Ray tracing Maya Hair and Fur

HENRIK RYDGÅRD

Examensarbete

Civilingenjörsprogrammet för datateknik

CHALMERS TEKNISKA HÖGSKOLA
Institutionen för data- och informationsteknik
Avdelningen för datorteknik
Göteborg 2007

Abstract

Rendering good-looking hair is regarded as a hard problem in realistic computer graphics. Turtle, a commercial renderer written and sold by Illuminate Labs, is a fast ray-tracer with powerful support for global illumination, that until now has completely lacked the ability to render hair and fur. This thesis presents work that was done to implement Maya's interpretation of these things in Turtle, and contains some discussion of the various methods and tradeoffs that were used, and a number of suggestions for future work. The result was a working method for rendering Maya Hair and most aspects of Maya Fur. Performance is decent, although memory usage is on the high side.

Sammanfattning

Att rendera hår och päls anses vara ett relativt svårt problem inom realistisk datorgrafik. Turtle, Illuminate Labs' kommersiella renderare, är en snabb raytracer med avancerat stöd för global ljussättning, som tills nu har saknat möjligheter att rendera hår och päls. Denna rapport presenterar arbetet som gjordes för att implementera rendering av Maya's hår- och pälssystem i Turtle, och innehåller diskussion om olika tänkbara metoder och avvägningar som gjorts, plus lite förslag på framtida utökningar. Resultatet blev en fungerande metod för att rendera Maya Hair och de flesta aspekter av Maya Fur. Prestandan är ganska god, men minnesförbrukningen är högre än önskvärt.

Preface

This report presents the implementation and results of a masters thesis at the Computer Science and Engineering program at Chalmers University of Technology in Gothenburg. The work has been performed at Illuminate Labs, Gothenburg.

I'd like to thank David Larson for inviting me to do the project, and the rest of Illuminate Labs for all the support. I'd also like to thank Ulf Assarsson, the examiner, for the feedback.

Contents

Chapter 1

Introduction

1.1 Organization of the report

The report consists of the following parts:

- Introduction
  Provides some background, in the form of a quick introduction to the differences between ray tracing and rasterization, a look at the various renderers for Maya, and some discussion on previous work in the area.

- Implementation
  Discusses various possible implementations and their problems

- Results
  Presents some example images using the code added to Turtle in this thesis work

- Conclusions
  Quick summary of what was done

- Future work
  Discusses the results and proposes further areas of work

1.2 3D rendering

Rendering is the process of taking numerical information describing a three-dimensional world, and turning it into (ideally) realistic images. The two main ways of doing this are rasterization (scanline rendering) and ray tracing.

1.2.1 Rasterization (Scanline rendering)

Scanline rendering (figure 1.1), or rasterization, is the process of taking a boundary representation, generally a triangle mesh, of a 3D scene, transforming it into screen space, and filling the resulting 2D geometry on the screen. This is the approach used by the vast majority of computer games and other interactive

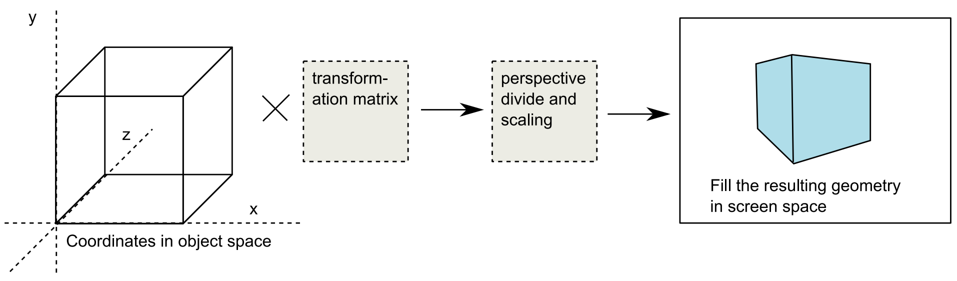


Figure 1.1: Scanline rendering (rasterization)

visualizations, because modern commodity 3D graphics hardware can rasterize at completely mind-boggling speeds.

The primary two reasons that rasterization can be done so quickly is the extremely high level of memory coherency that can be achieved, and the large amount of implicit parallellism that can be exploited. The shading and texturing of one pixel will in most cases access roughly the same memory and perform similar calculations as the ones around it, and they are no interdependencies. Modern hardware thrives on memory coherency, since memory latencies are rising relative to the processing power, and caches (which exploit coherency to reduce latency) are used to mitigate this. Graphics hardware can contain carefully chosen amounts of extremely specialized caches to optimally exploit coherency, and can easily contain duplicated processing units to fill more pixels in parallel. Another advantage of rasterization over ray tracing is that when rasterizing, the entire scene does not have to be kept in memory, because it can be drawn piece by piece.

1.2.2 Ray tracing

Ray tracing generates images by shooting imaginary rays into the scene and computing where they will hit (figure 1.2), and then use derived information like the surface normal and the locations of light sources to compute the color of the surface at the hit points. If the user is willing to spend more time generating images, ray tracing makes it easy to add secondary effects such as reflections, shadows and global illumination by simply shooting more rays in smart ways. These effects are much harder or even impossible to do correctly in a scanline renderer.

Ray tracers generally achieve much lower levels of memory coherency than rasterizers do, because neighbouring rays often spawn sub-rays that take completely different paths through the scene, and objects are generally not textured and shaded in order, leading to complexity and cache thrashing. Also, a general ray tracer must keep the entire scene in memory (there are ways around this, but they are less than elegant and have a large performance cost).

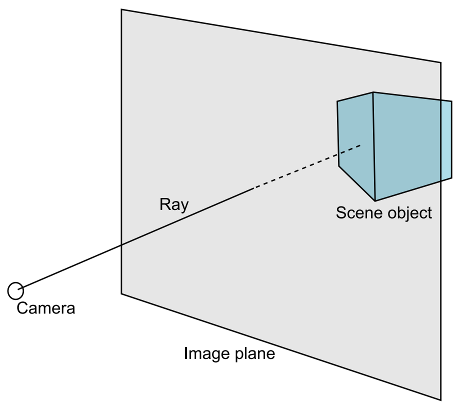


Figure 1.2: Ray tracing

1.2.3 The eternal debate

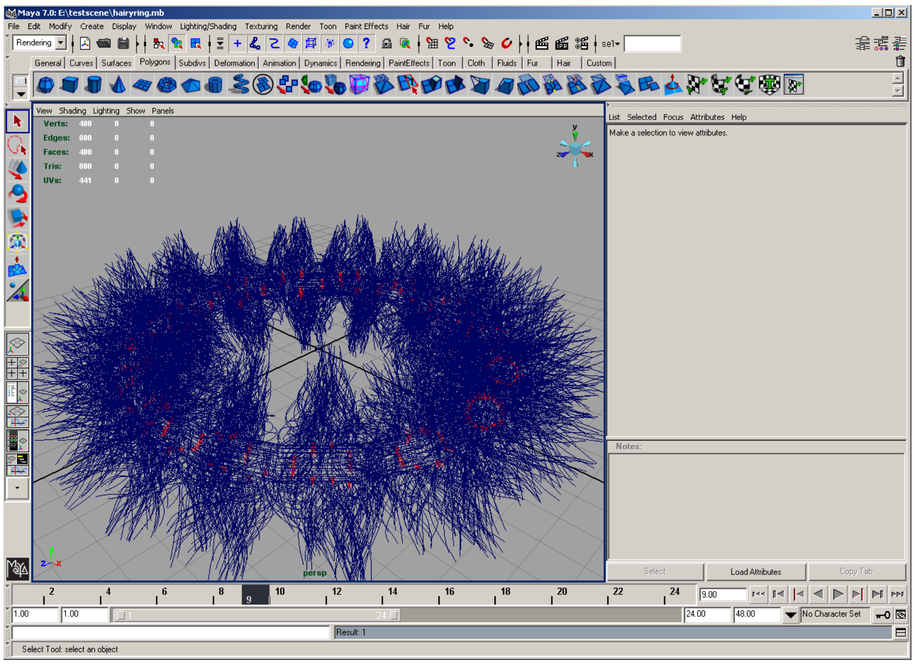
Since ray tracing speed, with proper acceleration structures, is $O(\log n)$ to the number of primitives in the scene, while rasterization is generally $O(n)$, ray tracing proponents have long said that it will eventually overtake rasterization as the fastest and most popular rendering algorithm. Not only because of its inherent computational complexity advantage, but also the ease with which complicated optical effects can be implemented in ray tracers.

However, the incredible advances of the graphics hardware industry over the last few years have prevented this from happening. Rasterization is still several orders of magnitudes faster for game-type interactive scenes, with raytracers only beating rasterizers in extreme cases of static high-poly scenes, like rendering a 350 million polygon model of an entire Boeing 777 airplane [2], where the $O(n)$ performance of rasterizers really hurts compared to the $O(\log n)$ performance of ray tracers.

The transition to ray tracing, or various combinations of rasterization and ray tracing, is however already happening in the offline rendering scene, where high real-time performance is not necessary, but where rendering quality is the most important goal. Rasterization is still in widespread use, however.

The ray tracing proponents who emphasize the complexity advantage of ray tracing are also often forgetting one important thing: The idea that ray tracing is $O(\log n)$ only applies when rendering a static scene, since all ray tracing acceleration structures are per definition at least $O(n)$ to build (they must contain all the geometry of the scene). Some structures do support incremental updates, however, making many kinds of dynamic scenes more viable.

Recently, there has been some research into implementing hardware to accelerate ray tracing, such as [21]. This hardware is of course also affected by the problem of acceleration structure building.


File Edit Modify Create Display Window Lighting/Shading Texturing Render Toon Paint Effects Hair Fur Help
Rendering
General Curves Surfaces Polygons Subdivs Deformation Animation Dynamics Rendering PaintEffects Toon Cloth Fluids Fur Hair Custom
View Shading Lighting Show Panels
Verts: 400 0 0
Edges: 800 0 0
Faces: 400 0 0
Tris: 800 0 0
UVs: 441 0 0
persp
List Selected Focus Attributes Help
Notes:
Select
Load Attributes
Copy Tab
No Character Set

Maya 7.0: E:\testscene\hairyring.mb

Make a selection to view attributes.

Result: 1

Select Tool: select an object

Figure 1.3: The Maya main window

1.3 Maya

Alias Systems' *Maya* (figure 1.3) is one of the most popular 3D modelling and rendering software packages. It's used by many big special effects houses, in game development studios, in advertising companies, and more. Many special effects in blockbuster feature films are designed and rendered using Maya. Some examples of films where Maya has been used are The Fifth Element, The Matrix, Shrek and Final Fantasy - The Spirits Within.

1.3.1 Maya renderers

Maya has the capability of supporting multiple renderers, supplied as plug-ins. *Mental Ray* and *Maya Software* are two such renderers, both of which are shipped together with Maya Unlimited by default.

Maya Software is a reasonably fast, generic scanline renderer that supports rendering almost all features in Maya, and has some limited ray-tracing support.

Mental Ray is, as the name suggests, primarily a ray tracer. It also supports most, but not all, features in Maya. It's got much more complete ray tracing support than Maya Software, but it is also quite a bit slower in most cases.

Turtle, by Illuminate Labs, is also a ray tracer. In many cases, especially complex ones, it's much faster than Mental Ray and produces images of the same or better quality. However, it doesn't support quite as many specialized features. Hair, fur, and particle support were all missing until this project started. Hair and fur have now been partially implemented, and the design, development and results of these features is what this report is about. A proof of concept implementation of getting particle data out of Maya and then rendering purple placeholder spheres was also done, but is out of the scope of this report.

1.3.2 Maya Hair

Maya Unlimited has a system for generating and physically simulating the behaviour of bunches of hair, called *Maya Hair*. It can generate straight or curly hair, and has parameters to specify things like stiffness, thickness and other such things that will affect the hair simulation.

1.3.3 Maya Fur

Maya Fur is a system that makes it easy to apply nice-looking fur to models. It has support for easily painting on attributes like length, curl, inclination etc, which makes it possible to virtually comb it using a mouse. However, it has no support for physical simulation. It is possible to make it react to movement, though, by running a sparse invisible Maya Hair system in parallel and setting the fur up to imitate the hair movement, by using a feature called ”Attractors”.

1.4 Problem statement

The problem tackled in this paper is how to extend the Turtle ray tracer with support for rendering Maya Hair and Fur. As many subfeatures as possible are desirable, and the rendering quality should be comparable with the results obtained from Maya's default renderers.

It is important to note that the main focus of this work has been to find ways to efficiently add the ability to render Maya Hair and Fur in Turtle, not to create innovative new hair rendering techniques.

1.5 Previous Work

When rendering hair and fur, the main problems are modeling or generating the hair/fur, simulating its movements, and rendering it. With hair, the first two of these are already taken care of by Maya's system. With fur, all three had to be handled.

1.5.1 Categorisation

Previous approaches to rendering hair and fur can be classified into a few different categories:

1. Renderers that try to capture the general appearance of fur at a distance without actually rendering the hairs [5]

2. Methods for grossly approximating the appearance of fur using textured geometry (mostly used in realtime applications like games, where speed always is more imporant than image quality) [9] [18] [14]

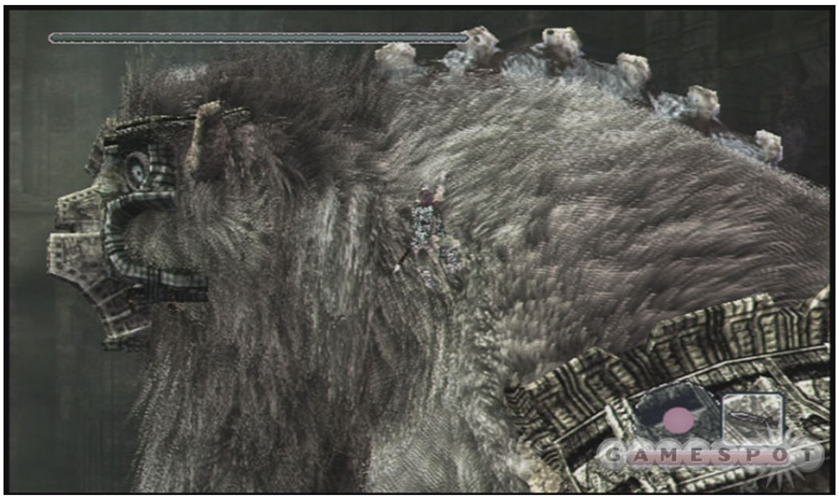


Figure 1.4: Shells + fins fur rendering in *Shadow of the Colossus* on the PS2 - nowhere near movie-quality rendering, but thanks to good artistry it easily looks good enough for a game and runs at realtime framerates

3. Renderers that actually render individual hairs [4] [1]

1.5.2 Implementations

Most papers focus either on hair or on fur. Since fur can be regarded as short uniform hair, in many cases decent fur can in theory be rendered by many of these methods.

Almost all previous implementations that belong to the third category actually render fur by rasterizing various varieties of paint strokes or layers. Usually, hairs are splines that are split up into short straight-line segments for easy rendering.

[13] is a combination of categories 2 and 3. They render invididual hairs up close, and ”clumps” when less detail is needed, and wide ”strips” as the lowest level of detail, used for example at large distances. They motivate the choice to ”clump” with hair’s natural tendency to do just that in the presence of static electricity, and due to the properties of the oils naturally present on hair strands.

Hoppe[9] proposed a fur rendering algorithms that simply uses a large number of transparent texture layers containing dots, extending out from the surface of the model, called ”shells”. He also added ”fins” extending out from the edges between triangles, to improve the appearance from grazing view angles. The result is a very fast way to blast low-quality but decent looking short fur onto a model using graphics hardware, an ideal scenario for games. This method clearly belongs to category 2. It has been successfully used in a number of games. The method can be spotted in for example *Halo 2*, *Conker - Live and Reloaded* and *Shadow of the Colossus* (figure 1.4). This technique requires nearly no extra geometry storage on hardware that supports ”vertex shading”, like all modern consoles except the Nintendo Gamecube and Wii, but consumes huge amount of texture mapping fill rate. Fortunately, many modern game consoles and all new PC graphics cards have enourmous fill rate, often a number of gigapixels per second.

Joe Alter's Shave and a Haircut is a commercial fur and hair generator, which has its own renderer that is meant to be used as a post-process, but can also integrate into Mental Ray. The rendering techniques it uses are basically the same as what Maya Software and Mental Ray do with Maya Fur, but it appears to be highly regarded for its intuitive hair sculpting and combing tools. Maya Fur also has such tools, but they appear to be quite clumsy in comparison.

1.5.3 More academic papers, and custom movie industry work

James Kajiya proposed a volume rendering approach for fur rendering in [12]. Essentially, he precomputed volumetric fur tiles by drawing 3D lines into volume textures, and bent tiled volumes of those around the surface of the animal models. This method never seems to have become popular, most likely because it's too slow for realtime rendering (unless you do heavy approximations and tricks, turning it into the shells+fins method), and too low-quality and too hard to control for high-quality offline rendering.

For *101 Dalmatians*, Dan B Goldman of Industrial Light and Magic used a traditional rasterizing fur renderer for closeup shots, and gradually transitioned to a simple Renderman surface shader as distance from the dogs to the camera increased. At a distance, the approximation is so good that it is virtually indistinguishable from the real fur rendering, so no actual fur has to be rendered. This only works for animals with very short fur, of course, and not for closeup shots.

For the many Agent Smith clones in *The Matrix Reloaded*, ESC Entertainment used "Gossamer", an in-house proprietary tool that, just like was done for this report, duplicated Maya Fur, but they added some parameters of their own. They did the fur generation as a geometry plugin however, and used Mental Ray for rendering. As described in their writeup [22], they had to collaborate with Mental Images to acheive a solution that could render the hairs efficiently. It is not known whether this work made it into the commercial Mental Ray package in any form.

1.5.4 Renderers that do Maya Hair and Fur

For the specific task of rendering Maya Hair and Fur, many other Maya renderers simply choose to not support it. The partially hardware accelerated plugin renderer nVidia Gelato, for example, does not support Maya Hair and Fur at all. Pixar does have a product called UltraFur which claims to use intermediate files from Maya's fur renderer to be able to render the fur in Renderman. The temporary files produced by the fur render does not appear to contain fur hair data, however, so it is unknown what intermediate files it is talking about. Also, RenderMan is traditionally not a ray tracer, although the newer PRMan does do some ray tracing. Without access to a Renderman licence, it was not possible to investigate exactly how good its Maya Hair and Fur support is.

1.5.5 Classification of this paper

Turtle is fundamentally a ray tracer and as such, a method for ray tracing hair and fur is required for this project. It would certainly be feasible to add rasterized hair as a post process, like Maya Software, although there is no infrastructure in Turtle for doing this at present, and it also would negate the fact that Turtle is usually used where realistic rendering of secondary lighting (see 2.1.9) is required. Therefore, ray tracing was chosen, unlike most other papers.

The methods described in this paper mostly belong to the third category (1.5.1). A variable-width cylinder segment primitive is integrated into Turtle, and it's used to intersect and render both segments of hair and fur strands. The shading is not yet very sophisticated, it's essentially a variation of the anisotropic shading method that was first introduced by Kajiya in [11]. ”Level of Detail” through stochastic pruning, as introduced by Pixar in [17], is used to reduce aliasing and shimmering, giving the method somewhat of a category 2 flavor.

Chapter 2

Implementation

2.1 Design

2.1.1 Wish list

- Ability to render all scenes with hair and fur that Maya Software and Mental Ray can handle
- Preferably do so with better speed and/or quality than Maya Software and Mental Ray.
- Interact nicely with Global Illumination techniques supported in Turtle, like Final Gather and Photon Mapping.

2.1.2 Programming language

Since Turtle is written in C++, this is what has been used for the main code. To access information from Maya and plug the new settings into the GUI, Maya's C++ bindings and *MEL*, Maya's scripting language, have been used.

2.1.3 Modeling hair

Maya Hair is a system for hair modeling and simulation. It's bundled with Maya Unlimited. To use it in Maya, select the object that should be hairy, and then use the Hair tool on that. A dialog pops up where various parameters affecting the distribution of "master hairs" can be tweaked.

For each fully physically simulated master hair, Maya Hair creates a configurable amount of simple hairs that will follow their master hair. These hairs together are called a "hair clump". This is probably done to reduce the computational load of simulating every hair.

2.1.4 Getting access to the hair and fur data from the 3D scene

Fortunately, Maya Hair has a public API for accessing data like coordinates and colors of each simulated hair.

There is, however, no publicly available API for accessing the individual hairs of Maya Fur. But through some complicated, and unfortunately somewhat risky, Maya API usage, it is possible to access all the parameters and attribute maps, and reimplement the fur generator from scratch. This is what was done. The method that was designed for intersecting Maya Hair works just as well for fur hairs, and a similar lighting equation was be used.

2.1.5 The fur reimplementation

The fur generator was implemented from scratch, without any source code or any real specifications to look at. Therefore, the behaviour of the generator is not identical to Maya's, but it is similar for most of the supported parameters.

Fur placement

When generating fur, we need a way to place the hairs on the surface of the model. It is fortunately pretty easy to place large amounts of hairs reasonably uniformly across the surface of triangle meshes:

- For each triangle in the base model:
  - Repeat a number of times proportional to the integer rounded result of [triangle area plus a small random number]:
    1. Generate two random numbers between 0 and 1
    2. If their sum is > 0.5, flip them both (number := 1-number). The two numbers now represent a random coordinate inside the 2D triangle made up of the points (0,0), (1,0), (0,1)
    3. Get world coordinates by treating the two numbers as coordinates across the 3D triangle ("barycentric coordinates")
    4. Put a hair root at those coordinates

If the goal would be to place hairs at an exact number of points randomly but evenly distributed over the triangle mesh surface, more accurate algorithms could be developed, but this one is good enough for fur generation.

After this is done, loop through the fur roots and generate a fur hair at every one of them, according to the rest of the fur parameters like base and tip curl, polar direction, slope, etc. This is the most code-heavy part of the process, and consists mostly of line after line of rather uninteresting vector math and various ad-hoc methods that appear to work roughly like Maya's fur generator.

Missing features

- Fur clumping and attraction (control fur using hair)
- Fur/hair tracking (used to control fur systems with hair systems)
- Ambient light can not be noised/mapped, due to limitations in Turtle's color management

Features that are different

- The shading is not the same.
- Fur density and scaling behaves differently in some situations.
- Transparency (opacity) does work but is very inefficient.

2.1.6 Intersecting hair

Maya's Software renderer simply paints the segments by projecting the 3D line coordinates into 2D and drawing simulated brush strokes, which is of course very fast but does not support effects like reflection, refraction and anything other than simple shadow map shadows.

Turtle is a raytracer, however, so we need a way to intersect a hair segment with a ray to be able to render it.

Since hair strands are generally very thin when rendered, small intersection errors are acceptable, since they will be practically unnoticable. Therefore, if an approximation can be found that computes almost correct intersections, while being faster than intersecting the "perfect" tapered cylinder shape (which itself is a crude approximation), it could still be very usable.

Both Maya Software and Mental Ray cheat to some degree. A comparison shot of a closeup of a small bunch of hair can be seen in figure 2.1.

In the implementation described in this paper, tweaked cylinder segments with variable radius along their axis were used. Using clipped cones would perhaps be the obvious solution, but intersecting arbitrary-oriented cylinders and faking the varying radius can be done quite easily [10] without having to transform the rays into a local cone space, and looks just as good in practice.

2.1.7 Hair data structure

In Turtle, all primitives (before this project, there were only triangles and displacement-mapped triangles) inherit from the Intersectable class. Triangles don't directly contain the coordinates of their corners, but 3 indices into a list of vertices. The logical thing to do was to process with the same structure for hair segments.

So, a hair segment now contains:

Data	Type	Total Size
Endpoint indices	2x Integer	8 bytes
T values (along hair)	2x Float	8 bytes
Precomputed tangents	6x Float	24 bytes
Length	1x Float	4 bytes
2xAlpha	2x Float	8 bytes
Lod level	1x Integer	4 bytes
Total	-	56 bytes

For initial ease of implementation, tangents are stored in the hair segments instead of in the vertices, duplicating their memory usage. In addition, there's the vertex data that the endpoints refer to. On average, we need slightly over one vertex per hair (since they share vertices except at the endpoints).

A simple vertex contains the following:

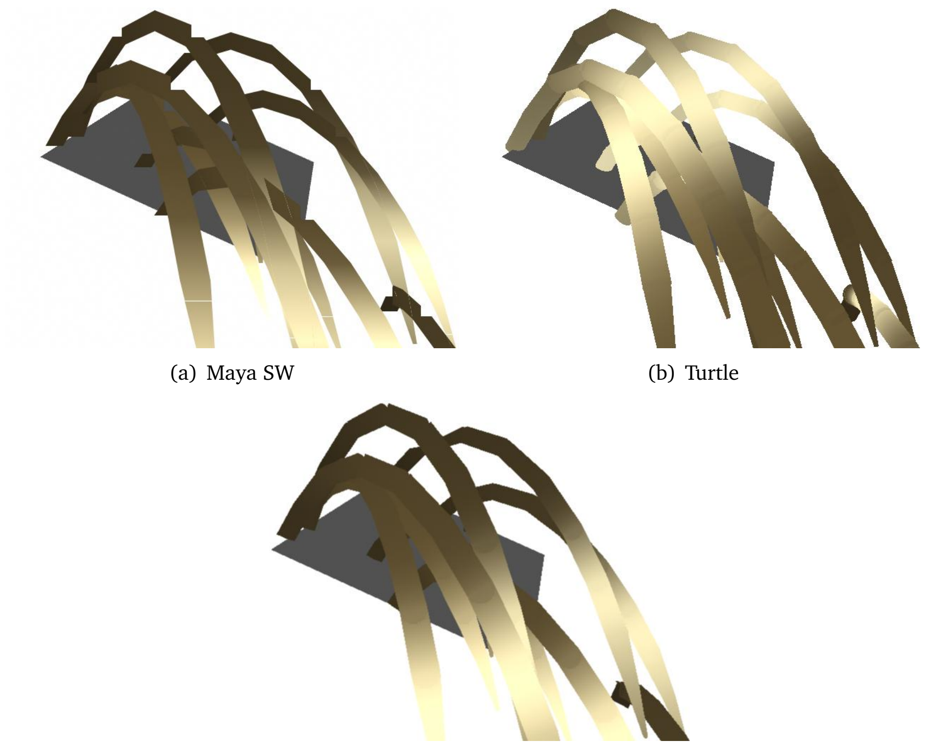

(a) Maya SW (b) Turtle (c) Mental Ray

Figure 2.1: Closeup of some hair. Note the uniform shading across the width of the hairs, and the less than perfect shapes, caused by approximations in intersection formulas, or in the case of Maya Software, a rasterization algorithm that cheats

Data	Type	Total Size
Point	3x Float	12 bytes
Normal	3x Half-Float	6 bytes
UV	2x Float	8 bytes
Total	-	26 bytes

In total, one hair segment consumes approximately 56+26 = 82 bytes of memory.

The focus when doing this work was directed more towards adding functionality than towards maximizing efficiency. It would certainly be possible to greatly reduce the memory usage in the future. Here are some possibilities:

1. Compute segment length on the fly when needed instead of storing it
2. Storing float values at half (16-bit) precision, wherever full 32-bit precision is not necessary
3. Moving the tangent vectors to the vertices, so they don't have to be stored twice

Some of these optimizations have a performance hit, which would have to be weighed against the reduction of memory usage. However, due to the presence and behaviour of memory caches in modern computer architectures, and the comparatively high speed of computation of modern CPU:s, reducing the memory consumption can directly improve performance, more than covering the costs of the additional computation. This means that in many cases, what first looks like a speed/size tradeoff is often a win/win situation. However, this reasoning ignores the additional development costs, of course.

2.1.8 Shading model

Hair generally consists of a large amount of very thin primitives, which means that using the surface normal of each intersection point with the hair is quite meaningless. The normal vector will vary extremely quickly across the very non-continuous perceived ”hair surface”, so the result will be an large amount of noise and Moiré patterns, depending on the sampling method.

It would be useful to have some direction vector that's more consistent over a volume of approximately codirectional hair, and that can also be smoothly interpolated along the hair strands across segments. Fortunately, the tangent vector along the hairs is a very good candidate. The tangent vector can be approximated at each vertex by subtracting the worldspace coordinates of the previous vertex from the worldspace coordinates of the next vertex, and normalizing[1] the resulting vector, as shown in figure 2.2. The lighting equation has to be modified too. This method is mentioned and used in [18].

This also means that the shading is completely uniform across the width of a hair, which can be easily seen in figure 2.1. It is very obvious that the shading doesn't reflect the ”roundness” of the hairs, and that all three renderers cheat a lot. At larger distances, this cheating isn't noticable at all.

This approach to lighting is called Tube Shading in Maya. If the hair strand is very zig-zaggy, this will not give very good results, but then it will also look jaggy and should have been made out of more segments to start with. Maya provides a ”subsegments” parameter that subdivide each hair strand into smaller segments to smooth them out, if this ever would become a problem.

One may view this type of lighting as approximating the sum of the simulated lighting over the ”disk” of normals perpendicular to each shading point on the hair. Internal scattering ([19]) is not yet taken into account.

The actual lighting calculation methods used are somewhat ad-hoc, and don't really have a sound mathematical foundation, especially not 2.6. However, this is the case with a lot of common techniques used in rendering, like the specular highlights of Phong shading. Rough approximations that look good and are easy to tweak are generally more useful than mathematically rigourous methods that take enourmous amounts of time to run.

Legend:

$Shade(\mathbb{P}, \mathbb{N})$ = Turtle standard non-attenuated lighting using this position and normal

$\mathbb{L}$ = Light vector (towards light from intersection point)

$\mathbb{T}$ = Tangent vector

$Norm(\mathbb{V}) = \frac{\mathbb{V}}{|\mathbb{V}|}$

[1]Rescaling to length 1

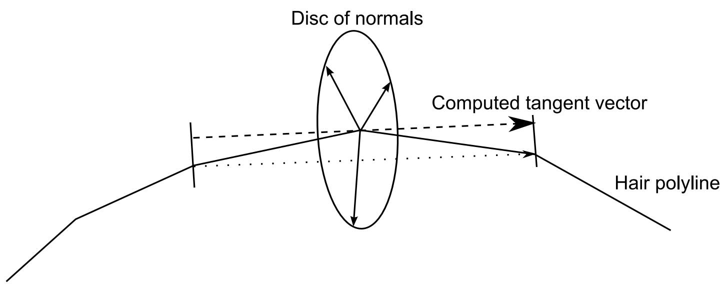


Figure 2.2: Computing the hair vertex tangents used for shading

Hair

$$\begin{aligned}
H &= norm(\mathbb{L} - rayDir) & (2.1)\\
S &= C_{spec}(1 - (\mathbb{T} \cdot \mathbb{H})^2)^{specPower} & (2.2)\\
C_{final} &= C_{ambient} + C_{diffuse} Shade(\mathbb{P}, \mathbb{T}) + SC_{specular} & (2.3)
\end{aligned}$$

Fur

$$\begin{aligned}
H &= norm(\mathbb{L} - rayDir) & (2.4)\\
S &= C_{spec}(1 - (\mathbb{T} \cdot \mathbb{H})^2)^{specPower} & (2.5)\\
D &= 1 - (\mathbb{T} \cdot \mathbb{H})^4 & (2.6)\\
C_{final} &= C_{ambient} + DC_{diffuse} Shade(\mathbb{P}, \mathbb{T}) + SC_{specular} & (2.7)
\end{aligned}$$

It's not actually correct to pass the hair tangent into Shade, but it does deliver okay-looking results, since for many types of lights the normal is mostly irrelevant in this step, and a wrong but continuous normal is better than no or random normals...

2.1.9 Global illumination

Global illumination is a collective name for techniques that attempt to simulate more than direct (local) lighting. In traditional computer graphics, both in classic rasterizers and Whitted[23] style ray tracers, only light coming directly from a light source is considered when computing the shading of a point. However, this is not enough. For example, it's not completely dark under a table with a lamp hanging above it, even when there are no other light sources, since light bounces around in a room. This is called the *secondary* or *indirect* component of lighting. Often, it's enough to consider one bounce, since a lot of energy is lost in each one. Some well-known global illumination algorithms, each with advantages and drawbacks, are radiosity, photon mapping and final gather.

It turns out that Turtle's existing final gather code handles global illumination of hair pretty well without any hair-specific code or parameter tweaking.

It is quite slow, though, as is to be expected with all the detail. Also, the hemisphere sampling, currently biasing directions near the normal as more imporant, is wrong for hair. There was not enough time to find a way to integrate a different final gather sampling method.

2.2 Problems

2.2.1 Geometry aliasing

As mentioned in Shading Model above, hair is composed of thousands of small primitives. Any high detail geometry is prone to geometry aliasing, where the point sampling that ray tracing results in will often miss primitives because the distance between two neighbour rays may be larger than the entire width of a strand of hair, as shown in figure2.3. Adaptive supersampling can help to reduce, but not completely work around, the problem.

There are other possible ways to work around this problem. Pixar call their method Stochastic Pruning [17]: When the primitives are small enough onscreen, replace every N of them with one single primitive, upscaled N times. This is also the solution that was tried. Pixar also proposes reducing the contrast of the primitive colors, the motivation for this can be found in their paper. This worked well for their bush models with highly varying shades of green, but the shades of the hairs generated by Maya tend to be more uniform, and I'm not sure how much of a quality benefit this would produce. There is never enough time, and other features were prioritized.

2.2.2 The LOD catch-22

Unfortunately, while generating multiple Levels Of Detail (LODs) of a model and using different LODs when tracing rays of different importance (as defined through ray differentials, see appendix A) isn't a particularly difficult concept, it is not very compatible with most ray tracing acceleration structures. Turtle's "Small Scene" mode, which is the fastest, is simply a bunch of nested regular grids, into which all of the scene geometry is inserted.

In standard scanline rendering (rasterization), the usual method is to compute the distance between the object and the camera, and to simply use that to decide which precomputed detail-reduced model of the object to draw. In ray tracing, this doesn't work very well since there are many paths a ray can take: it can come "from" the camera, it could be reflected off a mirror, it could be refracted through a dielectric material like glass, for example. Therefore, there's no single level of detail we can assign to an object, so we have to choose a level of detail for each object for every ray.

There lies the catch-22: We don't know how far away along the ray the object is until we have already shot the ray, which may even miss the appropriate level of detail model! There are two simple ways around this problem: 1: Putting a bounding volume around each object and 2: shoot the ray incrementally in steps through space, with step lengths adjusted so that you should go down one level in the LOD hierarchy every time you restart the ray from the last end location. These methods are sketched in figure 2.4. Because Turtle's acceleration structure works the way it works, method 2 was chosen.

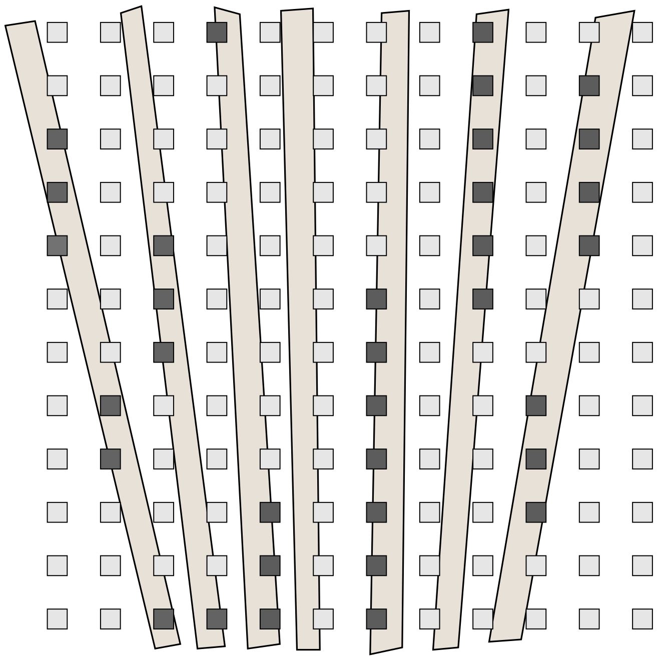


Figure 2.3: With standard regular grid sampling, rays are infinitely thin and only fired through pixel centers, so entire objects can be missed completely. Note that the pattern of dark dots does not resemble the underlying geometry at all. This phenomenon, as predicted by the Nyquist-Shannon sampling theorem, is called aliasing, and is caused by the underlying data containing frequencies exceeding half the sample rate (in this case, the ray density).

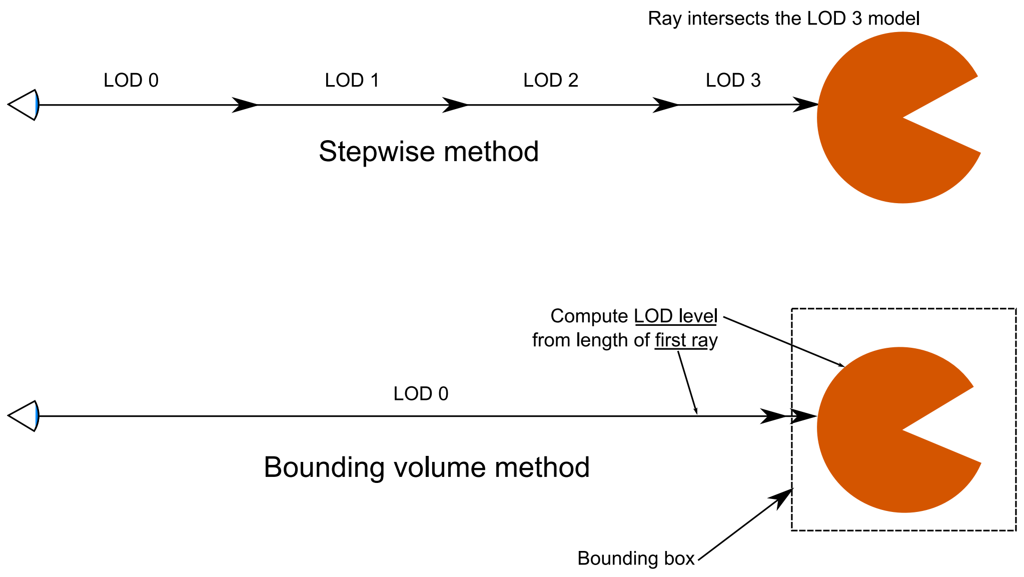


Figure 2.4: Stepwise ray shooting vs bounding volume ray shooting

A first attempt

The first approach was to build eight grids of the scene, each with lower LOD than the previous. This alone still doesn't help as we need a method to choose which grid to shoot each ray in. Fortunately, Turtle supports the concept of ray differentials, and by using those it's possible to compute how "thick" a ray is at a certain distance. When shooting rays in Turtle, it's also possible to limit the length of the ray.

Combining these two features, we can do the following:

1. Set current grid to the top grid
2. Compute how far a ray will have to go to be as thick as a hair in the current grid
3. Trace a ray, up to this distance. If something was hit, go to step 6
4. Set current grid to the next grid
5. Go to step 2
6. Return intersection details (distance, surface normal, etc)

The obvious problem with this solution is that you can't expect "correct" LOD selection when you have several different hair systems, each with a different hair thickness, in the same scene. This problem is basically unavoidable when using a straight-forward nested grid, like Turtle does in the faster small-scene mode.

Figure 2.5 shows a color visualization of the switching of LOD levels.

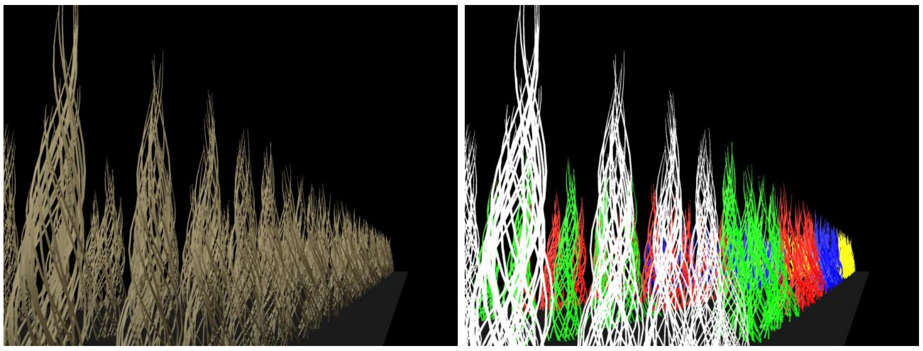

(a) Normal rendering (b) LOD color visualization

Figure 2.5: LOD level switching by distance. Note that the green hairs are just as thick as the yellow hairs in image space, but there are more of them. Without the colorization, it's quite difficult to spot this.

2.2.3 Pruning

Now that we have a working, although costly, system for storing and ray tracing different levels of detail of the same scene, we have to decide when it's time to switch to a smaller level of detail. An intuitive measure would be to switch level of detail when the width of a hair goes under about a pixel. This is also the result we will get if we switch to a lower level of detail (with thicker hairs) when the ray grows thicker than a hair at the current level of detail. See figure 2.6.

To generate a LOD using the pruning method, the model now has to be decimated. One simple approach is to double the hair width for each LOD, and also halve the hair count by randomly removing hairs. This means that since we use 8 levels, $2^8 = 256$ hairs will be replaced with a single, massively thick one at the lowest level. This will generally only happen when viewing a head from a large distance, shrinking it into a few pixels in screen space, where any detail is useless anyway. One could also reduce the number of linear segments each hair is made up of in each lod level. This hasn't been implemented.

On the other hand, in many scenes such a large range of possible pruning amount may not be wanted or necessary, so a softer slope could be used, making the transitions less noticable. We could name the proposed approach the $d = 2.0$ method, and call d the decimation coefficient, for example. $d = 1.5$ would be possible, and is the default value for this parameter in the implementation.

Also, it's not necessarily correct to replace say 5 hairs with a single 5 times thicker one. The 5 hairs will often overlap, thus reducing the screen space area covered. Replacing them with a single thick one may therefore increase the perceived density of the hair, and decrease the amount of light that passes through cracks. This effect is visible in figure 2.6. A potential solution would be to add a configurable option, specifying a percentage (such as 80%, for a 4 times thicker hair instead of 5 times) of ”thickness compensation”.

The pruning of hair can be viewed as a generalization of the multiple representations proposed in [13]. It shares many of the same issues, like visible transitions between levels of detail. However, due to the way the LOD switch happens at an exact distance for each ray sample rendered, not on the model

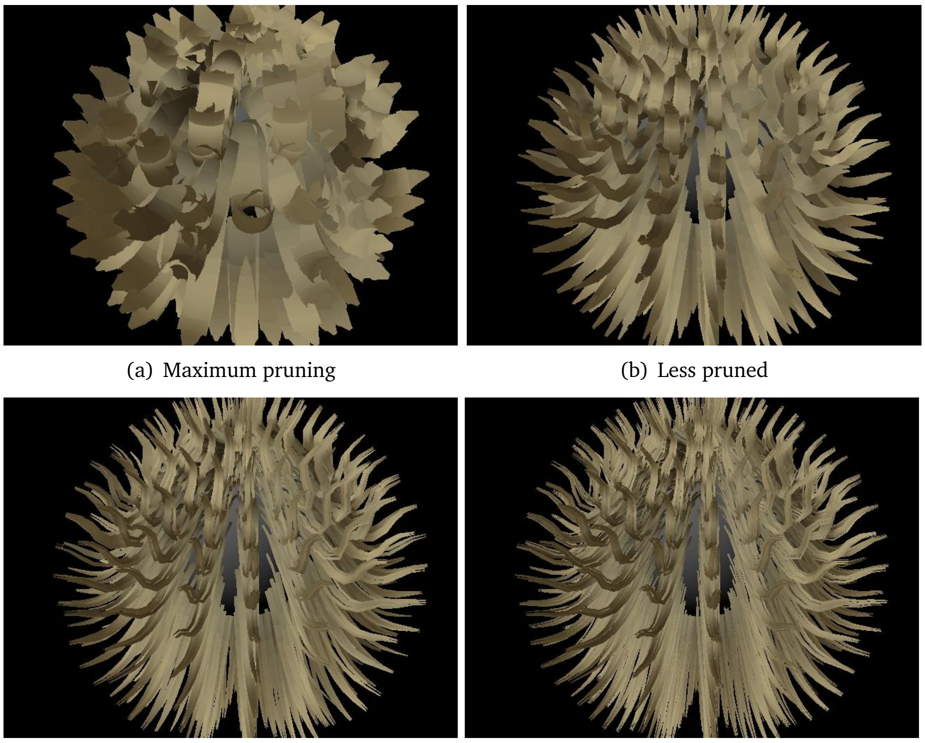

(a) Maximum pruning
(b) Less pruned
(c) Even less pruned
(d) Full detail

Figure 2.6: Different pruned LOD levels, rendered at the same scale. At a distance, they look almost identical, with the most pruned one the least aliased.

as a whole as happens in a rasterizer, the transitions in Turtle can be nearly invisible. Jittering or blending, along with shooting more rays in the transition areas, could be used to further reduce the visibility of the phenomenon.

Making it reasonably viable

Having to build 8 full grids of the entire scene, even though the only difference is the hair detail, is prohibitely expensive. A solution that makes the cost less outrageous is to keep hair in a separate chain of grids, and shoot rays both into the hair grid chain and the normal scene grid, and always select the closest hit. Although there is a performance cost, this method works well and was implemented.

Turtle has another rendering mode called Large-Scene, where every object is enclosed in its own grid. In the future, the hair and fur code could be extended to support this system, which would also solve the problem that you can really only have one hair thickness in each scene act correctly together with the LOD scheme, because all hair systems would have their own set of LOD grids.

Other proposed acceleration structures

A group at the University of Texas has developed [6] a lazily[2] built multiresolution *kD-tree* [3] acceleration structure. However, it's dependent on the scene geometry consisting mostly of displacement-mapped lowpoly models, which is certainly not true for any sort of majority of scenes. Something similar could be implemented into Turtle though, in another project.

An alternate system, known as *R-LOD* [20], has been proposed. They reduce complex triangle meshes to node-filling planes in a kD-tree. This work is very interesting for triangle mesh LOD, but unfortunately hair can't really be approximated very well using only planes. At a distance it is not inconceivable, but actually performing the reduction would not be easy, and would only work somewhat decently for dense, uniform hair. Curly hair would completely break such a system.

A standard kD-tree could also be tried, to see how well it would compete with Turtle's grids. However, there was not enough time to implement and benchmark this.

2.2.4 Memory optimization

Generating eight (or some other arbitrary number, but eight was used in this paper) full hierarchical grids of the scene greatly increases the memory requirements. To ease the requirements, hair and fur was split out into a separate array of grids. This means that the non-hair geometry of the scene does not need to be duplicated, which in most cases is a big memory saving, at the cost of some additional complexity when shooting rays.

2.2.5 Implementation problems

First of all, there are the obvious problems of coming completely new to a massive code base like Turtle, and not knowing where to start. Fortunately, the Illuminate Labs employees were very helpful, so getting started wasn't very hard after all.

At the beginning of the project, Turtle only supported triangles and displacement mapped triangles as geometry primitives. Support for adding other primitives was there in theory, but there were several instances of special-cased code that assumed that everything was either a Triangle or a DisplacedTriangle. A number of these assumptions had to be removed through fixes to code, before the PaintEffectSegment primitive (which was used for both fur and hair) could be added.

The mechanics of Maya's fur generation had to be manually reverse engineered by tweaking the fur parameters, rendering, seeing if it looked similar to the same fur as rendered by Maya Software, and if it didn't - speculate about what could be wrong, change the code, recompile and try again. Rinse and repeat. This method took a lot of time and effort, but succeeded in approximating the behaviour of most of the fur parameters.

[2]in the sense of deferring computations to exactly when they are needed, useful when it is known that all results will not be used

Disassembling and analyzing the Maya Fur plugin binary is another possible approach, but doing that would be extremely time consuming due to its large size, and likely illegal in many countries.

The basics of the MEL scripting language [7] had to be studied to be able to integrate some UI for configuring the hair rendering into Maya. Using previously existing code for other Turtle parameters as a guide, it wasn't a very difficult undertaking.

2.2.6 Theoretical performance

Pure first-hit-only ray tracing with good acceleration structures is generally considered to be $O(m \log n)$, where M is the number of pixels in the output image, and N is the number of primitives in the scene, but tracing the primary rays isn't the entire job.

In the initialization pass, the geometry data has to be generated from the hair data we get from Maya, and inserted into acceleration structures. The complexity of preparing the actual geometry is $O(n)$ to the number of individual hair segments. The complexity of inserting things into Turtle grids isn't easily estimated, although theoretically it should amortize out to something like $O(n \log n)$.

So, in conclusion, we get $O(n) + O(n \log n) + O(m \log n) = O((n + m) \log n)$. However, the very different constant factors of the various stages, and the fact that scenes may make use of global illumination and not just first-hit rendering, makes this estimation rather useless.

2.2.7 An antialiasing hack

Because of its very high geometric complexity, hair looks fairly aliased (see figure 2.3) if simply raytraced without supersampling. LOD cannot alone fix this, as hairs may thin along their lengths, and the LOD level for the hairs are chosen based on the average width of the hair. One solution is to keep the hairs their full width, even if Maya says that they should thin out, but instead have the thinning affect transparency. This removes this source of aliasing, at a heavy performance cost - transparency is very expensive in Turtle, since it always results in another ray being fired.

In conclusion, this hack was an interesting experiment, but most of the time Turtle's generic adaptive supersampling delivers equally good antialiasing, that works for everything (not just hair/fur).

2.2.8 An idea for faster transparency

Transparency of hair segments is not only useful for the antialiasing hack described in 2.2.7, it can also contribute to a softer look of dense fur, applied in moderation. If more time was available, one could imagine a faster scheme to do transparency without refraction:

- When finding a transparent intersection, add that to a list of partial intersections (store intersection point, normal/tangent, contribution weight (from cumulative inverse transparency so far) and whatever else is necessary) and keep walking the acceleration structure, until the ray hits an opaque object.

- Sort the list by depth (or assume that hits will arrive roughly in depth order)
- Throw away all but the N closest hits
- Reweight them to a sum of 1
- Shade them, getting a list of colors
- Use the weights to blend these colors together into the final color

Since this would avoid restarting the ray all the time, which has considerable overhead in Turtle, it would likely be much faster than casting new rays at every transparent hit.

Chapter 3

Results, conclusions and future work

3.1 Results

3.1.1 Comparisons with Mental Ray and Maya Software

In some corner cases, Turtle augmented with the new hair and fur code was able to outperform both Maya Software and Mental Ray. Most of the time, however, Maya Software would be ahead, Turtle second and Mental Ray last, set at comparable antialiasing quality. Maya Software has a huge advantage in that it does not perform ray tracing, but simply rasterizes the hairs. It appears that Mental Ray has some of this advantage too, as it does not appear to render fur or hair in reflections! However, fur and hair does seem to affect global illumination (although hair is not lit correctly by it), so the fur rendering of Mental Ray could be a weird hybrid of some sort.

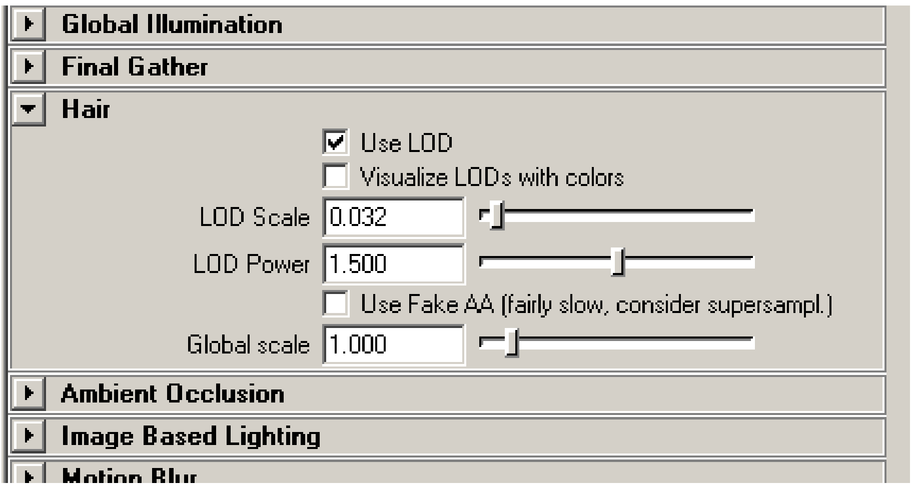
In the comparisons, Turtle has an unfair advantage over Mental Ray when rendering fur - it wasn't possible to implement transparency efficiently in Turtle (see 2.2.7), so that effect is normally simply skipped, while Mental Ray implements it with reasonable quality. This is also some of the reason that Mental Ray's fur has a somewhat softer look than that of Turtle.

All in all, it's very hard to compare performance numbers of the three renderers since they all implement different feature sets with different quality tradeoffs.

It bears repeating that that the main focus of this work was not to create innovative new hair rendering techniques, but to find reasonable ways to render Maya Hair and Fur in Turtle,

3.1.2 The Gained Tradeoff of LOD

Ultimately, what the LOD feature really buys us is the ability to choose between noisy but fully detailed renderings of hairs at a distance, and less noisy but somewhat less detailed images. Noisy renderings can sometimes be perfectly acceptable for still images, but when rendering animations it's essential to keep



✓ Use LOD
☐ Visualize LODs with colors
LOD Scale 0.032
LOD Power 1.500
☐ Use Fake AA (fairly slow, consider supersampl.)
Global scale 1.000

Figure 3.1: Maya UI customization for hair

noise to a minimum, to maximize coherence between subsequent frames. Sampling noise is ugly and makes compression of the resulting video file harder.

One could imagine that LOD would really speed things up, but due to the extra complexity of shooting and reshooting a ray along its path, render times using LOD were not generally improved in the test runs.

3.1.3 General applicability

The LOD method has not been integrated into mainline Turtle. The reason for this is that the changes necessary to the core are too risky and disruptive to be introduced without detailed planning, and that it's too specialized. It is possible that Turtle will be extended to support a more generic level of detail system in the future.

The approach studied is not only useful for a Maya renderer, but could be used by renderers for any other 3D platform. Performance numbers are naturally not general, because other renderers may use other acceleration structures, may have a more or less efficient implementation, and may be running on other computer architectures with different performance characteristics, such as possible future ray tracing hardware similar to the [21].

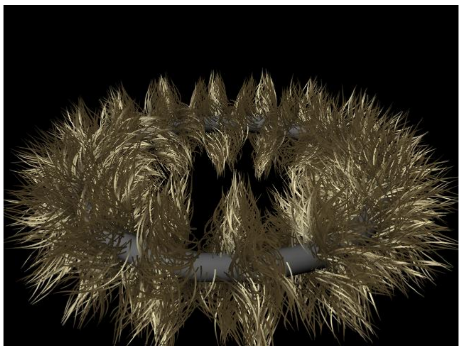
The lighting model itself is applicable to realtime rendering on modern programmable hardware, although actually ray tracing the hairs isn't really practical. Some other method of rendering them, such as camera-aligned triangle strips, would be necessary. These would be prone to the usual aliasing problems, although this can be mitigated by clever use of texturing and alpha blending.

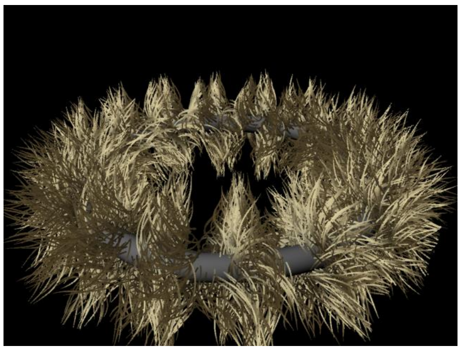
3.1.4 Maya integration

Using Maya's MEL scripting facility, some new panels and sliders were added to the Turtle interface in Maya to control the hair rendering system (figure 3.1).

3.2 Performance

All performance statistics are from a single-core Athlon 64 3000+, running 32-bit Windows XP and Maya 7. Keep Turtle's inherent advantage mentioned in 3.1.1 in mind when evaluating the fur rendering numbers.


(a) Maya SW (scanline renderer): 4s


(b) Turtle (raytraced): 19s

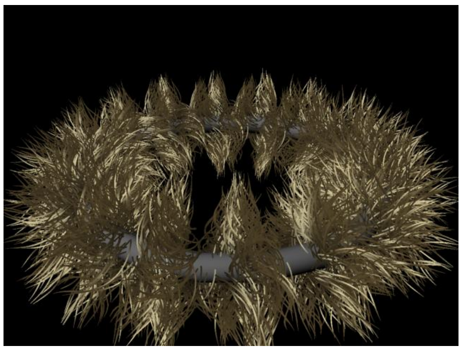

(c) Mental Ray (raytraced): 67s

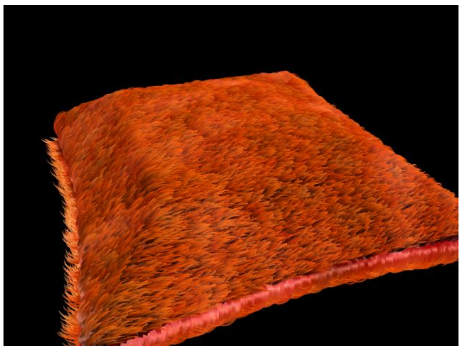
Figure 3.2: Hairy torus (Maya Hair)

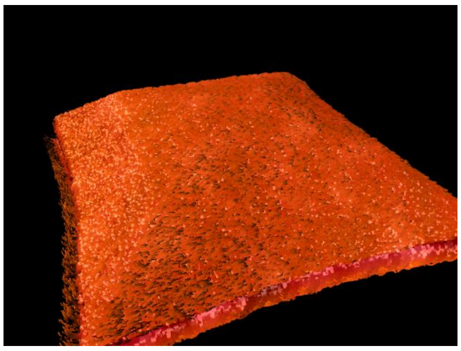
3.2.1 Hairy torus (figure 3.2)

Renderer	Sampling	Time (s)
Maya SW (scanline)	N/A	4
Turtle (raytraced)	2x adaptive, edge tracing	19
Mental Ray (raytraced)	1x to 4x (adaptive)	67

3.2.2 Pillow scene (figure 3.3)

Renderer	Sampling	Time (s)
Maya SW (scanline)	N/A	8
Turtle (raytraced)	2x adaptive, edge tracing	20
Mental Ray (raytraced)	1x to 4x (adaptive)	56


(a) Maya SW (scanline renderer): 8s


(b) Turtle (raytraced): 20s (The white sparkles are specular highlights. These need another viewing angle to be visible in the other two renderers)

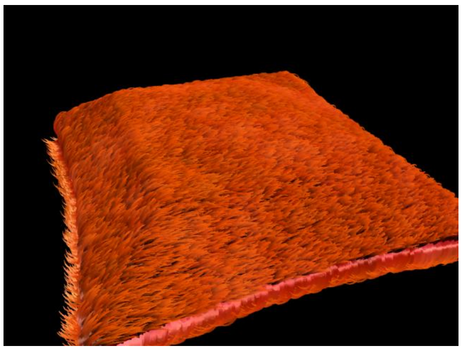

(c) Mental Ray (raytraced): 56s

Figure 3.3: Pillow with fur (Maya Fur)

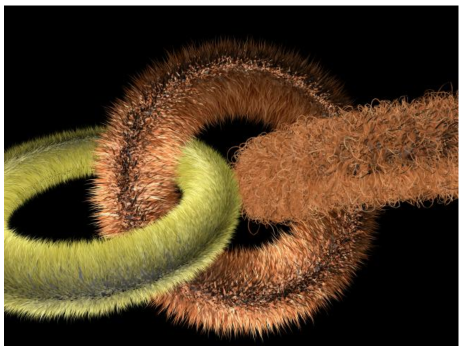
3.2.3 The Rings of Fur (figure 3.4)

Renderer	Sampling	Time (s)
Maya SW (scanline)	N/A	14
Turtle (raytraced)	2x adaptive, edge tracing	39
Mental Ray (raytraced)	1x to 4x (adaptive)	144

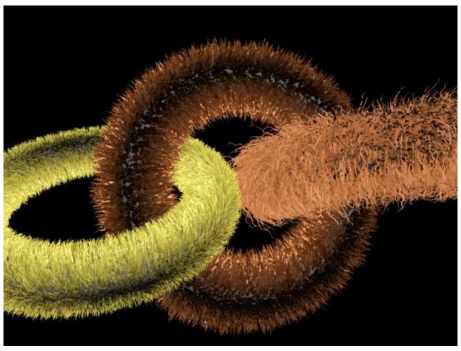
3.3 Conclusions

Maya Hair and Maya Fur can now be rendered decently with Turtle, with a few missing features and somewhat different lighting. Large amounts of fur still consume way too much memory, although there is lots of room for reducing the memory consumption.

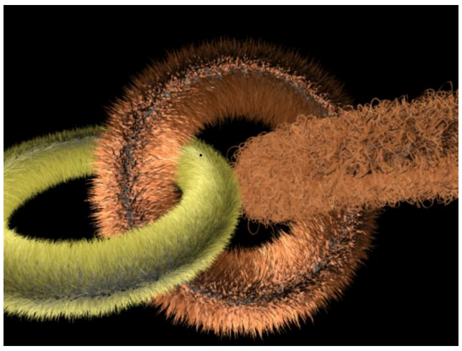
There are still many possible improvements left to make, as outlined in 3.4.



(a) Maya SW (scanline renderer): 14s



(b) Turtle (raytraced): 39s



(c) Mental Ray (raytraced): 144s

Figure 3.4: The Rings of Fur (Maya Fur). From left to right: Duckling, Squirrel, Llama fur presets

3.4 Future Work

As mentioned in 2.1.7, memory usage could be greatly reduced, with a relatively small amount of work.

The pruning algorithm currently only removes and thickens hair. It would also be possible to reduce the number of segments that each hair is made of for each LOD level.

Generating all the eight grids for LOD is costly in terms of time, and could perhaps be done lazily. Instead of creating new grids in voxels with too many triangles, it would be possible to just postpone doing it until the first time the voxel is intersected with a ray. One would have to be careful with synchronization, and use a mutex or other similar mechanism, because since rendering is fully multithreaded, another ray tracing thread could wander into the voxel while the new subgrid is being generated, and decide to generate the subgrid itself, causing all kinds of mayhem.

Also, in many scenes, such as with a single hairy head at some distance from the camera, only one or two LOD levels are used. In these cases, it's a waste to generate those grids at all, so lazy evaluation is applicable there.

The interaction of global illumination with the hair and fur works and looks OK, but the final gather sampling isn't correct.

The missing features mentioned in 2.1.5 could be implemented.

The way to do transparency can be discussed. LeBlanc et al[1] notes that layer upon layer of transparent hair will "saturate" and not let any light through, except possibly very strong HDR style lights. (The rest of their paper details a conventional rasterization approach).

The shading model is still substantially different to the Maya one. It can be argued which one is more correct, but Maya's fur shading model usually looks better and more varied. On the other hand, it appears to approximate all kinds of lights with standard directional lights. For the hair shading model, the differences mostly manifest themselves as differently positioned highlights. A more advanced direct shading model, like the one described in [19], could also be implemented. Alternatively, a more expensive path tracing method such as [15], simulating the multiple scattering of light in hair, would provide yet more realistic results, at high performance and complexity costs.

A softer look could be achieved with transparency, by using the method described in 2.2.8.

Shadowing is a problem that hasn't been investigated very deeply. Using raytraced shadows works, but they will be hard (sharp), which doesn't look very good on hair, which in reality transmits quite a lot of light and generally casts fairly fuzzy shadows. Many other renderers use things like deep shadow maps instead. There are also some possible approaches based on voxel grids that track the hair density in regions of space, such as [16], and a similar method in [4], although some rather large architectural changes to Turtle may be necessary to integrate such techniques.

Recently, integrating some simple texture space ray tracing into the pixel processing of rasterizers has become a popular method to add fake displacement mapping to textures in games. It's possible that Kajiyas old [12] algorithm could be resurrected to run on the newest graphics cards.

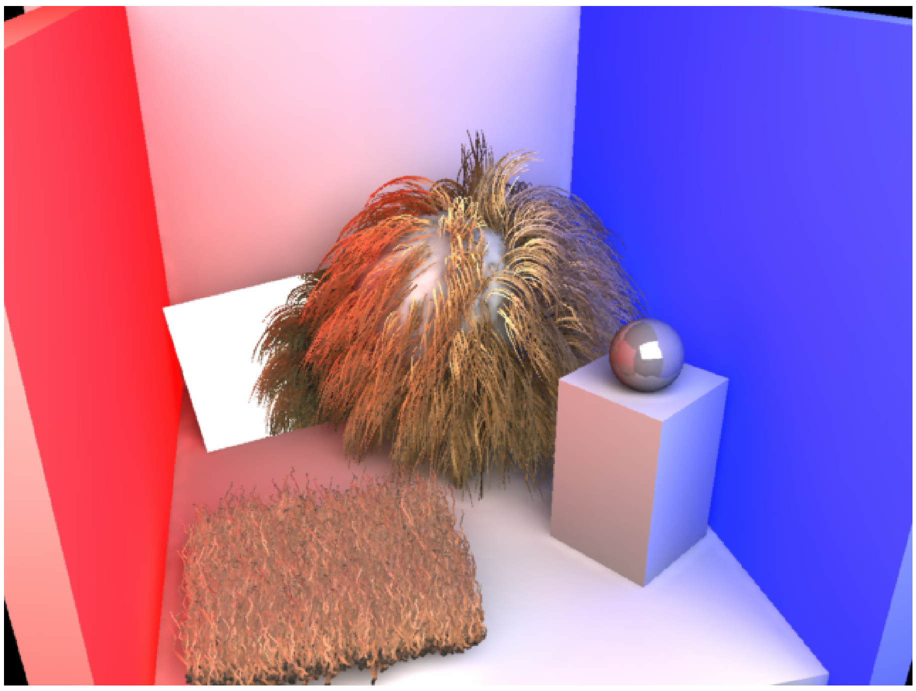


Figure 3.5: Hair, a mirror, and fur rendered with global illumination (final gather). Render time: 30 min.

Bibliography

[1] Daniel Thalmann André M. LeBlanc, Russel Turner. *Rendering Hair using Pixel Blending and Shadow Buffers*. Computer Graphics Laboratory, Swiss Federal Institute of Technology, 1991.

[2] Philipp Slusallek Andreas Dietrich, Ingo Wald. Interactive visualization of exceptionally complex industrial cad datasets. In *Proceedings of ACM SIGGRAPH 2004*, 2004.

[3] Jon Louis Bentley. *Multidimensional binary search trees used for associative searching*. ACM Press, 1975.

[4] Marie-Paule Cani Florence Bertails, Clément Ménier. *A Practical Self-Shadowing Algorithm for Interactive Hair Animation*. GRAVIR - IMAG/INRIA, Grenoble, France, 2005.

[5] Dan B Goldman. Fake fur rendering. `http://www.cs.washington.edu/homes/dgoldman/fakefur/`, 1997.

[6] Peter Djeu Rui Wang Ikrima Elhassan Gordon Stoll, William R. Mark. Razor: An architecture for dynamic multiresolution ray tracing. Technical Report 06, 2006.

[7] David A. D. Gould. *Complete Maya Programming*. Morgan Kaufmann Publishers/Elsevier Science, 2003.

[8] Homan Igehy. Tracing ray differentials. In *Proceedings of ACM SIGGRAPH 1999*, 1999.

[9] A. Finkelstein; H. Hoppe. J. Lengyel, E. Praun. *Real-Time Fur over Arbitrary Surfaces.* ACM Press, ACM Symposium on Interactive 3D Graphics 2001, 227-232., 2001.

[10] Jr Joseph M. Cychosz, Warren N. Waggenspack. *Intersecting a ray with a cylinder (Graphics Gems IV)*. Academic Press, 1994.

[11] James Kajiya. Anisotropic reflection models. In *Proceedings of ACM SIGGRAPH 1985*, 1985.

[12] Timothy L. Kay Kajiya, James T. Rendering fur with three dimensional textures. In *Proceedings of ACM SIGGRAPH 89*. Addison Wesley, 1989.

[13] Joohi Lee. Susan Fisher. Dean Macri. Kelly Ward, Ming C. Lin. Modeling hair using level-of-detail representations. In *Computer Animation and Social Agents*, 2003.

[14] Hans-Peter Seidel Martin Koster, Jörg Haber. *Real-Time Rendering of Human Hair using Programmable Graphics Hardware*. MPI Informatik, Saarbrücken, Germany.

[15] Jonathan T. Moon and Stephen R. Marschner. Simulating multiple scattering in hair using a photon mapping approach. In *Proceedings of ACM SIGGRAPH 2006*, 2006.

[16] N Magnenat-Thalmann R Gupta. Scattering-based interactive hair rendering. `http://www.miralab.unige.ch/papers/376.pdf`, 2005.

[17] John Halstead Robert L. Cook. *Stochastic Pruning*. Pixar Animation Studios, 2006.

[18] Thorsten Scheuermann. *Hair Rendering and Shading (GDC presentation)*. ATI Research, Inc, 2004.

[19] Mike Cammarano Steve Worley Stephen R Marshner, Henrik Wann Jensen and Pat Hanrahan. Light scattering from human hair fibers. `http://graphics.stanford.edu/papers/hair/`, 2003.

[20] Dinesh Manocha Sung-Eui Yoon, Christian Lauterbach. R-lods: Fast lod-based ray tracing of massive models. `http://gamma.cs.unc.edu/RAY/`, 2006.

[21] Jörg Schmittler Sven Woop and Philipp Slusallek. Rpu: A programmable ray processing unit for realtime ray tracing. In *Proceedings of ACM SIGGRAPH 2005*, July 2005.

[22] George Borshukov Tadao Mihashi, Christina Tempelaar-Lietz. *Generating Realistic Human Hair for The Matrix Reloaded*. ESC Entertainment.

[23] T. Whitted. An improved illumination model for shaded display. In *Communications of the ACM*, June 1980.

Appendix A

Ray differentials

Ray differentials [8] (figure A.1) is a way to define a measure of ”thickness” of a ray. A ray differential is simply a measure of how different a ray’s origin and direction vectors are to those of its neighbours. For example, reflection rays generated by curved surfaces tend to have large ray direction differentials.

Ray direction differentials can be used as an estimate of the quality or sharpness of rendering needed for a pixel, since most of the time rays with large ray direction differentials will not be individually highly influental to the image. Primary rays (those from the camera) all have small ray differentials. Also, when sampling textures, ray position differentials are useful for shaping the texture filter.

Turtle uses the concept of ray differentials throughout, and tracks them for each ray.

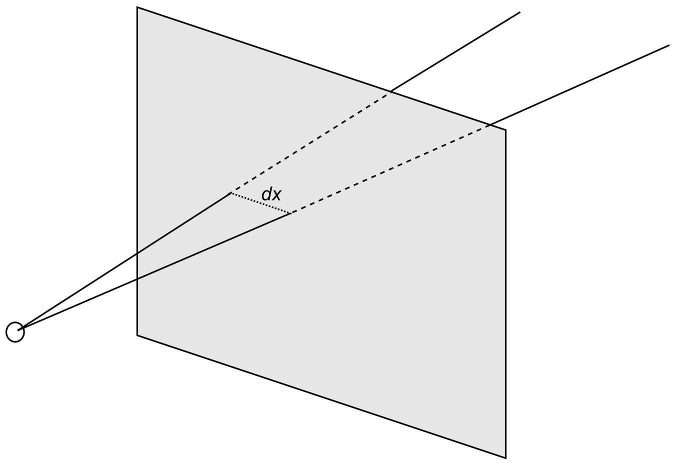


Figure A.1: Primary rays crossing the image plane, with the ray x direction differential drawn