of blending in wrong order will be:

$$C_2A_2 + (C_1A_1 + B(1 - A_1))(1 - A_2),$$

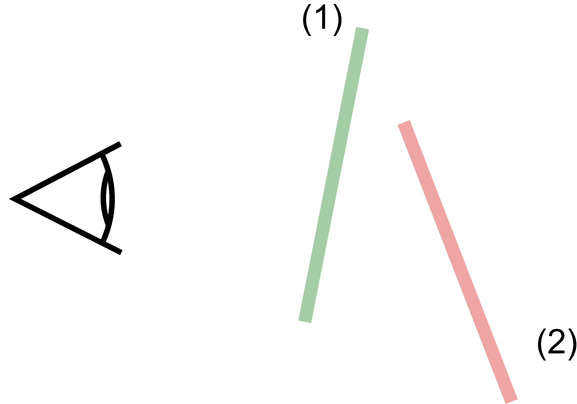


Figure 1.13: Line segments rendered in wrong order: first (1), then (2).

while with enabled depth buffer update, only the second line will be drawn, resulting in:

$$C_1A_1 + B(1 - A_1)$$

When rendering typical hair it is fairly safe to assume that the color of two hairs that are very close together will be similar. The background color can be fairly different, however. Let us say that $C_1 = (1.0, 0.8, 0.6)$, $A_1 = 0.5$, $C_2 = (0.8, 0.6, 0.4)$, $A_2 = 0.5$, $B = (0.2, 0.2, 0.8)$. Correctly blended color will be: $(0.75, 0.6, 0.6)$. Without depth buffer update: $(0.7, 0.55, 0.55)$ and with buffer update $(0.6, 0.5, 0.7)$.

When the color of overlapping hair fibers is similar, the error is plausibly small (if hairs have the same opacity, the error equals $(C_1 - C_2)A^2$). Significant errors may occur with strong self-shadowing as fully illuminated hairs may get covered by those that are shadowed, if rendered in wrong order. However, the combination of light colored hair and strong self-shadowing, which is the worst combination for this method, is not very realistic so in most cases the approximation behaves well.

Hair segments do not need to be re-sorted in every frame. It is possible to separate the visibility ordering from the actual drawing. For example, during interactive modeling, the viewpoint does not change much from frame to frame. This coherence allows reusing the computed order for subsequent frames.

With a high number of hair segments, there could be a significant lag when the re-sorting is performed. This does not matter too much during modeling but can be troublesome, for example, in games. The renderer presented in this thesis is supposed to be used during modeling so this issue is not considered to be crucial.

1.3.4 Order independent transparency

The Order Independent Transparency (OIT) denotes any technique that allows rendering translucent surfaces without the need of the fragment sorting. An example of such technique is the aforementioned Depth-peeling, see Section 1.3.2. Those techniques are, however, usually either too slow or lack quality.

The Occupancy maps method (described in section Section 1.5.5), which seems fast enough and provides sufficient quality, requires that all fragments have the same opacity. It computes color addition of every fragment in the fragment shader. This allows alpha blending to be set as additive, which is commutative operation and can be executed in any order, thus does not require fragment sorting.

Stochastic transparency presented by Enderton et al. (2010) [6] is another example of OIT technique. It is based on the so called Screen-door transparency by Mulder et al. (1998) [7], which replaces semi-transparent surfaces with a set of pixels that are either fully on or off. Enderton et al. shows an efficient way of sampling those pixels and adds an alpha correction and accumulation pass, all with massive use of modern hardware. However, images that are rendered at interactive frame rates contain some noise. The main advantage of Stochastic transparency is that it provides a unified approach to OIT, anti-aliasing, and deep shadow maps.

1.3.5 Discussion

For my renderer, I used the sorting approximation method presented in Section 1.3.3. The main reason for it is that I need the sorted geometry for deciding line width (see Section 1.2.3). Sintorn and Assarsson (2008) [5] proposed a GPU algorithm for fast approximate sorting based on quick sort, which could be a better alternative to presented sorting method though. Implementation of the GPU based sorting in my renderer remains a subject of future work.

Occupancy maps method requires hair fibers to have the same opacity, which is too restrictive. Stochastic transparency technique looks promising but it is not certain if an implementation of the stochastic transparency would give good results for complex hair at interactive frame rates.

1.4 Light-scattering model of a hair fiber

Extensive research has been done in order to capture the light scattering of human hair. This section provides an overview of some significant models that were developed.

1.4.1 Kajiya-Kay model

The lighting model known as Kajiya-Kay model was presented by Kajiya and Kay(1989) [8] and is formed by two components, the diffuse and specular. The diffuse component is computed by adapting the Lambert shading model to an infinitely thin cylinder. The specular component uses Phong light reflection model that has been modified for cylinder surfaces.

The diffuse component

The diffuse component is an integration of a Lambert surface model along the illuminated half of the cylinder. The result of the integration is a simple function

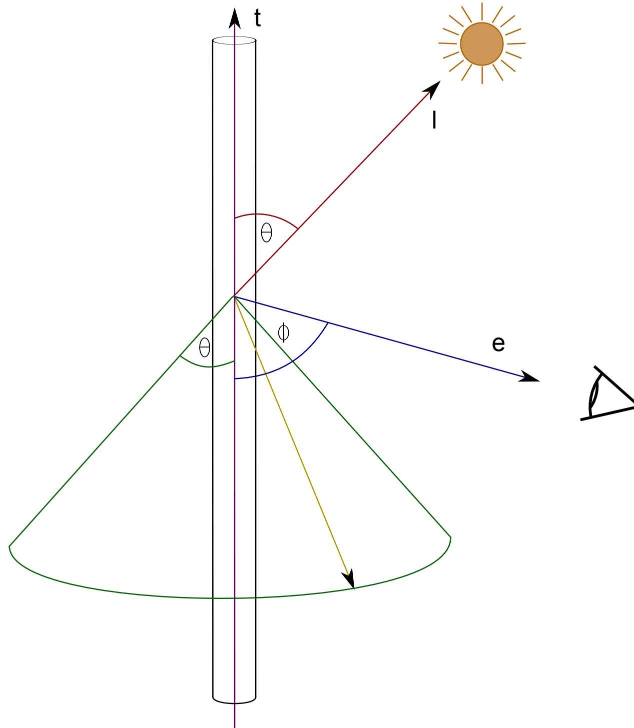


Figure 1.14: Notation for Kajiya-Kay scattering geometry.

of angle θ (between the hair tangent t and the light direction l) given by:

$$\Psi_{diffuse} = K_d \cdot \sin\theta$$

where K_d is the diffuse coefficient. This computation does not take into account self-shadowing of the hair fiber but the human hair is quite translucent so even the shadowed area of the fiber transmits diffusely an amount of light that is comparable to the amount reflected diffusely on the non-shadowed side.

The specular component

The specular component computation is based on the Phong specular model. It exploits the fact that the reflection of a parallel beam from the cylinder surface forms a cone centered on the hair axis, see Figure 1.14. The actual highlight intensity is maximal if the eye vector is contained in the reflected cone and falls off with Phong dependence:

$$\Psi_{specular} = K_s \cdot \cos^n(\theta - \phi)$$

where K_s is the specular coefficient.

The final fragment color is given by

$$FragmentColor = C_a + \Psi_{diffuse} + \Psi_{specular}$$

where C_a is the ambient color.

1.4.2 Marschner model

Marschner et al. (2003) [9] presented a physically-based scattering model for hair based on measurements of real human hair. The fiber is modeled as a translucent elliptical cylinder with pigmented interior.

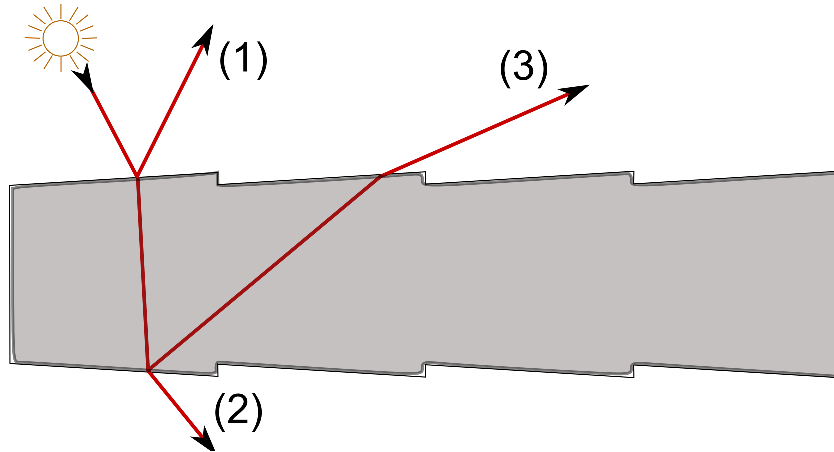


Figure 1.15: A schematic of Marschner model for a hair fiber. Shows three different reflection modes.

It brings two improvements to Kajiya-Kay's model. It introduced azimuthal dependence into the fiber scattering model based on the ray optics of a cylinder with tilted surface scales, and it shows three possible modes of reflection. The first mode is a reflection on the hair surface and is also considered by the Kajiya-Kay model. The second mode is a transmission through hair fiber, which produces a bright component that is focused on the opposite side of hair fiber. In the third mode, the light refracts to the fiber volume, reflects off the inside of the hair surface and refracts back with shifted angle, see Figure 1.15. This makes two visually distinguishable highlights.

Nguyen and Donelly (2005) [10] showed a real-time approximation of the Marschner model by using a precomputed look-up texture.

1.4.3 Multi-scattering models

Multiple scattering between neighboring hairs can be observed especially in light colored hair. The Kajiya-Kay model is accurate for black hair and Marschner model is accurate for dark hair but both fail to capture multiple fiber scattering effects. Those effects are accounted for in model presented by Moon and Marschner (2006) [11]. However, this method use photon mapping and cannot be made interactive.

The dual-scattering model introduced by Zinke et al. (2006) [12] divides the scattering function into global multiple scattering and local multiple scattering. The global multiple scattering approximates self-shadowing properties of hair considering the forward-scattered light through hair fibers through the light path, while the local multiple scattering accounts for the scattering events within the neighborhood of a hair fiber. The dual-scattering method can achieve interactive times.

1.4.4 Discussion

The light-scattering model used in my renderer is the Kajiya-Kay model. It is mainly because this model can be easily and effectively implemented on the graphics hardware. It is probably the most commonly used model for real-time rendering and the quality of this model is sufficient for the interactive preview during hair modeling.

1.5 Self-shadowing in hair

Even a very thin hair fiber casts a shadow, just like any object. A shadow from a single fiber may be insignificant, but shadows in a dense volume of hair produce a very distinctive effect. Self-shadowing creates important visual patterns that distinguish one hairstyle from another, see Figure 1.16.

Hair exhibits complex light propagation, as each hair fiber transmits and scatters the incoming light. It makes the shadow computation very difficult.

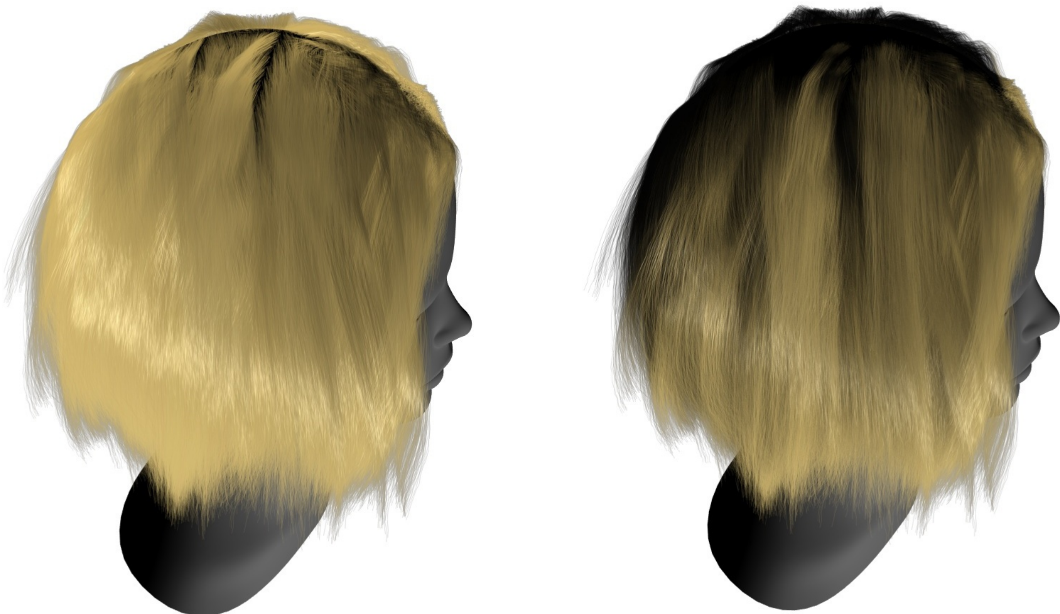


Figure 1.16: The difference between shadowed and non-shadowed hair. Note how the shadows reveal the details of the hair shape.

1.5.1 Depth-based shadow mapping

Most commonly used techniques for hair self-shadowing are based on Depth-based shadow maps (DBSM) presented by Williams (1978) [13]. In the first pass of shadow mapping, shadow casting objects are rendered from the light's point of view, and the lowest depth values for each fragment are stored in a depth map. During the final render pass, each fragment to be shadowed is projected onto the light's camera and the fragment's depth is compared with the depth in the shadow map. If the fragment's depth is higher then the corresponding depth value from the depth map, it means that the fragment is not visible from light, thus shadowed.

This approach has two main issues - the thin hair fibers and the translucency. The problem with the thin geometry is very similar to one discussed in Section 1.2. This is not surprising if we note that the shadow computation is just one instance of the more general visibility computation problem. As showed in Section 1.2.3, this problem can be overcame with the translucency. That brings us to the second issue with shadow maps. The binary decision nature of the shadow mapping does not allow translucent object to cast attenuated shadows. The fragment can be either fully illuminated or fully shadowed.

Also, light-colored hair fibers do not fully block the incoming light, thus they are translucent and it is impossible to capture their shadows by shadow mapping.

Depth-based shadow mapping can be effectively implemented on graphics hardware, but its inability to compute shadows of translucent object makes it unsuited for volumetric objects such as hair.

1.5.2 Deep Shadow Maps

Lokovic and Veach (2000) [14] presented a high quality self-shadowing method for off-line rendering. For each texel of a deep shadow map a transmittance function in the light direction is computed. Sampling of the transmittance function is done by finding intersections with semi-transparent objects by shooting a ray from the light position. Those samples are then compressed into a list of pairs of transmittance and depth value.

The computation of a shadow value for a hair fragment is found similar to shadow maps, but the depth of the fragment is used to evaluate the transmittance function at the corresponding texel of the deep shadow map.

This technique produces very accurate results, but due to the underlying data structure (linked list), an efficient hardware implementation is difficult to achieve.

1.5.3 Opacity Shadow Maps

Opacity Shadow Maps (OSM) by Kim and Neumann (2001) [15] is essentially a simpler version of deep shadow maps that is designed for interactive hair rendering. It exploits the fact, that the transmittance functions vary smoothly in the hair volume. This allows an approximation by linear interpolation of fixed number of samples, see Figure 1.17.

OSM use a similar principle as the one presented in Section 1.3.3. The hair geometry is divided by the opacity maps, planes that are perpendicular to the light direction and are identified by their distances from the light source. Each texel of a map contains the line integral of densities along the path from the light to the texel.

An opacity map can be efficiently rendered on graphics hardware by rendering the hair geometry from light's point of view. A separate rendering pass is performed for each opacity map with maximum depth for rendering set to the corresponding depth of the opacity map. The computation of hair density is performed by additive blending on graphics hardware.

In the final render pass, the actual light transmittance (τ) for a fragment (p) is computed from linear interpolation of the corresponding texel values from the

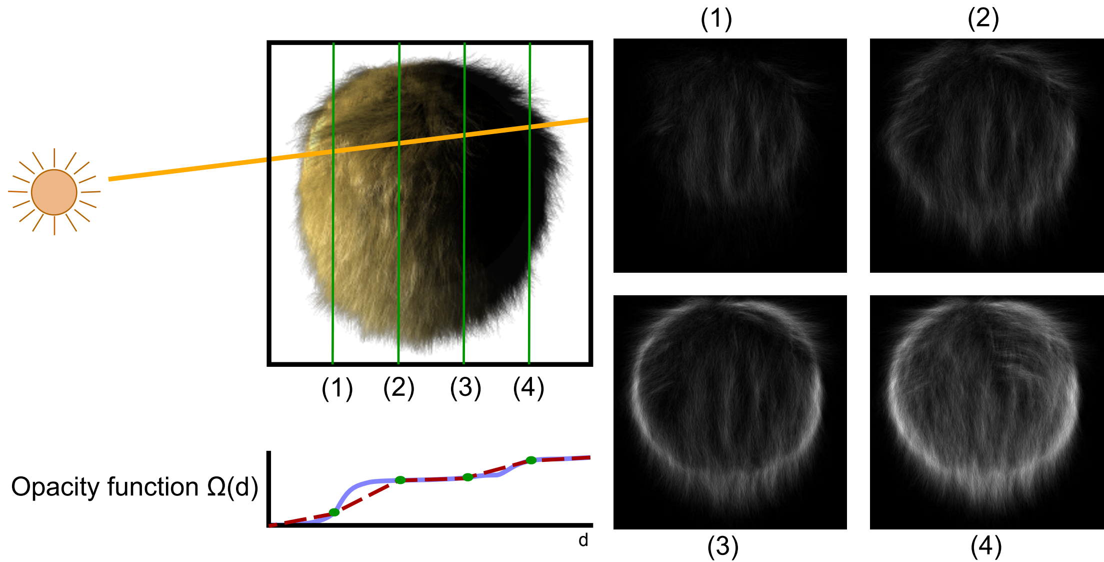


Figure 1.17: Hair volume divided by four opacity maps. Generated opacity maps are shown on the right. The opacity function $\Omega(d)$ shown as blue curve is approximated by linear interpolation of values from these maps (red line).

adjacent opacity maps (Ω):

$$\tau(p) = \exp(-\Omega(p_z))$$

where p_z is the depth of fragment p in light direction.

Depending on the number of opacity maps used, the quality of self-shadowing can be much lower compared to deep shadow maps. Due to the interpolation of the opacities between the maps, the result may contain visible artifacts on the hair (Figure 1.18). In order to minimize those artifact, a large number of opacity maps has to be rendered.

Figure 1.18: Visible artifact when 8 opacity shadow maps are used.

Nguyen and Donelly (2005) [10] showed how to render 16 opacity maps in a single pass by using multiple render targets.

Sintorn and Assarsson (2008) [5] presented a way to increase the speed of rendering a large number of opacity maps by sorting the geometry by depth from light source (same principle as in Section 1.3.3). This way, the opacity maps can be rendered in a single pass.

1.5.4 Deep Opacity Maps

The Deep Opacity Maps (DOM) algorithm by Yuksel and Keyser (2008) [16] is a further improvement of shadow maps for hair self-shadowing. It is a combination of a DBSM (Depth-based shadow mapping) and OSM (Opacity Shadow Maps) techniques. It requires much fewer map layers than OSM to produce an image of hair without visible artifacts. Therefore, it is faster and uses less memory.

DOM uses two passes prior to the final hair rendering. First, the depth map is rendered from the light's position. The hair volume is then divided into slices just like it is done in OSM algorithm. But this time, the depth of an opacity map is not constant for all texels. Every texel in the map can represent an opacity at a different depth. The depth of the texel in i-th map is computed as $z_0 + D_i$, where z_0 is value from the same texel in the depth map, and D_i is the depth of i-th opacity map. $D_0 = 0$; $D_i - 1 < D_i$. The size of the spacing between the opacity maps $(D_k - D_{k-1})$ does not have to be constant. Since z_0 varies by texel, the opacity maps take shape of the hair geometry see Figure 1.19.

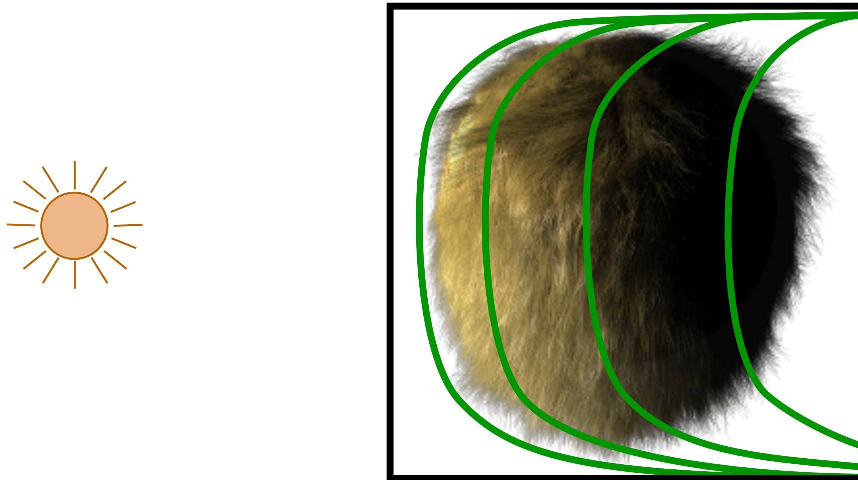


Figure 1.19: Deep opacity map layers conforming to the shape of hair.

The second step renders the opacity maps using the depth map computed in the first step. All opacity maps are rendered in a single pass. Which fragment falls into which map is decided in the fragment shader. The z_0 value for the fragment is looked up from the depth map. The fragment's opacity contribution is assigned to opacity map O_i if: $z - z_0 < D_i$. Total opacity of an opacity map is the sum of contributions from assigned fragments, efficiently computed by additive blending on graphics hardware.

In the final rendering, the fragment's light occlusion is computed in a similar way as in OSM. The difference lies in the need to subtract the depth map value z_0 from the projected fragments depth z in order to look up opacities from corresponding opacity maps.

Each opacity map is represented as a channel in a texture. One channel is reserved by the depth map. By enabling multiple render targets the number of opacity maps that can be rendered is $4n - 1$, where n is the number of draw buffers.

With a small number of layers, it is more difficult to ensure that all hair fragments are assigned to an opacity map. Points beyond the last layer do not

correspond to any opacity map. We can ignore them, thus those points will not cast shadows. Or we can add them to last opacity map. Then, they will shadow themselves. Ideal solution would be to add another layer that will contain these points but it would lead to unnecessary extra layers. The points beyond the last layer are not expected to be strongly illuminated so the second options gives reasonable results in most cases.

The most sensitive part of hair for self-shadowing is the fully illuminated hair surface. The main advantage of DOM over OSM is that the illuminated hair surface is rendered very accurately. For best results, the distance between opacity maps should be linearly increasing from the first to last. It ensures that the shadows in more illuminated areas of hair are computed more precisely.

The drawback of DOM over OSM is in the hardware filtering. Graphics hardware can linearly interpolate texture values. OSM maps can exploit this feature resulting in less aliased shadow edges. This is not possible with DOM maps, because an interpolated value between adjacent texels does not equal interpolated value between adjacent fragments. Hardware texture filtering must be turned off for Deep opacity shadow maps. Otherwise, visible artifacts on shadow edges will appear. The filtering is possible in fragment shader but requires more texture look ups (at least 2 look ups for every sample) and thus is much slower.

As showed in Figure 1.20, DOM produces much better self-shadowing than OSM, when similar number of layers is used.

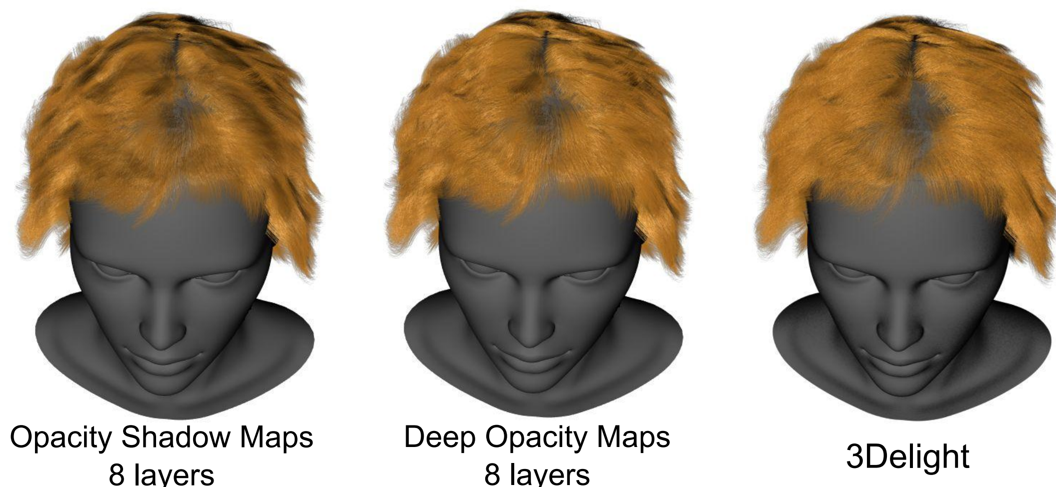


Figure 1.20: The difference between Opacity Shadow Maps and Deep Opacity Maps. With 8 layers, the Opacity Shadow Maps contains visible artifacts.

1.5.5 Occupancy Maps

The Occupancy Maps, presented by Sintorn and Assarsson (2008) [17], show a modification to DOM that improves sampling of the light transmittance function. It uses several types of maps.

There are two depth maps; first captures the nearest fragments and second captures the farthest fragments. It allows finding the exact length of the hair volume for every texel in the light direction.

Then, there are occupancy maps. The hair volume for every texel is divided into N parts corresponding to N maps and each part is further divided into S sub-parts, where S is the number of bits in occupancy map texture. If there is a hair fiber fragment that falls into the i-th part and j-th sub-part, the j-th bit of the i-th occupancy map is marked as 1. Each bit signify an opacity increase of $\frac{F}{S}$ where F is the total number of fragments that falls into this occupancy map's texel. The F values are stored in so-called slab-maps, see Figure 1.21.

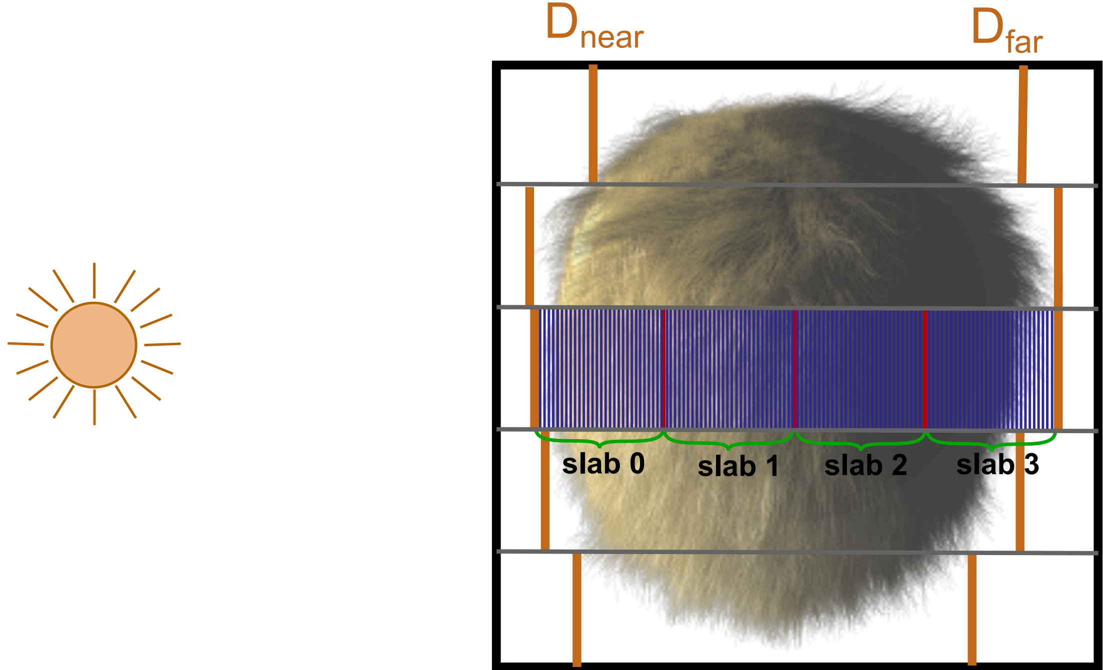


Figure 1.21: Visualization of Occupancy maps.

Generating the maps

First, there are two render passes for near and far depth maps. In next pass, the occupancy maps are created. The ideal format for occupancy maps is a four-channel int texture with 128 bits total. The opacity values are computed in the fragment shader, where the fragment's relative depth is obtained as:

$$d = \frac{frag.z - D_{near}}{D_{far} - D_{near}}$$

The fragment falls to i-th occupancy map and j-th bit if:

$$i = \lfloor d * N \rfloor$$

$$j = \lfloor d \cdot N \cdot S \rfloor - i \cdot S$$

The final occupancy maps are given by using the bitwise-or blending operation on the framebuffer.

Slab maps are generated in a similar way as deep opacity maps.

Finding the depth order of the fragment

During the final rendering, the number of fragments that have lower depth than fragment x can be computed as:

$$x_o = (\sum_{i=0}^{x_{map}} F_i) + (\sum_{i=0}^{x_{bit}} B_{i,x_{map}}) \cdot \frac{F_{x_{map}}}{\sum_{i=0}^{S} B_{i,x_{map}}}$$

$$x_{map} = \lfloor x_d * N \rfloor$$

$$x_{bit} = \lfloor x_d \cdot N \cdot S \rfloor - i \cdot S$$

where F_i is the number of hair fragments in i-th slab map and $B_{i,j}$ is the value of i-th bit in j-th occupancy map.

The computed order can be used to compute the opacity for self-shadowing as $opacity = x_o * w$, where w is the opacity of one hair fiber. It requires that all fragments have the same opacity value.

The order can be also used for order independent alpha blending, see Section 1.3.4. With additive blending enabled, the color of fragment is computed as:

$$C_{out} = (1 - \alpha)^{x_o} \cdot \alpha \cdot C_{in}$$

where C_{in} is the fragment's color and α is fragment's opacity. This computation assumes that all hair fibers have the same opacity.

Comparison with DOM

The Occupancy Maps (OM) can sample the light transmittance function more precisely than Deep Opacity Maps. Where DOM uses linear interpolation between opacity maps, the OM uses occupancy maps between slab maps. The OM can capture non-smooth attenuation of the transmittance function much better than DOM (see Figure 1.22), but requires that all hair fibers have the same opacity.

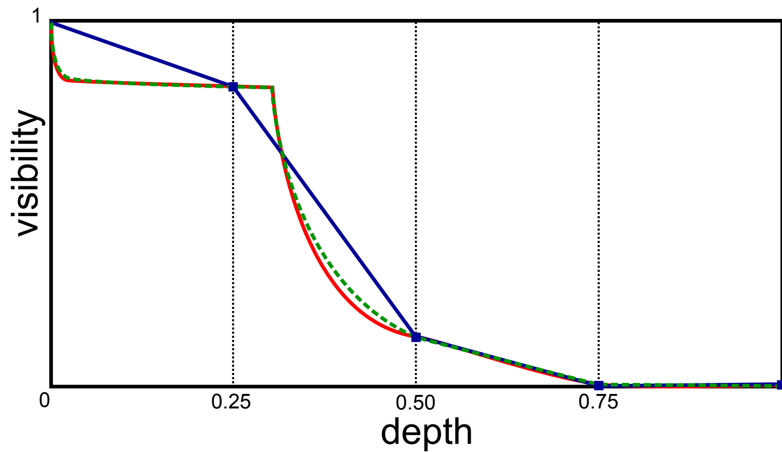


Figure 1.22: Example of visibility function. Red is true visibility, blue is Deep Opacity Maps approximation and green is Occupancy Maps approximation.

1.5.6 Discussion

The self-shadowing technique used in my renderer is the Deep Opacity Maps (discussed in Section 1.5.4). It provides sufficient quality and performance, and does not require all hairs to have the same opacity (as OM), which is important because the hair tip and hair root can have different opacity. Furthermore, the opacity of a hair fiber also depends on the distance from the light camera plane as described in Section 1.2.3.

2. Implementation

This section will briefly discuss the architecture of Maya and Stubble, then it will show the architecture of the presented renderer and finally, the implementation details about individual aspects of rendering (as described in Chapter 1) will be discussed.

2.1 Autodesk Maya

Autodesk Maya is a popular 3D animation software. Its feature set includes, among others, tools for animation, modeling, simulation and rendering. It is widely used in production of animated movies, television shows, video games and other CGI content.

Maya's architecture is designed to be very flexible in terms of adding new functionality. The architecture is based on a database called the Dependency graph. The functionality for modeling, animating, rendering and others, is hidden in nodes of the Dependency graph. Nodes can have input and output attributes. An output attribute of one node can be freely connected to an input attribute of another node, as long as the attributes have similar data type. Each node can modify input data or create new data and send it to next node. Some nodes can perform special operations such as rendering.

New functionality can be easily added by implementing it into a new node and connecting the node to the Dependency graph. This enables great flexibility, but it also makes orienting in the Maya environment more difficult.

By combining a large set of nodes, complex scenes can be represented. There is a number of basic types of nodes in Maya. Here are some of them for illustration:

- **MPxDeformerNode** - A deformer is node which takes some geometry data on input and places modified data into the output geometry attribute.
- **MPxEmitterNode** - This nodes represents particle emitters.
- **MPxSurfaceShape** - This node defines shapes in Maya and produces geometry data.
- **MPxHwShaderNode** - A shader node that controls rendering to the screen.

Maya provides two APIs for implementing new functionality. Python API and C++ API. Both APIs provide similar means for developing new plug-ins, but the C++ API runs faster and thus is more suitable when performance is required.

2.1.1 Viewport 2.0

Viewport 2.0 is an optional viewport that was introduced in Maya 2011 to provide high-quality interactive preview of the scene. It is designed to improve performance on large scenes and offer high-quality per-pixel lighting and effects. The

geometry in Viewport 2.0 is cached when possible, which makes rendering, especially in large and complex scenes, much faster compared to rendering in the default Maya viewport.

In Maya 2012, the Viewport 2.0 was significantly improved. It introduced new effects such as anti-aliasing, motion blur, gamma correction, depth of field and screen-space ambient occlusion.

I decided to use the Viewport 2.0 as it seemed to be better alternative to the default viewport. The presented features were suitable for my renderer. However, as it turned out, it brought some problems. See Section 2.3.2.

2.2 Stubble

Stubble is a plug-in for Maya for modeling hair styles. The hair styling is performed by modifying hair guides. Final hair fibers are then interpolated from those guides. The main goal of Stubble is to allow modeling of different hair styles in Maya software and to export the generated hair geometry to an off-line renderer.

Stubble was developed as a software project at the Faculty of Mathematics and Physics, Charles University in Prague. It was successfully defended in January 2012. I was given the source code of the this project in order to implement realistic and interactive hair renderer that will use the geometry generated by Stubble.

The renderer that was initially included in the Stubble did not provide preview of sufficient quality, which would allow 3D artist easy recognition of the hair style, see Section 2.2.3. During modeling, the hair styling artist had to preview the hair shape by using an off-line renderer, which is slow and ineffective for such task.

2.2.1 Implementation of Stubble in Maya

The hair generating functionality of Stubble is implemented in class HairShape, which overrides Maya's MPxSurfaceShape class. It means that the HairShape is a node in Dependency graph that defines a shape. It has an input attribute, which is connected to another shape node with output mesh attribute. The hair root positions are then generated on surface of the given mesh. Another input attribute of a HairShape is connected to the Time node for animation purposes. Other attributes are fetched from the Maya UI.

From Maya's point of view, the HairShape is a black box that accepts some attributes, knows how to draw itself and how to respond to selection requests. The internal data and process of hair generation is not known to Maya.

2.2.2 Hair generation in Stubble

The hair geometry generation is done in the HairGenerator class. For communication with a renderer, this class needs two other classes, which implement interfaces PositionGenerator and OutputGenerator respectively. Since every renderer may require different data types of geometry and other properties (such as color), each renderer has different implementation of those interfaces. It

uses class HairProperties, which provides access to various properties of the generated hair (for example scaling or randomization).

The PositionGenerator interface serves for generation of the root hair positions on a given mesh. It samples UV coordinates of the mesh and produces UVPoints. Those are supplied to Mesh object, which will calculate a 3D position in the world space.

The OutputGenerator interface is used as a proxy between HairGenerator and the renderer. Individual generated hairs are sent by HairGenerator to OutputGenerator, where the geometry of whole hair is stored. Which data are stored depends on the renderer's requirements and the implementation. OuputGenerator then provides methods that will enable the renderer to access stored data. Those are:

- **Positions**: Each hair fiber is defined by a number of hair points, which the hair curve goes through. Those points are represented by position in 3D world space. Every hair has the same number of hair points. That number can be changed in user interface.
- **Colors**: Each hair point has a color assigned in RGB. The color is interpolated between consecutive hair points.
- **Opacities**: Similar to color, opacities are assigned to hair points and interpolated.
- **Widths**: Widths are also defined for every hair point.
- **Normals**: Normal can be any vector perpendicular to the curve tangent.
- **Hair index**: Unique identifier of hair fiber.
- **Strand index**: Several hairs can be generated around a main hair fiber. Those hairs form together one hair strand. This property is unique identification of such strand.
- **Hair UV coordinates**: Texture UV coordinates corresponding to the hair root position on the mesh.
- **Strand UV coordinates**: UV coordinates of the position on where the main fiber of a hair strand starts to grow.

Each of these properties must have data types defined by the class that implements the OutputGenerator. For example position is an output with type PositionType. The OutputGenerator has a template argument which must be the class defining these types.

The generating of a large number of hairs can be quite slow. In order to speed it up, the hair is divided into several hair groups, which are generated in parallel via OpenMP. Each of this group has its own set of data for hair generation stored in class ThreadData. In this class exists one instance of HairGenerator, OutputGenerator and PositionGenerator. The number of groups is defined by number of system threads (queried by OpenMP's method *omp_get_max_thread*).

2.2.3 Default interactive renderer in Stubble

The Stubble has an interactive renderer that can draw hair geometry in Maya's default viewport. It uses a non-rotating ribbon representation. As stated in Section 1.2, it has some crucial issues. Firstly, the hair width does change with rotation of camera. Secondly, it suffers from severe aliasing, especially with thin hairs.

The renderer does not solve the problems with transparency sorting, light-scattering model and self-shadowing. Therefore, the resulting images from this renderer do not provide a preview of sufficient quality, which would help with accurate hair modeling, as shown in Figure 2.1.

Figure 2.1: Default Stubble renderer fails to provide a plausible preview of hair.

The renderer is implemented in MayaOutputGenerator, which implements interface of OutputGenerator. During the hair generation process the hair ribbons are created, and when needed, drawn to the Maya default viewport. The hair is renderer in OpenGL as colored triangles without any light computation.

2.2.4 RenderMan and Stubble

RenderMan is an API developed by Pixar for high-fidelity rendering of 3D scenes. It is widely used as a rendering tool in the film industry. Hairs from Stubble can be rendered in 3Delight, which is an implementation of the RenderMan API. The huge advantage of 3Delight is that its version called 3Delight for Maya can be loaded to Maya as plug-in. Thus, it is easy to render Maya scenes with 3Delight through Maya user interface.

The hair geometry generated by Stubble is not, however, sent to 3Delight directly in the Maya environment. Instead, 3Delight, before rendering, loads a dynamic library, which generates hair geometry for it. That library is called StubbleHairGenerator.dll. It is basically an implementation of the HairGenerator, PositionGenerator and OuputGenerator mentioned above with the addition of few methods for communication with 3Delight. The required properties from Maya (such as color) are saved by the Stubble plug-in loaded in Maya to a file, and then read by the StubbleHairGenerator.dll loaded in 3Delight.

2.3 Architecture of the presented renderer

In this section, I will discuss the architecture of my renderer and its connection to the Maya Viewport 2.0 and Stubble.

2.3.1 Rendering in Viewport 2.0

Custom rendering is done in Viewport 2.0 by registering a draw override for an existing node. It can be either a shape node or a shader node. There are also some other overrides like geometry override or render override. Details are presented in Viewport 2.0 API reference paper[1].

The shader can be attached to any shape node in the scene, modifying the way how the shape would be drawn. If the shader override (MPxShaderOverride) is registered to a shape node, it will take control of the rendering in the Viewport 2.0. The shader is supposed to only add a shading effect to the shape and does not have a direct access to the shape's geometry data.

The shape override (MPxDrawOverride) can replace the rendering logic in a shape node. The MPxDrawOverride class can obtain a pointer to the shape node, which allows easy access to the geometry data and other properties.

My renderer is based on the shape override because it renders only one specific shape. The shader node override is useful for adding a specific effect to a generic shape.

In order to override a shape node, it is necessary to implement a class that derives from MPxDrawOverride and register that class with MDrawRegistry against a classification string for the shape node. The derived class must implement several virtual methods:

- **creator** - Static class that returns a new instance of the derived override class.
- **boundingBox** - Returns the bounding box of the rendered geometry.
- **prepareForDraw** - This method prepares data for rendering and saves it to a class derived from MUserData. That class is then sent to draw method. *prepareForDraw* is called for each screen refresh before rendering.
- **draw** - This static class is called by Maya whenever an overridden shape needs to be redrawn. It cannot access Maya's Dependency graph. The only non-static data it gets is through its parameters. First is of type MDrawContext and provides informations about the rendering context. This means viewport properties, light properties, world and projection matrices and GPU state informations. The second parameter is a constant pointer to the data class that was prepared in *prepareForDraw*.

2.3.2 Problems with Viewport 2.0

The Viewport 2.0 is relatively new. It was introduced in Maya 2011 and was heavily reworked in Maya 2012. As such, there is not a lot of documentation available.

[1] `http://images.autodesk.com/adsk/files/viewport_2_0_api_gold.pdf`

Except not very detailed Maya API reference, there is a short introduction paper and a few simple examples of basic usage [2].

There is also some strange functionality in Viewport 2.0. For example, if the draw override is registered against a shape node, it stops receiving input attributes as it should. This issue causes the hair geometry to stay in a position even if the mesh to which the hair is attached, moves. I did not find a way how to solve this issue.

Another problem arises from the fact that there is no way (or at least I did not find one) of influencing the render order in which is the Viewport 2.0 draw override executed. This may cause problems, when there are more translucent objects in the scene.

All these things make the development in Viewport 2.0 rather difficult. It will probably take some time before the Viewport 2.0 will be normally usable.

2.3.3 Shader programs

Shaders are small specialized programs that are executed on the graphics processing unit (GPU). I use two types of shader programs in my renderer: vertex shader and fragment shader. The shaders are written in OpenGL Shading Language (GLSL). The main reason for that is that this language is supported by most graphic cards and works well in OpenGL.

Vertex shader is executed in the GPU for every vertex before the rasterization. It is responsible for transforming the vertex from the object space into the clip space. It can also modify the color, normal, texture coordinates, and other parameters that are assigned to the vertex.

Fragment shader is executed after the rasterization and its main responsibility is to compute a color of the fragment. It can also modify the depth value of the fragment. The fragment shader has access to texture units.

Shaders for opacity map generation

Shaders for opacity map generation are named DeepOpacityMaps. The vertex shader DeepOpacityMaps.vert, besides transforming vertex into clip space, also transforms vertex to the depth map texture space. This transformed position is later used in fragment shader for accessing the depth map. The alpha value of the vertex is modified according to the line width correction, see Section 2.4. Fragment shader (DeepOpacityMaps.frag) outputs opacity values to opacity maps. For more details please refer to Section 2.7.2.

Shaders for final rendering

In the final rendering, the KajiyaKay shaders are used. There are four pairs of vertex and fragment shader programs and each of those pairs is used when different number of lights illuminates the hair (up to four). It is because of the architecture of the GPU. Shader programs are massively parallelized and work very efficiently if there is minimal branching in the program. If varying number of lights would be used in a shader program, it would include branching. Such

[2] `http://download.autodesk.com/global/docs/mayasdk2012/en_us/index.html`

program would run much slower with a single light than specialized one-light shader. Details about multiple light implementation are discussed in Section 2.5.3

The KajiyaKay vertex shader is similar to the DeepOpacityMaps vertex shader. It computes vertex transformation to the clip space and texture space and computes line width correction. Fragment shader computes Kajiya-Kay light model for the fragment and applies shadow. For more details please refer to Section 2.7.

2.3.4 HairDrawOverride

The HairDrawOverride class is the heart of the renderer. It derives from the MPxDrawOverride and is registered at the plug-in start-up against the Stubble hair shape. HairDrawOverride also overrides some virtual methods of MPxDrawOverride, which are then called directly from Maya. The main responsibilities of this class are:

Initialization

Whenever a Stubble HairShape node is created, the HairDrawOverride class is instanced and initialized. During that process, the context for shaders is created.

Next step is to compile the shader programs. Those programs are located in Maya Directory/bin/glsl alongside other Maya shader programs. Lastly, the parameters for those shader programs are bound.

The initialized properties are static and shared by all instances of HairDrawOverride. It means that the initialization process is done only once for all instances. The HairDrawOverride class keeps the number of instances and when the last instance is destroyed, the shader context is destroyed too.

Pre-draw preparation

When a screen refresh is required and the HairShape node needs to be redrawn, the *prepareForDraw* method is called by Maya before the rendering begins. In this class, the data needed for drawing the HairShape is prepared. It is done by filling the DrawData class. This class serves as a storage for render-related data, is derived from MUserData and handled by Maya. The DrawData class is instantiated in the first call of *prepareForDraw*. The destruction of this class is done after the HairShape node is destroyed and is handled by Maya. The DrawData class is sent as constant pointer to a draw function, which will be described later.

During the pre-draw preparation there are several tasks that need to be done:

- Get additional hair shader properties from UI.
- Get hair geometry data from the Stubble hair generator. See Section 2.3.5.
- Sort the hair geometry by depth from camera plane. See Section 2.5.1.
- Generate OpenGL buffer objects from sorted hair geometry for rendering. See Section 2.3.5.

- Update lights from Maya and shadow maps. See Section 2.7.

Final hair drawing

The final drawing takes place in draw method. As stated before, it receives a constant pointer to an instance of DrawData as a parameter. It also receives a constant reference to the Viewport 2.0 draw context, which is used to get modelview and projection matrices, and viewport properties such as resolution, and near and far depth boundaries.

The draw method sets up transformation matrices, OpenGL attributes, shader parameters, computes line width, and pixel width (see Section 1.2.3), binds OpenGL buffer objects and shadow map textures and draws the geometry. The geometry is drawn by the bins, in which the sorted geometry is stored, from back to front from camera plane, see Section 2.5.1.

2.3.5 HairModel

Figure 2.2: Class diagram of HairModel.

This class stores and manages all data regarding the hair geometry. The raw geometry data is obtained during pre-draw stage in *prepareForDraw* method. In this method, a reference to the HairModel object is sent to Stubble's HairShape class. Here is decided whether the hair representation is set to lines or

ribbons, see Section 1.2. From the HairShape class, the HairModel reference is sent to the InterpolatedHair class, which manages hair generation in Maya. In the InterpolatedHair, the HairModel object gets a pointer to the raw data, which is obtained from MayaOutputGenerator, see Section 2.2.2. Figure 2.3 illustrates the data flow of this process.

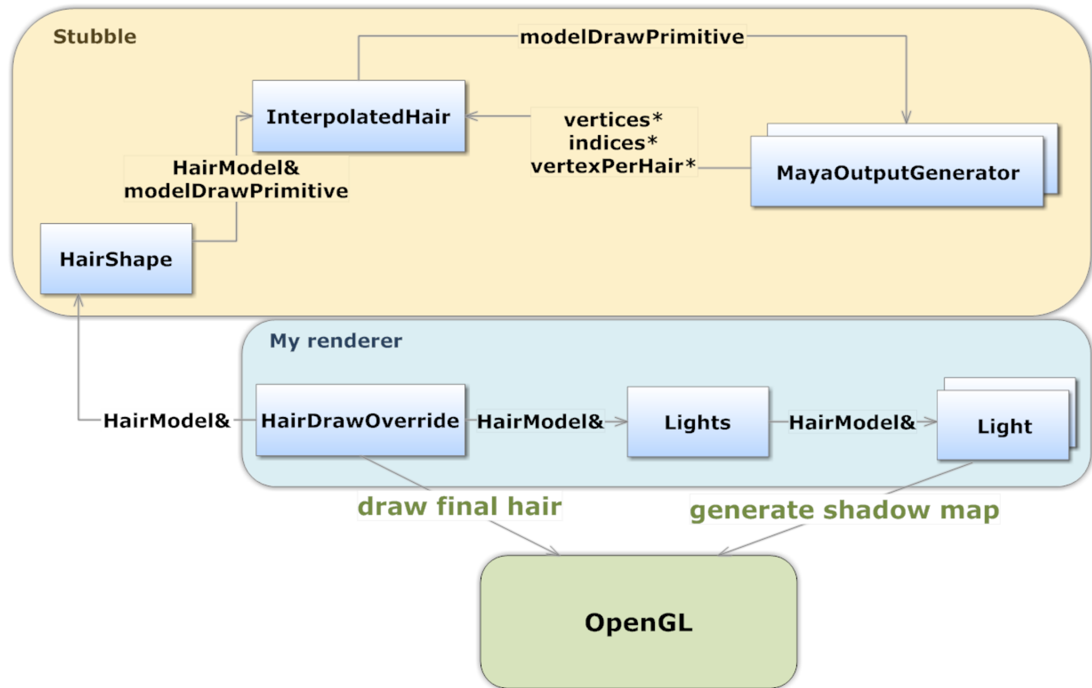


Figure 2.3: Data flow diagram for obtaining raw geometry data from Stubble.

The raw data is divided into one or more parts. The number of parts depends on the number of threads that generated hairs, see Section 2.2.2. Each part is, in fact, independent hair geometry and contains arrays of hair vertices and indices.

Vertex and index arrays format

Vertices are stored in an array of floats. Each vertex has 4 floats for color with alpha value in RGBA format, 3 floats for hair tangent, and 3 floats for vertex position. That means that the size of the vertex array is $10 * number\ of\ vertices$. Number of vertices equals number of hair points for line strip representation, and $2 * number\ of\ hair\ points$ for ribbon representation. Number of hair points is a $number\ of\ hairs * (number\ of\ hair\ segments + 1)$ (can be changed in UI). Vertices are stored sequentially with starting vertex corresponding to hair root point of the first hair, and continuing to hair tip, followed by next hair.

Index array defines which vertices are connected together in order to create a line or triangle. Individual indices are represented as unsigned integers. Each integer defines a position of a vertex in the vertex array. Since the hair vertices are stored sequentially, the index array for a line strip representation and 4 hair segments looks like 0112233455667 etc. Each pair of values represents index of two vertices that are connected to a line. The ribbon representation uses triangles, and how the indices are sequenced is shown in Figure 2.4.

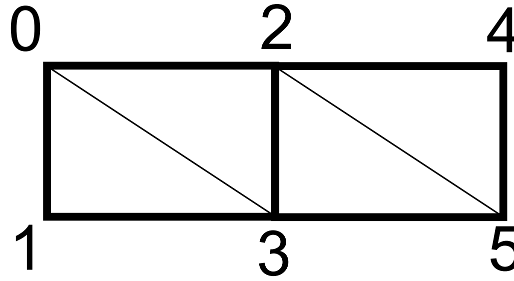


Figure 2.4: Index array for ribbons is stored as 320013542235 etc.

Pointers to vertex and index arrays are copied to a structure called THREADHAIRDATA. One instance of this structure contains pointers to arrays from one part of hair geometry that was generated by one thread. Besides those arrays, a number of hairs, a number of elements, a number of vertices, and an array of numbers of vertices in individual hairs is stored. An array of THREADHAIRDATA structures that together contains whole geometry, is stored inside the HAIRMODEL.

Vertex and index buffer object division

Vertex buffer object (VBO) and Index buffer object (IBO) is an OpenGL buffer that allows vertex (resp. index) array data to be stored in high-performance graphics memory on the server side. When the buffer objects are generated, it is relatively easy to render the geometry contained in them. First, the buffers must be bound. Then the format of vertex array must be specified. Finally, an OpenGL method for drawing with parameters that tells the GPU which draw primitives to use and number of elements to draw, must be called. GPU then reads the geometry data from the internal memory, which is very efficient.

The geometry data must be, however, divided into a number of parts in order to render hair properly. As stated above, the vertex and index arrays are divided into several parts depending on the number of threads that generated the hair geometry. Let's call that number *threadCount*. But that is not all. As mentioned in Section 2.5.1, the hair geometry is divided into a number of bins based on the distance from camera plane.

The vertex data can stay in the arrays as it was divided during hair generation. The number of vertex arrays is therefore equal to *threadCount*.

The index data must be divided into more arrays. Each part that was generated by a single thread is further divided into a number of parts that is equal to the number of sort bins. The final number of index arrays is $threadCount * number\ of\ sort\ bins$, which can be quite high (1024 on a quad-core processor).

The algorithm for rendering can be summarized as follows:

2.3.6 Lights

The Lights class is responsible for storing the data of lights that are placed in a Maya scene. Every time the HAIRSHAPE is asked to redraw itself, the method *updateLight* is called from *prepareForDraw*.

```
for each sort bin i, starting with the farthest
    for each system thread t
        - bind vertex buffer corresponding to thread t
        - bind index buffer corresponding to thread t and bin i
        - draw bound geometry
    end for
end for
```

Figure 2.5: Loop for rendering hair geometry.

The Lights class contains an array of instances of Light class. Each of those instances represents one light object from Maya. There are several types of light that can be represented by the Light class.

- **Directional light** - is an oriented light with an origin at infinity.
- **Point light** - is a light placed in 3D world that lights to all directions.
- **Spot light** - is a light placed in 3D world that lights to a directions limited by light cone. Those directions are defined by a direction vector and a light cone angle.

Ambient lights, producing light from all directions, are not saved as separate Light objects, but just as single color, where all ambient light contributions are summed to.

The Light class is also responsible for generating the shadow maps. If any change that affects shadow maps is done to a light, the corresponding shadow maps are updated. For more details on shadow maps generation please refer to section Section 2.7.

2.4 Implementation of hair fiber representation

As was mentioned earlier, the renderer provides two graphical hair fiber representations to chose from: the line strip and the ribbon. The ribbon representation is not very interesting so I will focus only on discussing the line strip implementation.

To draw the hair fiber with a line strip correctly, it is necessary to compute the line width and addition to the pixel color as described in Section 1.2.3. The width is computed before drawing. As stated in Section 2.3.5, the hair geometry is rendered by sort bins from back to front. Before rendering each bin, the width of all lines in the bin is computed. The lines in one bin have very similar distance from the camera plane and since the line width depends on the same distance, the one width, that is set for all lines in the bin, is very accurate for each individual line. The rendering loop with the line width correction is illustrated in Figure 2.7

Figure 2.6: Class diagram of Light.

When rendering opacity shadow maps Section 2.7, it is not so easy to set the line width. There are no bins that are sorted by distance from light. Fortunately, the opacity maps are not so sensitive to inaccurate computation of the line width. For plausible results, it is sufficient if the distance during the line width computation is set as a distance between the hair model center and the light position.

Note that the maximum line width that can be set depends on the hardware implementation. According to OpenGL capabilities database [3], the maximal width can be 63 pixels on most ATI cards and 10 on most nVidia cards.

The color addition of a thin hair fiber is set by modification of alpha component of line vertex color. The final color of individual pixels on the screen is computed by the hardware alpha blending.

The modification of alpha is computed in GPU vertex shader. The vertex shader is a small program that is executed whenever a GPU is processing a vertex. The main purpose of vertex shader is to transform a vertex from object space to clip space for a rasterizer, but can modify some of the vertex properties as well.

The computation of the alpha modifier is executed in the vertex shader and requires to find the distance between the vertex and the camera plane. This can be easily done by multiplying the vertex position with the modelview matrix. After the multiplication, the camera position will be in the origin and the distance between the vertex and the camera plane will be the z coordinate of the vertex position. Note that this computation is accurate even for opacity shadow maps.

During the rasterization of the line segment, the color with modified alpha is linearly interpolated into fragments (potential pixel color additions) and then blended to final pixels.

[3] http://feedback.wildfiregames.com/report/opengl/

```
for each sort bin i, starting with the farthest
    for each system thread t
        - get the depth of the first bin D_min and length of a single bin d
        - compute the depth D_i of the bin i as D_min + d · i
        - compute the width of a pixel P_3dw
        - get the width of a hair fiber F_w from user interface
        - compute the width of hair fibers W_i in bin i from P_3dw, D_i and F_w
        if W_i ≥ 1
          - set line width for rendering to W_i
        else
          - set line width for rendering to 1
        end
        - bind shader TODO
        - set P_3dw and F_w as parameter for vertex shader
        - render segments in bin i
    end for
end for

In vertex shader:
if primitive = line strip
  - compute the distance D_v from the camera plane to currently processed vertex
    by multiplying the vertex postion with modelview matrix
  - compute line width correction coefficient W_coef
  if W_coef ≤ 1
    - modify the alpha value C_a of the vertex as C_a = C_a · W_coef
  end if
end if
```

Figure 2.7: Rendering with the line width correction.

2.5 Implementation of hair alpha blending

2.5.1 Hair sorting

As discussed in Section 1.3.3, the sorting of hair geometry is required in order to achieve correct alpha blending (unless some order independent transparency technique is used).

The hair sorting takes place in the HairModel class and is executed in the pre-draw stage (see Figure 2.8 for pseudo-code), if the vector from the camera position to the hair model center has changed significantly (the cosine of angle between old and new vector is lower than a constant set to 0.95), or if the hair model has changed. This improves performance for scenes where the camera position or the hair model does not change rapidly.

The volume that is being divided into bins is defined by the hair geometry bounding box. The first bin starts at the closest point of the bounding box and the last bin ends at the farthest point. This way, it is ensured that all hair segments are assigned to a bin and the bins are reasonably small.

The line segment can overlap over multiple bins, but is always assigned to a bin by its farther vertex position. The bins themselves contain a pair of indices that form an assigned line segment (or 6 indices in case of ribbon representation). One bin creates one index buffer object as described in Section 2.3.5.

The geometry is divided into several parts, depending on the number of system threads, before hair sorting, as discussed in Section 2.2.2. It allows easy parallelization of the sorting algorithm. Each thread can sort each part separately, resulting in very good scalability. See Figure 3.5.

```
clear all sort bins
parallel for every system thread t
    for every hair h
        for every segment s of hair h
            - compute the depth of the end points of segment s
            - using the larger depth, compute the bin index i
            - add vertex indices of segment s to the bin i
    end for
end for
```

Figure 2.8: Hair geometry sorting.

During rendering the hair, the update of depth buffer is disabled. It means, that not a single hair's fragment is discarded due to being covered by another hair fragment. Depth buffer test must be enabled so the hairs can be covered by opaque objects that are in front of them and are drawn earlier.

There is one more problem however. Normally, opaque objects are drawn first followed by depth sorted transparent objects. This cannot be easily enforced in Maya. The draw overrides in Viewport 2.0 are executed after opaque rendering, but there is no way to ensure the correct order of rendering for overrides (at least for MPxDrawOverride). This means that if there are more transparent objects in the scene, there is no guarantee that the rendering will be executed in correct order.

2.5.2 Pre-muliplied alpha

The color of a fragment can be stored in RGBA format with either a conventional alpha or with so-called pre-multiplied alpha. The first one has a color stored in RGB components and alpha - representing transparency of the fragment, in A component. Standard blending ($sourceColor * sourcAlpha + destinationColor * (1 - sourceAlpha)$) of color C with a background color B is then:

$$C_{rgb} \cdot C_a + B_{rgb} \cdot (1 - C_a)$$

The color with pre-multiplied alpha is stored as $(\frac{R}{A}, \frac{G}{A}, \frac{B}{A}, A)$ and the blending with settings $(sourceColor * 1 + destinationColor * (1 - sourceAlpha))$ is:

$$C_{rgb} + B_{rgb} \cdot (1 - C_a)$$

This looks theoretically the same, but there is a difference in implementation because the color of a fragment must be clamped to an interval $[0, 1]$. Let's imagine a very transparent fragment that is lit by a strong light. If the vector from light to the fragment is perpendicular to the fragment normal, the diffuse component of color is usually computed as $L_c \cdot L_i \cdot C$, where L_c is the incoming light color, L_i is the light intensity and C is the color and alpha of the fragment, see Section 1.4.1.

Let's say that the $C = \{0.8, 0.7, 0.4, 0.3\}$ (blond), $L_c = \{1, 1, 1\}$ and $L_i = 3$. With conventional alpha, the color value of the fragment before clamping is $\{2.4, 2.1, 1.2, 0.3\}$. After clamping, the color is $\{1, 1, 1, 0.3\}$. When blended to a black background, the final color is $\{0.3, 0.3, 0.3\}$ which is dark gray and not highly lit transparent blond as it should be.

With pre-multiplied alpha, the color before clamping is $\{0.72, 0.63, 0.36, 0.3\}$, same as after clamping and blending to black background. This result is certainly much better than with the conventional alpha, see Figure 2.9.

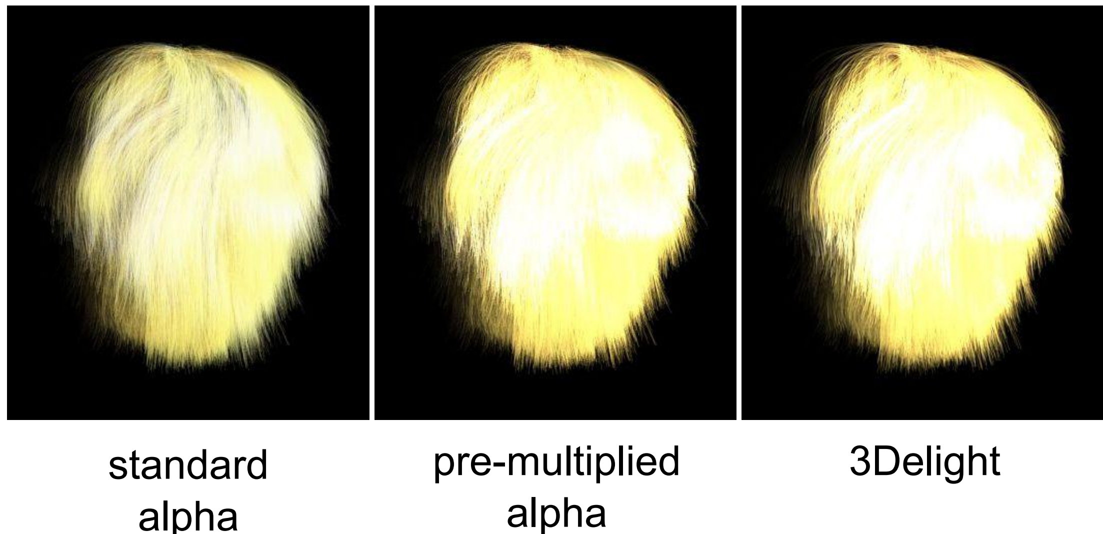


Figure 2.9: Strongly illuminated hair. Shows difference between the standard alpha and the pre-multiplied alpha.

The disadvantage of pre-multiplied alpha is that is does not work with the hardware line anti-aliasing. The hardware anti-aliasing requires the blending to be set to a standard $sourceColor * sourceAlpha + destinationColor * (1 - sourceAlpha)$. Otherwise, the rendering of anti-aliased lines does not give correct good results.

The final implementation uses pre-multiplied alpha because it produces more accurate results. Anti-aliasing can be achieved by enabling multi-sampling in Viewport 2.0.

2.5.3 Multiple lights

When computing a color of an object that is illuminated by multiple lights, it is possible to compute color additions from every single light and sum them together. It can be done per fragment in fragment shader or by rendering the object multiple times with additive blending. The final color of an object illuminated by two lights is:

$$C_{rgb} = (C1_{rgb} + C2_{rgb}) \cdot C_a + B \cdot (1 - C_a)$$

This can be easily done in the fragment shader by summing gains from each light. However, the parameters for all lights have to be uploaded to the fragment shader before rendering. Besides parameters like color, intensity or position, the opacity maps and depth maps must be uploaded. Those maps are 2D textures and the number of textures that can be uploaded to a fragment shader is limited and depends on the graphics hardware implementation. According to OpenGL capabilities database [4], that number is usually 16 on a modern consumer hardware. Each light, in order to cast shadows, needs 4 textures in the current implementation of the renderer. It means that the maximum number of lights that can be computed in the fragment shader is 4.

The rendering with multiple passes with additive blending is not limited to the number of lights. However, it is not usually possible to simply draw the hair once with standard blending and then start to draw it again with additive blending, while illuminated by another light. The additive blending would add full color of every single hair fiber without being covered by other fibers so the result will be far too bright. Note that this technique would be correct if each hair fragment would have had correct color addition computed before blending, as described in Section 1.3.4.

Solution to this problem would be to blend the gains from all lights for every hair segment separately. This method is, however, way too slow. Another approach is rendering the hair with standard blending to an off-screen texture, and then blending this texture to the screen with additive blending. This works well but the screen depth buffer must be copied when rendering to an off-screen texture, which slows down the process. Also, when using hardware multi-sampling, the hair must be rendered into a multi-sampled texture, which cannot be simply bound to an object and blended to the screen. The multi-sampled texture could be down-sampled into another off-screen texture by the framebuffer blitting, or the down-sampling could be done in shader. Both methods cost some additional time for rendering.

The final implementation uses both the rendering with multiple passes and the computing of multiple lights in the fragment shader. It renders with as many lights as possible in the KajiyaKay fragment shader in one pass, and if some more light gains needs to be computed, additional render passes are executed. On a common consumer 3D hardware, 4 lights can be processed in the fragment shader so the number of render passes is a quarter of the number of lights.

[4] `http://feedback.wildfiregames.com/report/opengl/`

2.6 Implementation of light model

The Kajiya Kay light model is implemented in the KajiyaKay fragment shader. The color of each fragment is computed as described in Section 1.4.1 and then multiplied by alpha to get the pre-multiplied alpha RGBA format as discussed in Section 2.5.2.

2.6.1 Light model in 3Delight

The free 3Delight distribution does not have any shaders for hair. Every available shader I tried did not look realistic on hairs, see Figure 2.10. Therefore I implemented, with a collaboration with a Stubble developer Martin Šik, the Kajiya Kay shader for 3Delight.

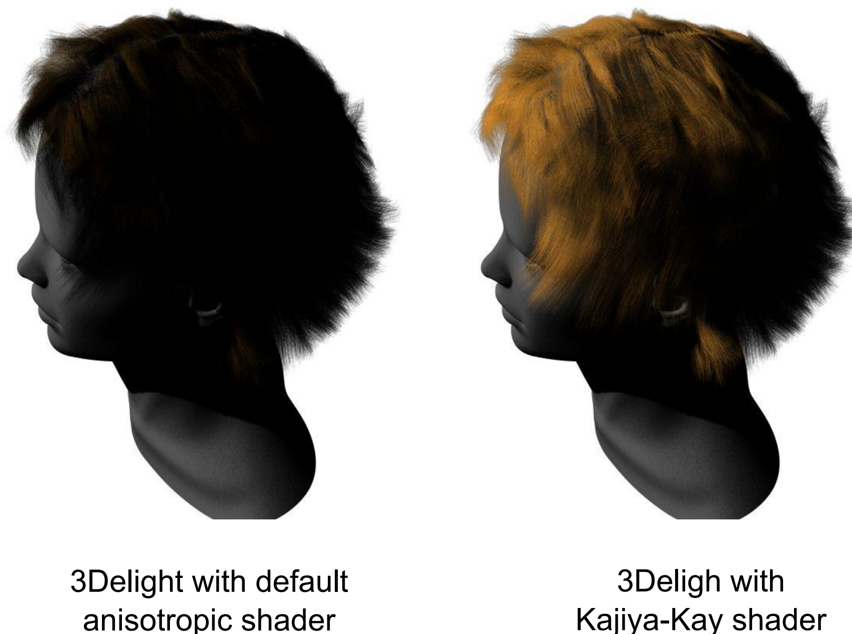


Figure 2.10: Hair rendered with 3Delight's default anisotropic shader is too dark.

Whenever 3Delight renders a Maya shape with a Maya shader, it tries to find a corresponding 3Delight shader in its directory. If a 3Delight shader is found, it is compiled and used for rendering.

In order to tell 3Delight which shader to use, I had to implement a proxy Maya shader. That shader does nothing in Maya, but if it is attached to some Maya shape, the 3Delight will use the corresponding shader to render the shape. The proxy shader is named Stubble3DelightShader and is automatically attached to a Stubble HairShape node at creation. The 3Delight shader is named Stubble3DelightShader.h and is located in *3DelightDirectory/maya/rsl*.

2.7 Implementation of Deep Opacity Maps

As mentioned in Section 2.3.6 na lights, the opacity maps are managed by Light class. The maps are not generated for every frame, but only when the light changes position or direction, or hair model is changed.

2.7.1 Setting camera matrices

The first step is to compute the camera viewmodel and projection matrices for the light. Those matrices differ depending on whether the light is directional or positioned (point light and spot light).

Since the directional light does not have a position, the position of the camera is set to be on the line that has the same direction as the light direction and pass through the hair model center. The projection matrix is orthogonal, which means that the viewport frustum has a shape of a box.

The point or spot light uses a projective projection so their frustum is a culled pyramid. The position of the camera is the same as the position of the light in this case.

The viewport frustum is set to be a minimal frustum that contains the bounding box of the hair model. The smaller the frustum is, the more detailed opacity maps can be rendered. The whole hair geometry must be inside the frustum or a part of the hair will not be correctly shadowed. Note that if the light position is inside the bounding box of hair geometry, the shadows will never be correct.

2.7.2 Rendering maps

All maps are rendered into a 2D textures. In order to render to a texture, the framebuffer object (FBO) is created. A FBO allows to attach a texture to it and then, if the FBO is bound, all the rendered pixels are not displayed on the screen, but saved to an attached texture.

There can be more attached textures than one. The actual maximum number of textures that can be attached depends on the hardware implementation. On most modern graphic cards, there can be 8 color textures, a depth texture and a stencil texture attached to a FBO.

In the first render pass, the depth texture is generated. It is then bound during the second pass to generate opacity maps as described in Section 1.5. Every individual opacity map is saved in one texture channel. The RGBA textures have 4 channels and there are 8 opacity maps rendered in current implementation so it takes two RGBA textures. One more is needed for depth map but depth texture has only one channel.

For opaque objects, there is another depth map. The Deep Opacity Maps cannot sample the light transmittance function very well if it is not smooth. If there is some opaque object in the maps, it will make a sudden drop of the light transmittance function and fragments that are closely in front of the opaque object will be shadowed too strongly. Therefore, opaque objects are handled separately by Shadow Mapping, see Section 1.5.1.

The size of each channel of an opacity map is set to 16 bits. The standard 8 bit channel is not sufficient as it can have only 256 values. It means that 256 hair fibers would make the maximal shadow. The fibers can also have different opacity so it would not be possible to encode the sum of fiber opacities in 8 bit accurately.

During the second pass, the actual opacity maps are rendered. In the Deep-OpacityMap fragment shader, fragments are assigned into individual opacity maps, based on their depth. The fragment shader is well optimized for vec-

tor operations, but is slower if the program is branched. With this in mind, the assigning of fragments into opacity maps is done like this:

$$\overrightarrow{MapColors} = max(0, sign(\overrightarrow{depths} - Z)$$

$$Z = frag_z - D_z$$

where $\overrightarrow{MapColors}$ is a vector of maps in one texture, $\overrightarrow{depths}$ is a vector of relative depths of those maps (predefined), $frag_z$ is the depth of the fragment and D_z is the value from the depth map.

The opacity of a single fragment depends on the fragment's alpha, which is a hair fiber opacity modified according to the fiber width, see Section 2.4. Fragments that fall into one opacity map are summed by additive alpha blending to create the final opacity map.

2.7.3 Final hair rendering pass

In the final hair rendering pass, the depth map with opacity maps are bound and accessed by the fragment shader. In order to compute the light transmitted to a fragment, we need a matrix that transforms a point from the world space to the opacity texture space. That matrix is computed during opacity map rendering and is a light's $modelviewMatrix * projectionMatrix * biasMatrix$ (let's call it opacity map matrix). $modelviewMatrix * projectionMatrix$ transforms a point from the world space to the clip space. In the clip space, the point that is on the left up corner of the screen has x, y coordinates $(-1, 1)$. The point at the left up corner of a texture in the texture space has x, y coordinates $(0, 1)$. The aforementioned $biasMatrix$ transforms the clip space into the texture space.

When a vertex is multiplied by the opacity map matrix, the resulting x, y coordinates corresponds to the opacity texture UV coordinates and resulting z coordinate corresponds to opacity maps depth. The multiplication of a point and an opacity map matrix takes place in the vertex shader and the final coordinates are interpolated for every fragment. With those coordinates, it is relatively easy to interpolate the light transmittance from opacity maps.

The implementation of the light transmittance computation from opacity maps is inspired by work presented by Nguyen and Donelly (2005) [10], but modified to work with the deep opacity maps. As stated above, a shader program works more efficiently if there is no branching. So the transmittance is computed as follows:

$$\begin{aligned}\overrightarrow{OpMapWeigths} = max(0, 1 - (&max(0, (\overrightarrow{depths} - Z) \cdot \overrightarrow{invDepthDeltas}) \\ &+ max(0, (Z - \overrightarrow{depths}) \cdot \overrightarrow{invDepthDeltas'})));\end{aligned}$$

where $\overrightarrow{invDepthDeltas}$ are inverse differences of relative map depths and $\overrightarrow{invDepthDeltas'}$ are those inverse differences shifted one component forward (similar to $>>$ bit operation).

$$Opacity = dot(\overrightarrow{MapValue}, \overrightarrow{OpMapWeigths});$$

where $\overrightarrow{MapValue}$ are values from deep opacity maps.

The transmitted light coefficient is given by:

$$L = \exp(Opacity)$$

The final color of a fragment is given by:

$$C_{out} = C_{Kajiya-Kay} \cdot L$$

where $C_{Kajiya-Kay}$ is the color which is computed by the Kajiya-Kay shading model.

3. Results

This chapter shows the results of the presented renderer. The first section reveals the testing environment. In the second section, the performance of individual renderer's parts is shown and discussed, and the final section presents visual results of the renderer compared to the results from 3Delight.

3.1 Testing environment

The machine used for the testing was a quad-core AMD Phenom X4 955 3.2 GHz, 4GB DDR3, ATI Radeon HD 5770 1GB GDDR5. The renderer ran on OS Windows 7 64-bit in Autodesk Maya 2012 64-bit and the reference images were rendered in 3Delight for Maya 6.0.17 free license version (uses only 2 cores).

3.2 Performance

This section will show measured performance of the individual renderer's parts. The time required for performing GPU-related tasks was measured by OpenGL timer query [1]. The hair segment sorting, which runs only on CPU, was measured by QueryPerformanceCounter.

The scene used for testing is a flat plane with hairs growing out of it, see Figure 3.1. The hairs are quite thick so the ribbon representation does not suffer from the aliasing too much and the number of fragments that are drawn with the ribbon and line strip are similar. Note that with thin hairs, the ribbon representation is significantly faster than the line strip representation, because the number of fragments drawn by the ribbon representation is lower. However, the resulting image does not correspond to the reference image. The shader for the Kajiya-Kay lighting model computation is enabled. The resolution of rendered images is 1024×1024 and self-shadowing is disabled, unless noted otherwise.

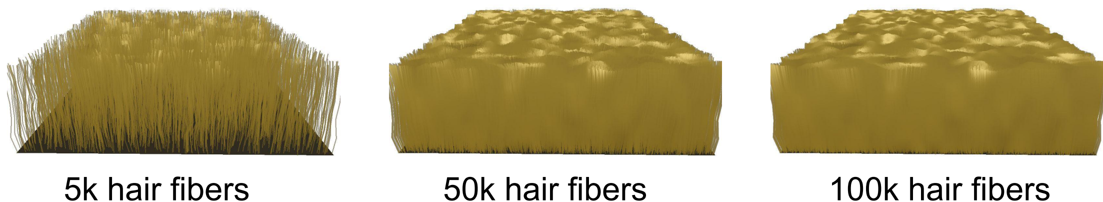


Figure 3.1: The scene for testing performance of the hair fiber geometry representations.

3.2.1 Hair fiber representation

Here, the performance of the ribbon and line strip hair fiber geometry representations will be compared. The influence on the performance of the multi-sample anti-aliasing and the number of hair segments will be also presented.

[1] http://www.lighthouse3d.com/cg-topics/opengl-timer-query/

Figure 3.2 shows the performance of different hair fiber representations. The line strip is slightly faster and dependence on the number of hairs is linear.

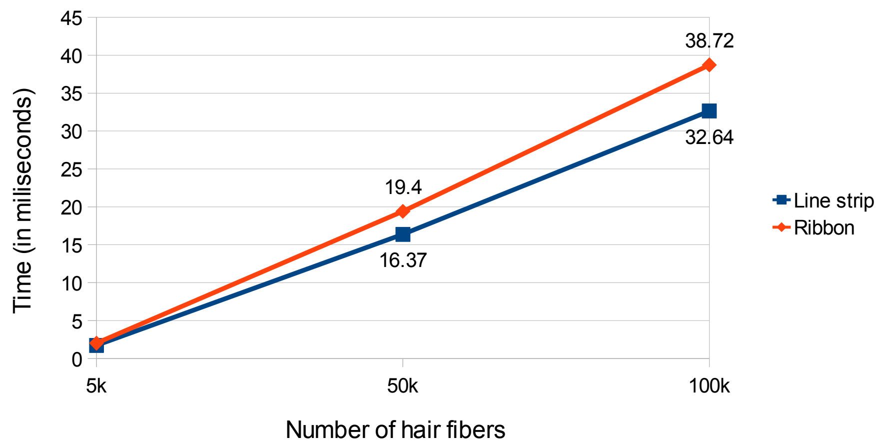


Figure 3.2: The performance of ribbon and line strip hair fiber representations with different hair count. Every hair has 5 segments.

Figure 3.3 shows that the number of segments in a hair fiber does not affect the overall performance very much.

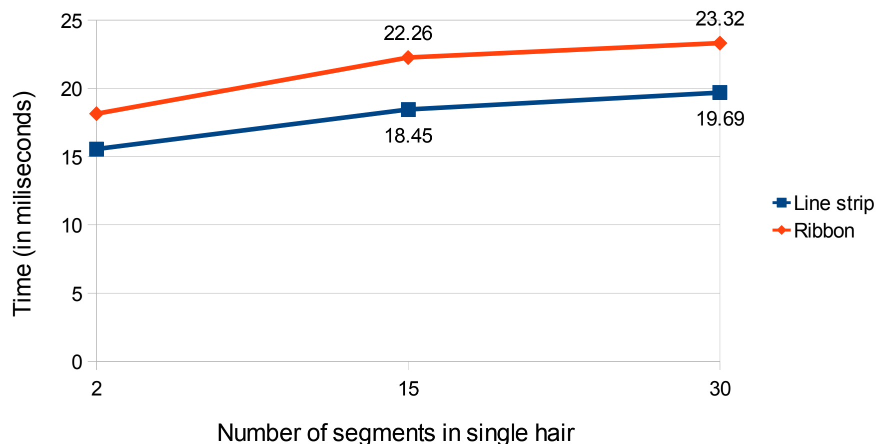


Figure 3.3: The performance of rendering depending on the number of segments per hair fiber. The number of hairs is 50k.

Figure 3.4 shows strong dependence of the performance on the image resolution and multi-sample anti-aliasing.

The line width correction described in Section 1.2.3 did not have any measurable influence on the renderer's performance.

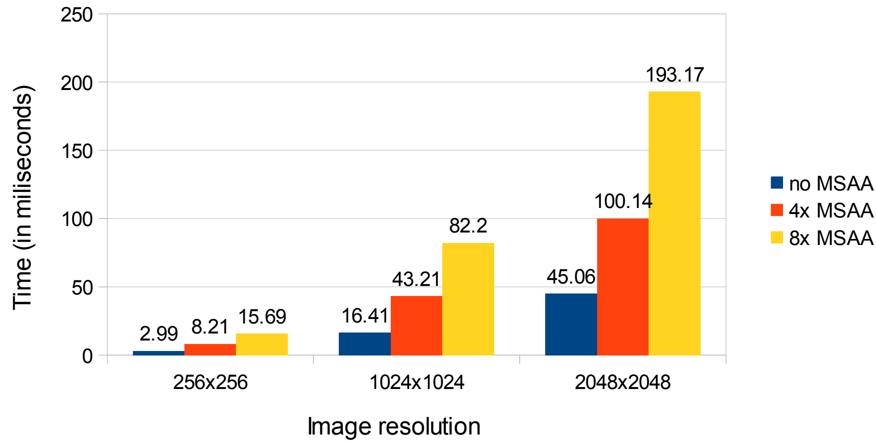


Figure 3.4: Different multi-sample anti-aliasing settings and image resolutions were used int this test.

Conclusion

The presented graphs show that the performance is not very dependent on the hair geometry. The main factor that affects the rendering speed is the number of fragments that are being processed in the fragment shader.

3.2.2 Hair segment sorting

The hair segment sorting algorithm (presented in section Section 2.5.1) relies only on the total number of hair segments. It can be easily parallelized with a very good scalability. See Figure 3.5.

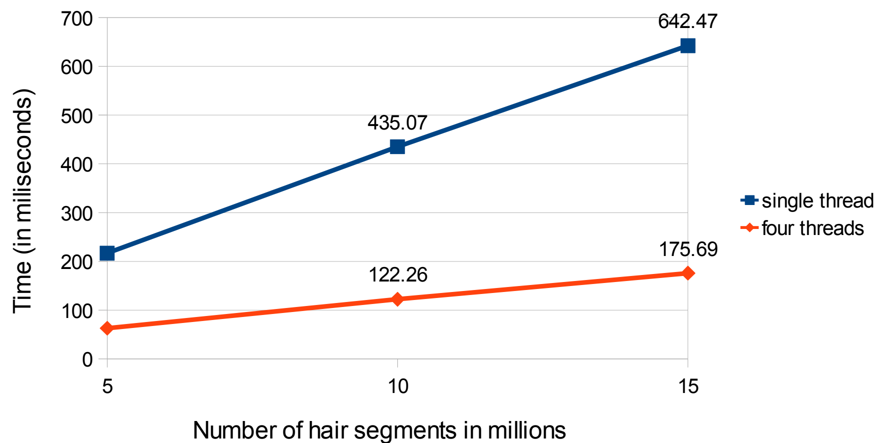


Figure 3.5: Performance of sorting algorithm.

Figure 3.6 shows the difference between the time needed for rendering and the time needed for sorting. Sorting clearly depends on the total number of segments and the rendering is more sensitive to the hair count.

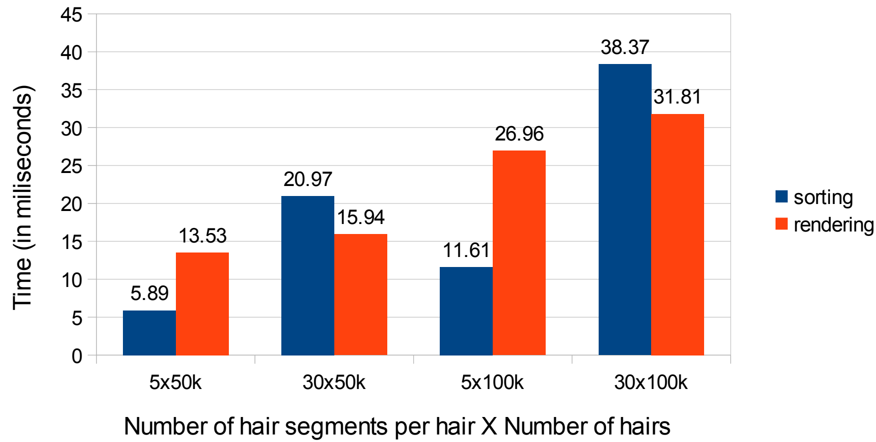


Figure 3.6: Comparison of rendering and sorting times.

3.2.3 Self-shadowing

Shadow map generation

Figure 3.7 shows the time required to generate 8 deep opacity maps and a depth map. It strongly depends on the shadow map resolution and the number of hairs. The key factor here is the number of hair fragments, which rises with both increasing hair count and resolution.

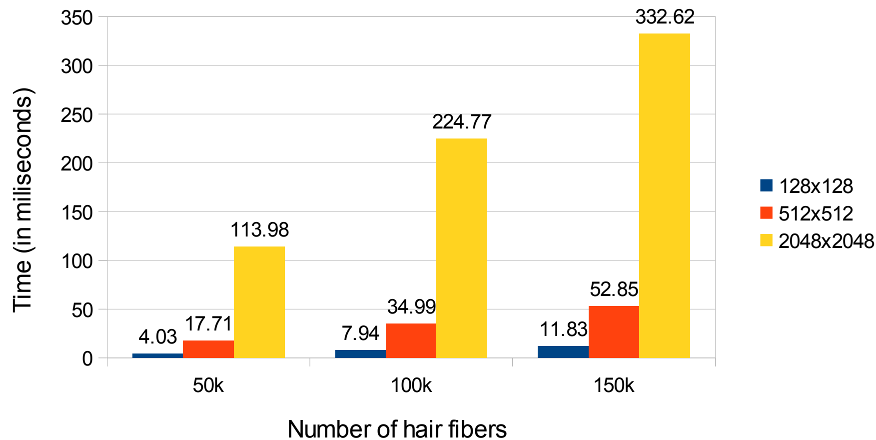


Figure 3.7: Performance of shadow map generation.

Even though the generation of all deep opacity maps is executed in one render pass, the time required by the generation is linear to the number of maps, as can be seen in Figure 3.8. Note that the depth map generation time does not increase with resolution as much as the opacity map generation time as it does not need any computations in the fragment shader. All maps had resolution 512x512.

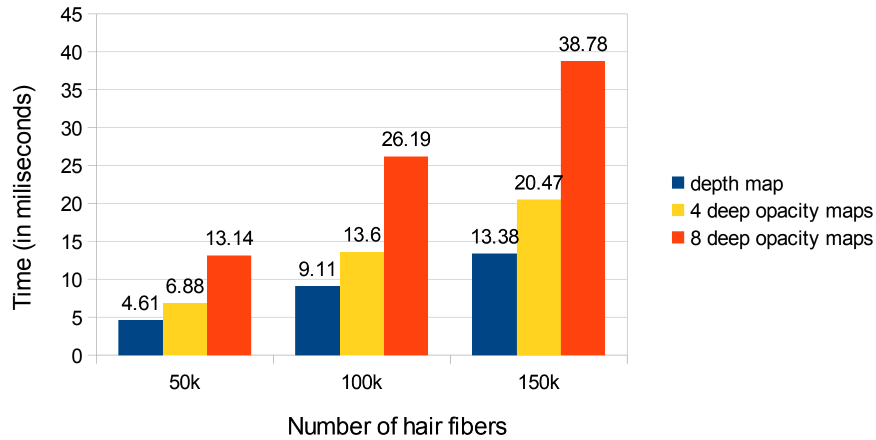


Figure 3.8: Generation times of depth map and deep opacity maps.

Hair rendering with self-shadowing

Figure 3.9 shows how the number of deep opacity maps influences the rendering of final hair.

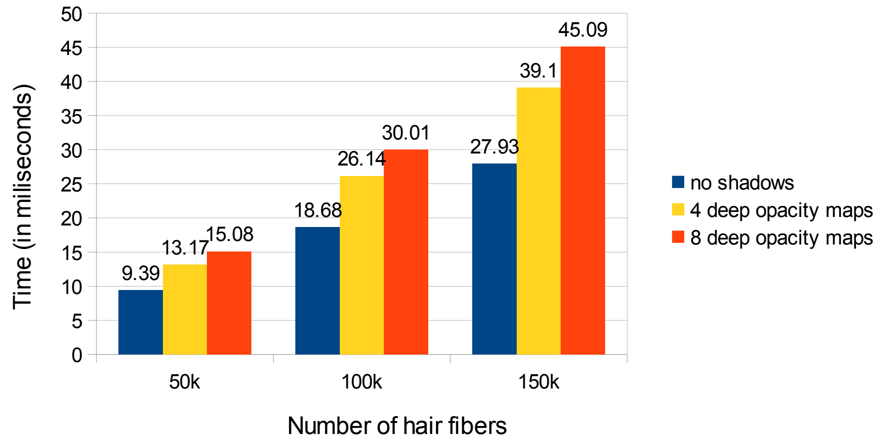


Figure 3.9: Rendering times with and without enabled deep opacity maps.

Memory consumption

Deep opacity maps are stored as channels in four channel textures. Each channel has 16 bits per texel so the 4 deep opacity maps take up $8 * number\ of\ texels$ bytes. Another channel is taken by the depth map. The format of the depth channel depends on the graphics hardware's internal depth format.

The current implementation of the renderer uses 8 opacity maps and the default resolution is 512x512 (can be changed in UI) so it takes up 4 MB plus the depth map.

3.2.4 Multiple lights

Using multiple lights seriously slows down the rendering. Almost all computations in fragment shader depend on light parameters so the computations have to be done for each light separately.

There are two ways to render hair illuminated by multiple lights: summing light gains in the fragment shader and using separate render pass for every light, see Section 2.5.3.

The Figure 3.9 shows the performance of those two methods. Without anti-aliasing, there is not much difference in performance even though with the multiple render pass method, the whole geometry has to be rendered more times. It is due to the fact that the bottleneck of rendering is in the fragment lighting computation and the work that has to be done in fragment shader is very similar for both methods.

However, if the multi-sample anti-aliasing is enabled, the multi render pass method is much slower. There has to be one more step performed: down-sampling of the off-screen texture. This proves to be very slow.

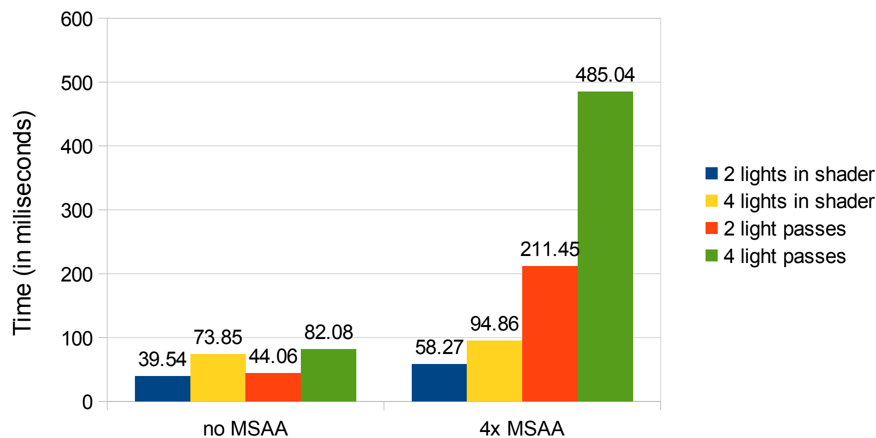


Figure 3.10: Performance of the multi render pass method and the single pass method. Rendered 50k hair fibers with 5 segments each.

The final implementation takes advantage of both aforementioned methods for multi-light rendering. The number of lights that can be processed in the fragment shader is limited to 4 on most hardware, so in every render pass, as many light gains as possible are computed in the fragment shader. If all lights cannot be processed in one pass, then simply more render passes are used.

Figure 3.11 shows how the number of lights affect the performance while using the aforementioned hybrid method.

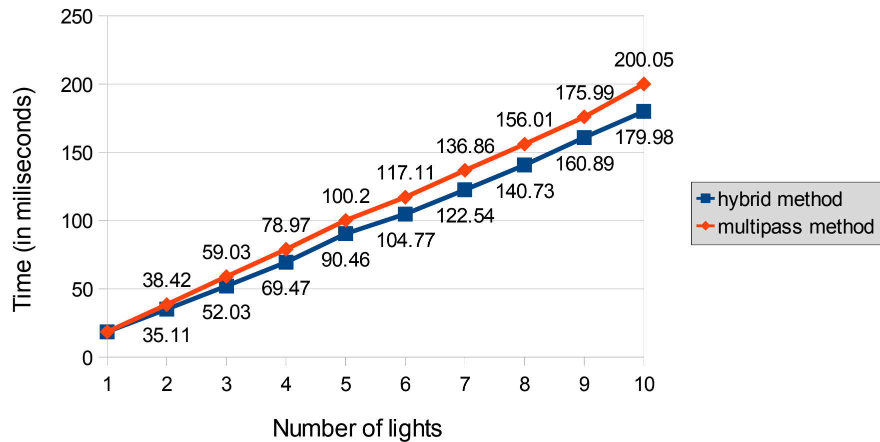


Figure 3.11: The dependence of performance on the number of lights.

3.3 Visual results

This section shows visual results of my renderer in comparison to the 3Delight.

3.3.1 Differences from 3Delight

Here, I will show main sources of differences between my renderer and 3Delight and discuss their causes.

Geometry

The hair fiber in 3Delight is represented as a curve based on Catmull-Rom splines. The hair fiber in my renderer is a line strip. Obviously, the 3Delight curves are smoother and have more precise tangents. The tangents in my renderer are linearly interpolated between vertices. This can cause a difference in specular highlight (see Figure 3.12).

There is also a slight difference in the facing direction of the line strip and 3Delight curve. Lines in OpenGL are drawn facing the camera plane while curves in 3Delight are drawn facing the camera position. In normal hair the difference is not noticeable but can be observer in Figure 3.13.

Light model and blending

As can be seen in Figure 3.13, there is no difference between light models. The difference in blending exists if the geometry is not accurately sorted, see Figure 3.14. Note that the differences are minimal if the blended colors are similar, which is usual for normal hair.

Shadows

The deep opacity maps, which are used for self-shadowing, are not filtered. This causes aliasing in shadows, as can be seen in Figure 3.15. Another issue is that hair cannot cast shadows on other objects rendered by Maya as I did not found

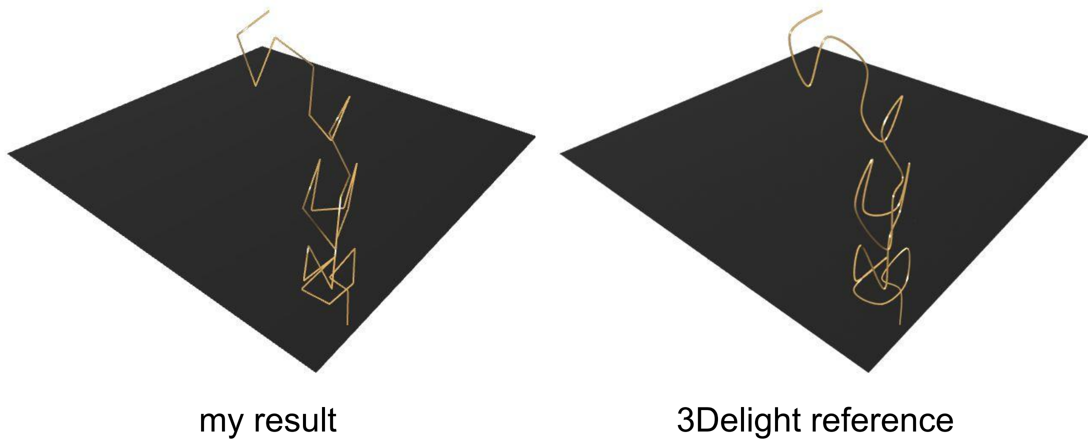


Figure 3.12: Difference in geometry.

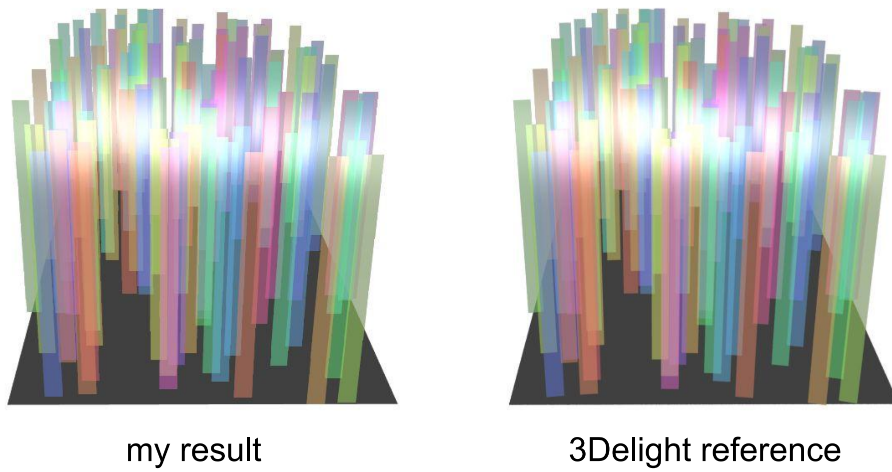


Figure 3.13: There is no visible difference in light model.

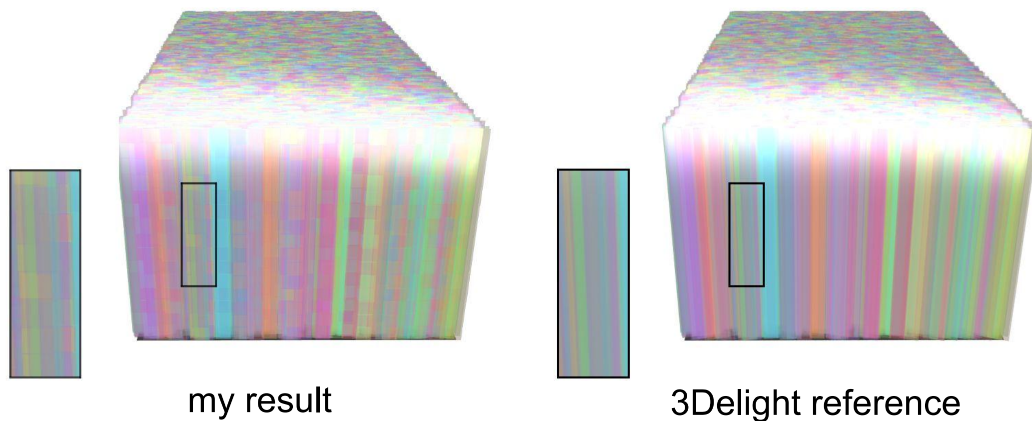


Figure 3.14: When there are many geometry segments close together, they will not be accurately sorted, which causes errors in blending.

a way how to send shadow information to Maya. Also, other objects do not cast

shadows on hair with the exception of the mesh from which the hairs grow. See Figure 3.16.

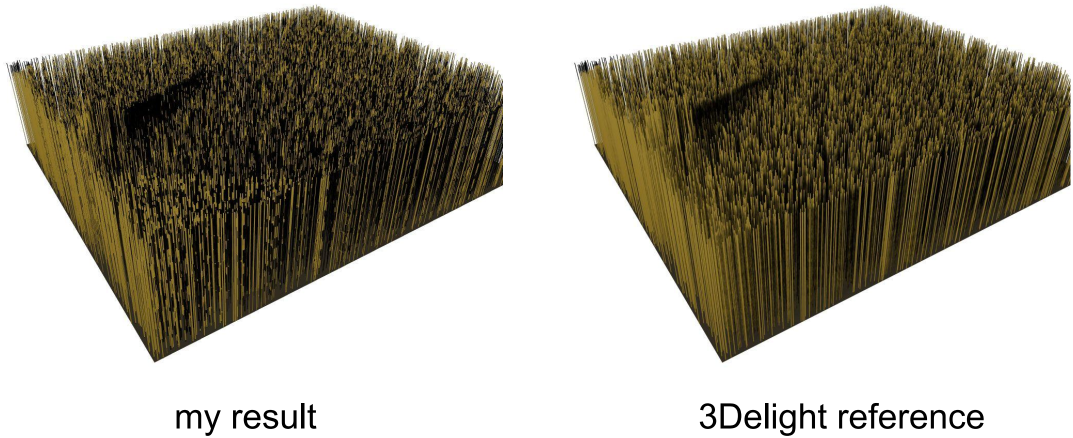


Figure 3.15: Strong shadows cast by thick fibers are visibly aliased.

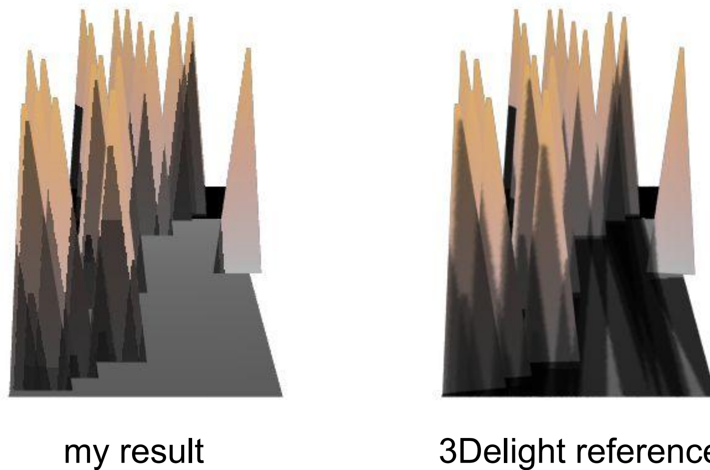


Figure 3.16: Shadows from hairs are not cast on other objects.

3.3.2 Final hair rendering

All presented images were rendered at a resolution of 1024×1024. The FPS values were computed by Maya and do not include the geometry sorting and the shadow map generation. The time values under 3Delight images are pure render times, not counting the time needed for data preparation, which may take few tens of a second.

Figure 3.17: Long curly blond hair with 50k hair fibers and 30 segments per hair.

Figure 3.18: Straight brown hair with 50k hair fibers and 20 segments per hair.

Figure 3.19: Red hair illuminated by 3 spot lights. The model contains 50k hair fibers with 30 segments per hair.

Figure 3.20: Grass rendered using the ribbon representation. The model contains 5k leaves with 30 segments per leave.

Conclusion

The main goal of this thesis was to create a plug-in for Autodesk Maya software that will provide a realistic hair rendering in the user interface of this software. The renderer should allow an interactive manipulation with hair while generating images as close as possible to the output of a high-fidelity rendering software such as Render Man. The hair geometry should be provided from the Stubble hair modeling tool. All these goals were successfully achieved.

Summary

There were several tasks I had to solve in order to implement the renderer.

Hair fiber geometric representation. The geometry of human hair is very complex as it can contain up to 150k thin fibers. The geometry of a single hair fiber must not be too complex in order to ensure real-time rendering. The fact that hair fiber is very thin (under 0.1mm) can cause severe aliasing.

I chose the line strip as the hair fiber geometric representation and developed a method that modifies the width of a line according to the hair fiber's width projected to screen. If the projected hair width is smaller than the pixel width, the color addition to the pixel is approximated. It has practically non-measurable effect on performance and allows to render non-aliased images of sub-pixel geometry, which is crucial for rendering hair.

Alpha blending. Proper usage of alpha blending to achieve transparency effects usually requires that the fragments are drawn in back-to-front order, which is not easily realizable. I implemented an approximate geometry sorting algorithm based on the one presented by Kim (2003) [4].

Light-scattering model of a hair-fiber. The light-scattering, especially in blond hair, can be very complex. However, the speed of computation of light-scattering is crucial in real-time rendering. I chose less accurate but very fast Kajiya-Kay shading model introduced by Kajiya and Kay (1989) [8].

Self-shadowing. The self-shadowing in hair produces very important visual patterns that distinguish one hairstyle from another. I implemented the Deep Opacity Map method for self-shadowing, presented by Yuksel and Keyser (2008) [16].

Future work

The renderer was not tested in practical environment. As such, it may need some improvements especially in communication with Maya and Viewport 2.0. Some issues that should be solved are mentioned in Section 2.3.2. The Viewport 2.0 is still under development and future versions of Maya may add some new functionality that can be exploited by my renderer.

Performance improvement

The fragment shader program should be more optimized. It is the bottleneck for rendering an some optimization in fragment shader may lead to significant improvement of the performance of the whole renderer.

The geometry sorting can be executed on the GPU as presented by Sintorn and Assarsson (2008) [5], which would probably bring some performance improvement.

Visual improvement

- It turned out that the number of hair segments does not influence the performance very much, so the line strip can be more tessellated in the geometry shader to produce curves more similar to the curves in 3Delight.
- Deep opacity maps could be filtered by adding an additional pass after the map generation.
- Finally, more light models can be implemented, as described in Section 1.4.

Bibliography

[1] EVERITT, C. *Interactive order-independent transparency.* Tech. rep., NVIDIA Corporation. 2001

[2] LIU, B. – WEI, L.-Y. – XU, Y.-Q. *Multi-layer depth peeling via fragment sort.* Tech. rep., nVidia, February. 2006

[3] BAVOIL, L. – MEYERS, K. *Order independent transparency with dual depth peeling.* Tech. rep., Microsoft Reasearch. 2008

[4] KIM, T. *Algorithms for Hardware Accelerated Hair Rendering.* Rhythm & Hues Studio. 2003

[5] SINTORN, E., – ASSARSSON, U. *Real-time approximate sorting for self shadowing and transparency in hair rendering.* Proceedings of the 2008 symposium on Interactive 3D graphics and games. 2008. Pages 157 – 162. New York, NY, USA. ISBN: 978-1-59593-983-8 doi: 10.1145/1342250.1342275.

[6] ENDERTON et al. *Stochastic transparency.* Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games. Pages 157 – 164 . New York, NY, USA. ISBN: 978-1-60558-939-8 doi: 10.1145/1730804.1730830.

[7] MULDER, J. – GROEN, F. – VAN WIJK, J. *Pixel masks for screen-door transparency.* In Proceedings of Visualization. pages 351–358.

[8] KAJIYA, J. T. – KAY, T. L. *Rendering fur with three dimensional textures. SIGGRAPH Comput. Graph.* July 1989, 23, 3, s. 271–280. ISSN 0097-8930. doi: 10.1145/74334.74361.

[9] MARSCHNER S. et al. *Light scattering from human hair fibers.* SIGGRAPH 2003 Papers. Pages 780 – 791 New York, NY, USA 2003. ISBN 1-58113-709-5. doi: 10.1145/1201775.882345.

[10] NGUYEN, H. – DONELLY, W. *Hair animation and rendering in the nalu demo.* GPU Gems 2. Pages 361 – 380 ISBN 0-321-33559-7. Available from: `http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter23.html`.

[11] MOON T.– MARSCHNER S. *Simulating multiple scattering in hair using a photon mapping approach.* SIGGRAPH 2006 Papers. Pages 1067 – 1074 New York, NY, USA 2006. ISBN 1-59593-364-6. doi: 10.1145/1179352.1141995.

[12] ZINKE A. et al. *Dual Scattering Approximation for Fast Multiple Scattering in Hair.* SIGGRAPH 2008 Papers. Article No. 32. New York, NY, USA 2008. ISBN 978-1-4503-0112-1. doi: 10.1145/1399504.1360631.

[13] WILLIAMS L *Casting curved shadows on curved surfaces.* SIGGRAPH '78 Proceedings of the 5th annual conference on Computer graphics and interactive techniques. Pages 270 - 274. New York, NY, USA 1978. doi: 10.1145/965139.807402.

[14] LOKOVIC, T. – VEACH, E. *Deep shadow maps.* Proceedings of the 27th annual conference on Computer graphics and interactive techniques. Pages 385 - 392. New York, NY, USA 2000. ISBN 1-58113-208-5. doi: 10.1145/344779.344958.

[15] KIM T.-Y. – NEUMANN U. *Opacity shadow maps.* Proceedings of the 12th Eurographics Workshop on Rendering Techniques . Pages 177 - 182. Springer-Verlag London, UK 2001. ISBN 3-211-83709-4. doi: 10.1145/344779.344958.

[16] YUKSEL, C. – KEYSER, J. *Deep opacity maps.* Computer Graphics Forum. Volume 27, Issue 2, pages 675–680, April 2008 doi: 10.1111/j.1467-8659.2008.01165.x.

[17] SINTORN, E., – ASSARSSON, U. *Hair self shadowing and transparency depth ordering using occupancy maps.* Proceedings of the 2009 symposium on Interactive 3D graphics and games. Pages 67-74. New York, NY, USA 2009. ISBN 978-1-60558-429-4. doi: 10.1145/1507149.1507160.

A. Attached CD's content

There is an attached CD to my thesis, which is structured in the following way:

- **source** This folder contains the source of whole Stubble project, which includes my renderer. The sub-folder **Stubble** contains the source of the plugin to Maya and the sub-folder **StubbleHairGenerator** contains the hair generator library for 3Delight RenderMan. The source files of my renderer are located in the **Stubble/HairShape/Interpolation/VP2_0**.

- **external** This folder contains several libraries which are used by the Stubble project.

- **manual** This folder contains both the developer and the user manual for the Stubble project. It also contains user manual for my renderer.

- The root of the CD contains my thesis as **thesis.pdf** and the instalator of Stubble with my renderer **stubble-setup.exe**. For instructions about Stubble installation and using Stubble see the Stubble user manual.