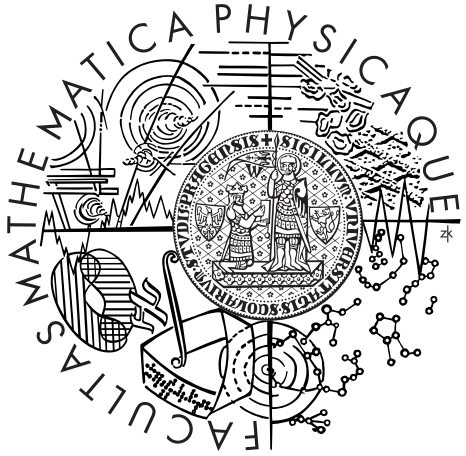
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Martin Šik

Generování vlasů interpolací

Department of Software and Computer Science Education

Supervisor of the master thesis:	Jaroslav Křivánek
Study programme:	Informatics
Specialization:	Software Systems, Computer Graphics

Prague 2012

I would hereby like to thank my supervisor Jaroslav Křivánek for his supervision and counseling which guided me throughout my work. I would also like to express my thanks to all members of the Stubble software project which includes the hair interpolation described by my diploma thesis.

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: Generování vlasů interpolací
Autor: Martin Šik
Katedra: Kabinet software a výuky informatiky
Vedoucí diplomové práce: Ing. Jaroslav Křivánek, Ph.D., Kabinet software a výuky informatiky
Abstrakt: Tato diplomová práce popisuje procedurální generátor vlasů, který je schopen vygenerovat vlasy z pouze pár řídících vlasů, které jsou přímo modelovány 3d umělcem. Generátor vlasů je součástí projektu Stubble – nástroje na modelování vlasů v Autodesk Maya. Procedurální generátor vlasů umožnuje generování vlasů během vykreslování, a tudíž není potřeba ukládat vlasy do souboru se scénou, což výrazně zrychlí vykreslování. Vlasy mohou být taktéž generovány interaktivně a zobrazeny pomocí OpenGL během modelování v Maye. Generované vlasy jsou hlavně spočteny pomocí interpolace z již zmíněných řídících vlasů, ale zároveň jsou ovlivněny mnoha nastavitelnými vlastnostmi. Tyto vlastnosti umožnují změnit geometrii vlasů pomocí šumových funkcí, definovat barvu a tloušťku vlasů a mnohem více. Abych určil pozice vlasů na dané trojúhelníkové síti, používám vlastní vzorkovací algoritmus, který generuje náhodné vzorky na trojúhelníkové síti dle hustoty dané 2-dimenzionální texturou. Můj vzorkovací algoritmus používá novou techniku ke generování vzorků z diskrétní distribuce. Tato technika může být použita v jiných aplikacích než je vzorkování trojúhelníkových sítí.
Klíčová slova: počítačová grafika, 3D, vlasy, modelování

Title: Guide hair interpolation
Author: Martin Šik
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 procedural hair generator that is able to generate hair from just a few hairs, called hair guides, which are directly modeled by a 3d artist. The procedural hair generator is a part of Stubble project – a tool for hair modeling in Autodesk Maya. The procedural hair generator can generate hair during rendering, thus avoiding storage of hair geometry in a scene file, which makes the rendering process very efficient. Furthermore, hair can be generated interactively and displayed by OpenGL during modeling in Maya. Generated hair geometry is mainly defined by interpolation from the mentioned hair guides; however it is also influenced by many hair properties. These properties can change hair geometry using noise functions, define hair color, width and more. To determine hair root positions on a given triangular mesh I use my own mesh sampling algorithm that generates random samples on a triangular mesh according to a density defined by a 2-dimensional texture. My sampling algorithm uses an innovative way of sampling from a discrete probability distribution, which can be used in other applications than mesh sampling.
Keywords: computer graphics, 3D, hair, modeling

Contents

Introduction

This thesis describes a procedural hair generator, which is a part of the *Stubble* project – a tool for hair modeling in Autodesk Maya[1]. Stubble was developed as a software project at Faculty of Mathematics and Physics, Charles University in Prague, under the supervision of Ing. Jaroslav Křivánek, Ph.D., and has been successfully defended in January 2012. Since creating a procedural hair generator is a complex task, my contribution to the *Stubble* project was from the beginning considered as a diploma thesis and therefore it covers both the software project and the diploma thesis.

Stubble project

The *Stubble* project's main goal was to reimplement a commercial tool for hair modeling *Shave and a Haircut*[2]. This goal was given to us (team members of *Stubble* project) by UPP[3], which is a company focused on visual effect creation and movie post production. They have used *Shave and a Haircut* for creating hairs on animal and human bodies for CG animated scenes, however they were unsatisfied with it in several ways, especially in the procedural hair generation part of *Shave and a Haircut*. See Figure 1 for a demonstration of a CG animated character with hairs/fur.

Figure 1: An example of an animated film using computer generated fur. Image Copyright © 20th Century Fox.

The most important aspect of *Stubble* which is similar to *Shave and a Haircut* is that final hairs (I will refer to them in the following text as interpolated or

[1] `http://usa.autodesk.com/maya/`
[2] `http://www.joealter.com/`
[3] `http://www.upp.cz`

generated hair) are never directly modeled by the user, instead they are interpolated from hair guides (see Figure 2). These hair guides are few hairs (from a hundred to several thousands) that can be directly modeled by the user in Autodesk Maya using specialized tools such as a tool for brushing or a tool for hair cutting. Modeling the final hair instead of the hair guides would be nearly impossible or at least very time consuming and time means money for companies such as UPP. It is also important to mention that both the generated hair and the hair guides must grow from a model surface defined by a triangular mesh. This surface represents a human or an animal skin.

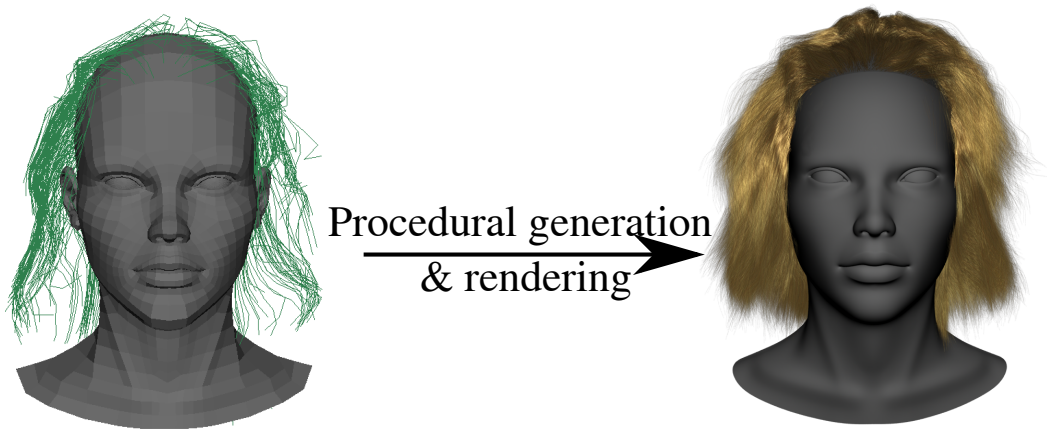


Figure 2: From the hair guides through interpolation and procedural generation to the final image rendered by the 3Delight RenderMan.

One key difference between *Shave and a Haircut* and *Stubble* is that in *Shave and a Haircut* the hair guides are always placed at each vertex of the triangular mesh on which hair grows, therefore the number of the hair guides is determined by the triangular mesh. This is a severe limitation from the user's point of view, since she/he cannot for example use a high polygon count mesh with a low number of hair guides. On the other hand, if the hair guides are only allowed to grow from the mesh vertices, hair interpolation is much easier, since every hair can be interpolated from three hair guides that lie at vertices of the triangle from which interpolated hair grows. Since UPP wanted to select the number of the hair guides independently of the resolution of the skin mesh, *Stubble* places the hair guides in a different way. Furthermore, UPP desired that both the interpolated hair and the hair guides are distributed on a model according to a given density texture. This affects the design of my procedural hair generator in several ways, as will be discussed later.

Another important advantage of *Stubble* is that it can generate the interpolated hair during rendering without the need to store hair geometry as *Shave and a Haircut* does. This is mainly important from a performance point of view, as I will discuss later in my thesis.

For more information about the *Stubble* project please refer to its documentation (it can be found on the accompanying CD, see Chapter A).

Thesis goals

As I have already mentioned, the goal of this thesis is to create a procedural hair generator. In order to achieve this goal, I have to accomplish the following important subtasks.

Hair root placement. I have already mentioned that both the generated hair and the hair guides are distributed on a model surface with a density proportional to a 2-dimensional texture (later referred to as a density texture). One way to approach the hair root placement is to use a sampling algorithm of a triangular mesh (model). Since no sampling algorithm existed that was sufficiently fast for my purposes or could support a non-uniform distribution of samples (i.e. a distribution defined by a density texture), I have developed a novel fast random sampling algorithm for triangular meshes. Furthermore, my sampling algorithm uses an innovative way to sample from a discrete probability distribution, which can be used in other applications than mesh sampling.

Hair representation. Before I can make any steps in procedural hair generation, I need to determine how to represent the generated hair. Since I have already discussed the hair root placement, it is obvious that I will represent each hair fiber individually (and not for example only a surface of all hair combined as a triangular mesh). However, I still have a number of options how to represent a hair fiber.

Hair interpolation. A key feature of a procedural hair generator is interpolation of hair from the hair guides. Since both the hair guides and the generated hair are arbitrarily placed over a triangular mesh, I need to use a sophisticated interpolation algorithm, which gives good results and is sufficiently fast. Furthermore, the interpolation has to be consistent during an animation. The hair guides can be manually or automatically (using hair dynamics) animated and also the triangular mesh from which the hair grows can be deformed in the course of an animation. All this must be taken into account when designing the hair interpolation, so the generated hair appears to be smoothly animated.

Procedural generation of hair. The hair interpolation is only a small part of the procedural hair generation. *Shave and a Haircut* uses many hair properties to influence the generated hair. There are properties that influence hair geometry by applying noise in many different ways, or that cause the hair to be cut at a defined position. Another property defines that the hair should be generated in strands, instead of single fibers. The hair can also have varying color, opacity and width. All properties that are in *Shave and a Haircut* must also be in my procedural hair generator and they must influence the generated hair in similar manner. Achieving this is not easy, since *Shave and a Haircut* is a commercial product and therefore no information is available on how it works internally. In this thesis I will not mention how exactly can the user set these properties, since that is a part of the Stubble UI, not a part of the procedural hair generator.

Hair rendering. Of course as a final step, the generated hair must be rendered. When a scene is rendered by specialized rendering software, it is usually output to a scene file of a defined format from which it is loaded by the renderer. *Shave and a Haircut* outputs complete hair geometry to the scene file. Since there can be up to 150,000 hairs on a human head and up to millions hairs on an animal body (even billions in reality, however such high numbers are never used) and each hair fiber must be stored in the scene file, the scene file becomes exceedingly large (several gigabytes for a single frame of an animation). Handling such huge files is a bottleneck in rendering, therefore it is better to store only small amount of data to the scene file, which is enough to define the generated hair. To generate hair from the stored data during render time, a specialized library for the hair generation must be created. This library is executed by the renderer and generates hair on the fly, completely avoiding storing hair in scene files. This approach provides a significant increase in rendering performance, which is always very important. Another important thing is to write the hair generator in such a way that it can easily be extended to support any renderer. In my thesis I will however mostly aim at the RenderMan renderer.

Interactive display. Since the final hair is influenced by the hair guides and the hair properties which are both handled by the user inside Autodesk Maya, it is important to give the user at least some idea how the final hair will look. Therefore the procedural hair generator should be able to generate hair and display it using OpenGL interactively in Autodesk Maya. Of course, only a small amount of the final hair can be displayed at a sufficiently high rate.

Thesis overview

The rest of this thesis consists of the following chapters:

- **Related work**: In this chapter I describe related work in hair modeling and sampling algorithms.
- **Methods and algorithms description**: This is the most important part of my thesis. It discusses my sampling algorithm (Section 2.1), the hair interpolation and the procedural generation (Section 2.2) and also the interactive hair generation in Maya and the hair generation during rendering in RenderMan (Section 2.3).
- **Implementation**: Here I discuss the implementation details of my hair generator.
- **Results**: This chapter shows results of both my sampling algorithm (Section 4.1) and the hair generator (Section 4.2).
- **Conclusion**: The conclusion of my thesis.

1. Related work

In this chapter I describe the work related to hair modeling and mesh sampling.

1.1 Hair modeling

Hair modeling is addressed by many recent papers, since virtual hair appears in both animated and live-action movies, or even in real-time computer animation, for example in computer games. As discussed in [42] (which is along with [48] an excellent overview of hair modeling), hair modeling can be divided into three categories: hairstyling, hair simulation, and hair rendering. My thesis mainly aims at hairstyling, which is concerned with defining the shape of hair. Hairstyling can be further divided to attaching hair to skin, determining overall hair shape and finally handling subtle hair details.

There are many methods to attach hair to skin, in some approaches [22, 47] the user places hair locations on a 2d map which is then mapped by various techniques on a 3d surface. An alternative approach is to place hair uniformly over the skin [22, 8], which is close to reality for human head scalp, however untrue for hairs over the whole body. Finally, hair can be placed according to a density function, which can be for example defined by the user painting over the 3d surface [8, 17]. This is also the case of my hair generator (a density texture I use can be created in Maya by painting over the surface).

Once hair is placed, its overall shape can be determined. There are again many techniques to do that and they can be divided to three main categories: geometry-based, physically-based or image-based techniques. Geometry-based methods [46, 40] handle hair as geometric primitives and allow the user to handle these primitives directly or via proxy, therefore using the hair guides falls into this category. Physically-based approaches [47, 14] usually limit user handling of hair and base hair appearance on physical simulations. Finally, image-based techniques [43] model hair shape according to a photograph or a sketch of hair.

As the last step of hairstyling, subtle details of hair, such as curls or waves, are determined. There are many methods to generate these details, most of them [8, 47, 17] use user parameters such as magnitude or frequency that control curling of hair, however some random offsets are always used to ensure non-uniformity of hair. Alternative approaches use physically-based simulation of hair curliness [5].

All of the above mentioned works discuss mainly modeling of hair on a human head. There are also papers [21, 45, 25] focusing on modeling and rendering of animal fur, which can also be created by my hair generator. While many of the aforementioned works create more realistic hair than *Shave and a Haircut*, they are not so wide-spread, since they usually have many limitations or do not offer a sufficient freedom for a 3d artist as *Shave and a Haircut* does. In my thesis I was given the task to create a hair generator similar to *Shave and a Haircut*, therefore I focus on that task rather than trying to implement some features from recent papers on fur or hair modeling.

1.2 Mesh sampling

As I have already mentioned, I will use a mesh sampling algorithm to determine hair root positions on a given triangular mesh. Most recent papers about sampling aim at generation of point distributions with blue-noise properties (see [24, 11] for an overview), however, relatively little research has focused on random sampling of surfaces embedded in 3D space, which is what I need for determining hair root positions. Surface sampling has been studied in different contexts such as remeshing [39, 33], point-based graphics [13], texturing [23], realistic rendering [20, 7, 9, 34], or non-photorealistic rendering [26]. The goal of these techniques, however, is not to generate unbiased samples from a prescribed probability density (in my case defined by an arbitrary 2-dimensional texture). In addition, their performance is usually insufficient for usage in a fast hair generator.

Several recent papers have addressed surface sampling with stratification [27] or blue-noise properties [6, 44, 10]. However, the quality of the resulting distribution comes at the cost of relatively low sampling performance (not more than 400,000 samples per second on GPU reported by [44]). Another important limitation of these methods is that they generate uniform samples without the possibility of controlling the sample density. Bowers et al. [6] does discuss a non-uniform version of their sampling algorithm but according to their measurements, it is 18 times slower than the uniform variant (approximately 10,000 samples per 1~2 seconds on GPU). This is an important limitation for my purpose, therefore I have to turn away from sampling algorithms that generate high quality samples and more concentrate on sampling speed.

Fast unbiased sampling of meshes is discussed in [28, 10] but these techniques are limited to uniform distributions. A widely known approach to generalize such techniques to generating samples with any given density is rejection sampling [31, p. 671]. Rejection sampling is usually a very fast generator of random samples; however its performance decreases greatly for highly varying densities (for example density defined by an HDR texture). The aforementioned program *Shave and a Haircut* uses a similar technique to rejection sampling to generate hair root positions. As was told me by UPP employees, highly varying density textures are often used to define distributions of hair over a surface. In those cases, performance of hair generation done by *Shave and a Haircut* is significantly reduced.

As rejection sampling, my algorithm is able to generate samples with arbitrary density, however with roughly the same speed as uniformly distributed samples, even for highly varying textures. Furthermore, my algorithm consistently outperforms any of the existing alternatives by a large margin.

2. Methods and algorithms description

In this chapter I will first describe my sampling algorithm (Section 2.1) of triangular meshes, that I use to determine the hair root position. Next, I will talk about the procedural hair generation, which includes the hair interpolation, applying noise on the generated hair, defining hair color and more (Section 2.2). Finally, I will discuss in Section 2.3 how is the generated hair displayed interactively and rendered by RenderMan.

2.1 Sampling of triangular meshes

This section describes a new mesh sampling algorithm. I use this algorithm for determining hair root positions on a given model with hair density defined by a 2-dimensional texture as described in Section 2.2.3, however it has other applications as can be seen in Section 2.1.8.

2.1.1 Problem definition

Given a set of n triangles $\{T_i\}_{i=1}^n$, $T_i \subset \mathbb{R}^3$, $\mathcal{T} = \cup_{i=1}^n T_i$, a density function $f : [0,1]^2 \to \mathbb{R}^+$ represented by an image texture, and a mapping $m : \mathcal{T} \to [0,1]^2$ that maps the texture onto the triangles, I want to draw samples from a distribution with the probability density function (PDF) $p : \mathcal{T} \to \mathbb{R}^+$ (w.r.t. the surface area measure) given by:

$$p(\mathbf{x}) = \frac{f(m(\mathbf{x}))}{\int_{\mathcal{T}} f(m(\mathbf{x}'))\mathrm{d}A(x')} \tag{2.1}$$

See Figure 2.1 for an illustration.

2.1.2 Possible approaches

A straightforward solution to the above problem is to use *rejection sampling*:
1) Pick a triangle T_i proportionately to its surface area,
2) Propose a sample $\mathbf{x}$ from a uniform distribution on T_i,
3) Generate a random number ξ from uniform distribution $U(0, M)$, with $M =$

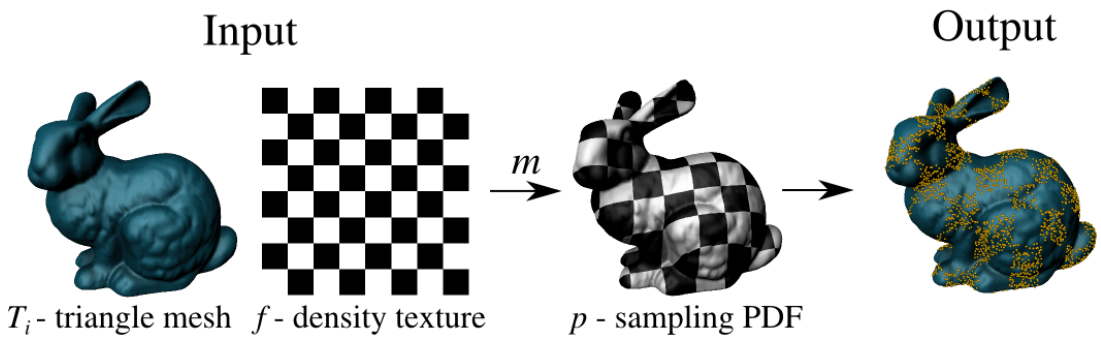


Figure 2.1: Input and output of a mesh sampling algorithm.

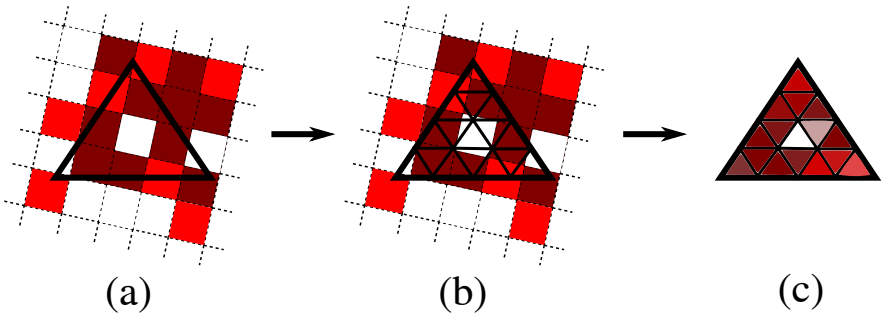


Figure 2.2: In the preprocess, I subdivide each triangle (a) until the sub-triangle size matches the density texture resolution (b), and subsequently resample the density texture on the subdivided triangle (c). The different colors represent various densities.

$\sup\{f(\mathbf{u}) \mid \mathbf{u} \in [0,1]^2\}$,
4) Accept $\mathbf{x}$ if $\xi < f(m(\mathbf{x}))$, otherwise go to step 1.
This approach suffers from the usual disadvantages of rejection sampling. Efficiency degrades rapidly for non-uniform density f, and the number of random numbers used to generate one sample cannot be bounded prior to the calculation.

Another alternative would be to carry out the sampling in the $[0,1]^2$ texture space, where the density function f is defined by an image that can be efficiently sampled [31, p. 671], and then transform the samples to the triangles using the inverse of the m mapping. However, generality of this approach is compromised by the fact that m is often not invertible (e.g. for a tiled texture).

2.1.3 Algorithm overview

My approach does not suffer from the performance and generality issues of the aforementioned alternatives. Consider a simple case, where the density texture f is constant over the area of any one triangle. Mesh sampling problem would then reduce to choosing a triangle from a suitable discrete distribution and drawing a sample from a uniform distribution on the triangle. The main idea of my approach is to map the general problem to this simplified case by subdividing the triangles of the input mesh until the density texture f can be considered constant over the sub-triangle area, and resampling the density texture on the subdivided triangles (Figure 2.2). Doing so avoids problems due to the incompatibility of the density texture and triangle mesh domains, and greatly simplifies the sampling algorithm. This idea is similar to the Mesh Colors approach for texturing 3D meshes [50], though the purpose is different.

My algorithm works in two stages. The preprocessing stage, executed once for a given mesh and density texture, subdivides the input triangles and creates a piece-wise constant PDF on the sub-triangles that can be efficiently sampled. The sampling stage then draws samples from this PDF. In order to make the approach practical, I use a memory-efficient representation for the subdivided triangles and an accelerated procedure for generating surface samples from this representation. The following subsections describe the technical details of my sampling algorithm.

2.1.4 Stage 1: Preprocessing

Pseudocode of the preprocessing stage is given in Figure 2.3. In this stage, I subdivide the input triangles and effectively replace the desired sampling PDF (Equation 2.1) by its approximation p' that is constant over the area of each sub-triangle. The PDF value for sub-triangle D_i is determined by taking a (bilinearly filtered) sample f_i of the density texture at the sub-triangle's barycenter. For $\mathbf{x} \in D_i$, the approximate PDF is defined as

$$p'(\mathbf{x}) = \frac{f_i}{\sum_{j=1}^{n_s} f_j \, |D_j|}, \tag{2.2}$$

where n_s is the total number of sub-triangles and $|D_i|$ denotes the surface area of sub-triangle D_i.

To make the above PDF approximation accurate, I need to subdivide the input triangles until the density texture can be safely considered constant inside each sub-triangle. To achieve this goal, each triangle is subdivided so that its sub-triangles are smaller than a texel of the density texture mapped on the triangle (Figure 2.2). Because the m mapping is affine over each input triangle (as per linearly interpolated per-vertex texture coordinates), the subdivision depth can be determined directly from the number of texels mapped on the triangle. Subsequently, I recursively replace sub-triangles with the same PDF value (if all 4 sub-triangle siblings has the same PDF value) by their parent, so that the total number of resulting sub-triangles is minimized.

Sampling of the piecewise constant PDF p' involves picking a sub-triangle with probability $P_{D_i} = \int_{D_i} p'(\mathbf{x}) \mathrm{d}A(\mathbf{x}) = f_i \, |D_i| / \sum_{j=1}^{n_s} f_j \, |D_j|$, generating a sample in the sub-triangle, and mapping it to the original mesh triangle. For this purpose, the preprocessing stage calculates and stores the cumulative distribution function (CDF) $F_i = \sum_{j=1}^{i} P_{D_j}$. I also need to store for each sub-triangle, a reference to the parent input triangle, as well as the barycentric coordinates of the sub-triangle inside the parent triangle. Why that is necessary will be obvious from the sampling stage description.

I take advantage of the fact that the subdivision scheme is the same for all triangles, so a unique sub-triangle index is sufficient to determine its barycentric coordinates (see Figure 2.4). A separate pre-computed table accessed by the sub-triangle index stores the actual sub-triangle barycentric coordinates. This significantly reduces memory consumption, since for each sub-triangle I only need to store its CDF value F_i and two indices (a parent triangle index and a sub-triangle index). In my implementation, I use 4 byte value for each of the three aforementioned values; therefore a sub-triangle takes up only 12 bytes in memory. During the preprocessing stage I need to efficiently determine a sub-triangle index for each sub-triangle. As can be seen in Figure 2.4, a whole triangle has a sub-triangle index 0. Index of each sub-triangle D_i which has a parent sub-triangle D_{i-1} (i.e. the sub-triangle D_i is not a whole triangle) can then be calculated as $\mathrm{index}(D_i) = 4 \cdot \mathrm{index}(D_{i-1}) + j$, where $j \in \{1, 2, 3, 4\}$ is the number of D_{i-1} sub-triangle child.

The calculation of the pre-computed data structure which stores sub-triangle barycentric coordinates is described in Figure 2.5). Before I can calculate barycentric coordinates for each sub-triangle, I need to determine the maximum subdivision depth of any triangle. This is easily done by iterating over all triangles

For each mesh triangle T:

1. Calculate the area of T. The subdivision depth i is now 0. Mark the triangle T as the sub-triangle D_0 (D_i denotes any sub-triangle in the subdivision depth i).
2. **If** the subdivision depth i is less than maximum (more than one texel of the density texture is mapped on the sub-triangle D_i):
   (a) Subdivide the sub-triangle D_i to four smaller sub-triangles D_{i+1}.
   (b) For each sub-triangle D_{i+1} store the index of the triangle T and the sub-triangle D_{i+1} position inside T.
   (c) Increase the depth of recursion i by one and call step **2**. for every sub-triangle D_{i+1} of the sub-triangle D_i.
3. **Otherwise** (the maximum subdivision depth was reached):
   (a) Calculate the probability P_{D_i} of the sub-triangle D_i as the area of D_i multiplied by the density texture value mapped on D_i's barycenter.
   (b) **While** each of four sub-triangles D_i with the same parent sub-triangle D_{i-1} are not subdivided and have the same probabilities P_{D_i}:
      i. Discard sub-triangles D_i and use their parent D_{i-1} instead with the probability $P_{D_{i-1}} = 4P_{D_i}$.
      ii. Decrease the subdivision depth i by one and if $i = 0$ exit the while-cycle.
   (c) Store sub-triangles probabilities P_{D_i}.

Figure 2.3: The preprocessing stage of my algorithm.

and calculating how many texture texels are mapped on each of them. From the texture texel count I can easily determine maximum sub-triangle count. The number of sub-triangles of a single triangle in depth d is 4^d and because there must be at least as many sub-triangles as there are texels on a single triangle, I come to the equation $\text{texelCount}(T) \leq 4^d$, where $\text{texelCount}(T)$ is the number of texels mapped on a single triangle T. I can then derive the maximum subdivision depth d_{max} as:

$$d_{max} = \frac{\log_2 \left(\max_T \left(\text{texelCount}(T)\right)\right)}{2} \tag{2.3}$$

Since I need to store a sub-triangle index of every sub-triangle in depth $d = 0, .., d_{max}$, the pre-computed data structure will hold $\sum_{d=0}^{d_{max}} 4^d = (4^{d_{max}+1} - 1)/3$ sub-triangles, however the memory consumed by the pre-computed data structure is still negligible.

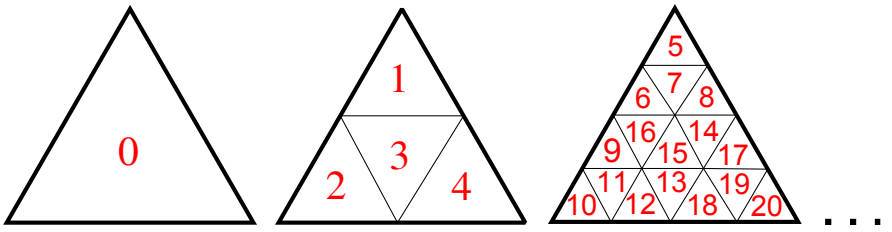


Figure 2.4: Index of a sub-triangle uniquely determines its position inside the parent triangle.

---

1. Allocate an array to hold all sub-triangles barycentric coordinates in depth $d = 0, .., d_{max}$.
2. Put a whole triangle barycentric coordinates $((0,0),(1,0),(0,1))$ to the data structure.
3. Mark $head$ as a pointer to the data structure start, $tail$ as $tail = head + 1$ and finally end as a pointer pointing just beyond the data structure end.
4. **While** $tail \neq end$
   (a) Select the barycentric coordinates of the sub-triangle D_{i-1} at which $head$ is pointing.
   (b) From these coordinates calculate barycentric coordinates of the sub-triangles D_i, which are children of the sub-triangle D_{i-1}.
   (c) Store calculated coordinates to the positions $tail + j, j = 0, 1, 2, 3$
   (d) $tail = tail + 4, head = head + 1$

Figure 2.5: The calculation of the pre-computed data structure which stores sub-triangle barycentric coordinates.

2.1.5 Stage 2: Generating point samples

Given the data computed in the preprocessing stage, generating a random sample is straightforward. First, I choose a sub-triangle D from the precomputed probability distribution (Section 2.1.6 provides details). I then draw a sample from a uniform distribution on the selected sub-triangle using a standard approach [31, p. 670] as follows: Given two random numbers ξ_1 and ξ_2 from the uniform distribution $U(0,1)$, the barycentric coordinates of the generated sample are $u_D = 1 - \sqrt{\xi_1}$ and $v_D = \xi_2 \cdot \sqrt{\xi_1}$. Finally, I map the sample to the barycentric coordinates (u, v) w.r.t. the parent mesh triangle:

$$\begin{aligned} u &= u_D \cdot u_1 + v_D \cdot u_2 + (1 - u_D - v_D) \cdot u_3 \\ v &= u_D \cdot v_1 + v_D \cdot v_2 + (1 - u_D - v_D) \cdot v_3 \end{aligned}$$

where u_i and v_i are the barycentric coordinates of the vertices of sub-triangle D inside its parent mesh triangle. I obtain u_i and v_i by a lookup in the pre-computed sub-triangle position data structure using the sub-triangle index.

As I will describe later, I also need one random number ξ to choose a sub-triangle. Sub-triangles are drawn from a 1D discrete distribution, but random number ξ is from continuous distribution $U(0,1)$, therefore for any $\xi \in [F_i, F_{i+1}]$ I will draw the same sub-triangle D_i (F_i represents a value of the CDF F corresponding to the sub-triangle D_i, as was defined before). Instead of generating two random numbers ξ_1 and ξ_2 to sample the sub-triangle area, I can generate only ξ_2 and calculate ξ_1 as $\xi_1 = (\xi - F_i)/(F_{i+1} - F_i)$. In the end I only need 2 random numbers for each generated sample which goes well with the fact that I am sampling a 2-dimensional surface. If I use a random generator which generates well distributed 2-dimensional points, good properties of these points have higher chance to improve quality of the distribution of the samples generated by my algorithm than if I would generate 3 random numbers per sample.

2.1.6 Fast sampling from a discrete 1D distribution

The first step of the procedure that generates a sample on the mesh involves drawing a sub-triangle from a 1D discrete distribution given by the CDF $F = [F_1, \dots F_{n_s}]$. The standard way to implement this is to use interval bisection to find sub-triangle $D_i = \arg\min_i \{F_i > \xi\}$, where ξ is a random number from $U(0,1)$ [31, p. 647]. Thanks to its logarithmic running time, this bisection algorithm is usually considered efficient for practical purposes. However, the number of sub-triangles can be large for complex meshes and the logarithmic search for generating each sample may incur a significant overhead.

I substantially improve the efficiency of sampling from a 1D discrete distribution by means of a uniform grid G over the codomain of F (i.e. the $[0,1]$ interval) created in the preprocessing stage. For each grid cell $C_k = [C_k^{\min}, C_k^{\max}]$ I compute two indices C_k^{begin} and C_k^{end}, as illustrated in Figure 2.6.

$$\begin{aligned} C_k^{\text{begin}} &= \arg\min_i \{F_i \geq C_k^{\min}\} \\ C_k^{\text{end}} &= \arg\min_i \{F_i \geq C_k^{\max}\} \end{aligned}$$

When drawing an element from the distribution (sub-triangle in our case), I first generate a random number ξ from $U(0,1)$ as before, then I look up, in constant time, the grid cell C_k for which $\xi \in C_k$, and finally I find the element using the bisection algorithm limited to the domain $[C_k^{\text{begin}}, C_k^{\text{end}}]$. Since $C_k^{\text{end}} = C_{k+1}^{\text{begin}}$ for every k, I only need to store one index per each cell.

Both the grid resolution and the characteristic of the probability distribution influence the performance of this approach. In my tests, the speedup due to grid-based search compared to the usual bisection varied between 2 and 14. Please note that this grid-based search is not limited to the particular mesh sampling application considered here. It is a general procedure for accelerated sampling from a 1D discrete distribution that may be used in other applications, such as sampling HDR environment maps.

The results of testing the grid-based search and the full sampling algorithm can be found in Section 4.1.1.

2.1.7 Random numbers

To generate a sample on a mesh I need to generate two random numbers first. The sampling algorithm is written in such a way, that if random numbers are

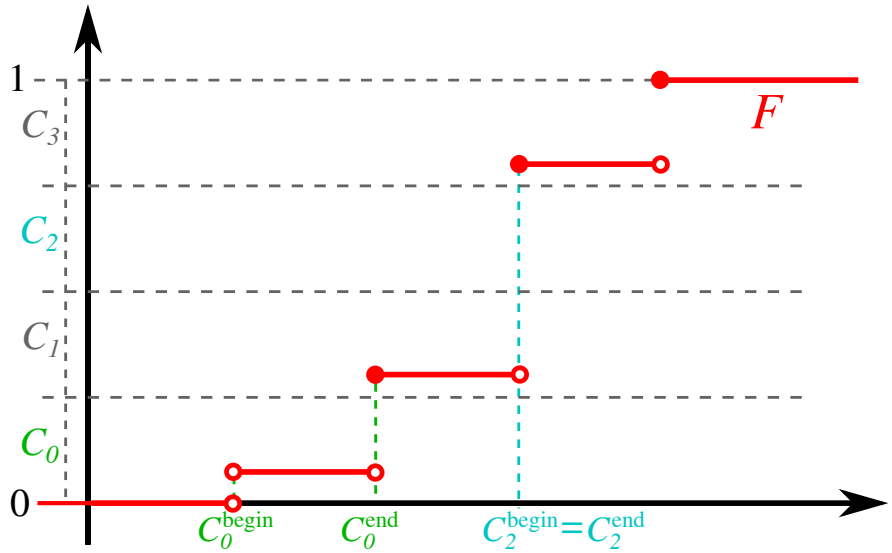


Figure 2.6: Uniform grid G built over the codomain of the cumulative distribution function F.

well distributed in 2 dimensions (two random numbers give one sample in 2D), samples on a mesh will also be nicely distributed. However generating well distributed random numbers in 2 dimensions is usually much slower than using a normal random generator. As I have said before, I use the sampling algorithm to generate both the generated hair and the hair guides root positions. Since there is maximally few thousands of the hair guides, I use the 2,3 Halton sequence (see Section 3.1.1) as the random numbers generator for my sampling algorithm. On the other hand, there can be millions of the generated hair, so I use the F. James random generator when generating their positions (see Section 3.1.1). In my tests, the F. James random generator proofed to be 8 times faster than the Halton sequence and as will be said later in Section 4.1.2, generation of random numbers always takes significant amount of time in the generation of a sample on a triangular mesh.

2.1.8 Sampling complex lights for Monte-Carlo rendering

I use the described sampling algorithm mainly for hair root positions generation; however it has other applications such as sampling complex lights in Monte-Carlo rendering (e.g. path-tracing). By a complex light I mean a light with a shape defined by a triangular mesh and emissivity defined by a 2-dimensional texture mapped on the mesh surface.

In Monte-Carlo rendering, Monte-Carlo methods are used to estimate the value of the fundamental rendering equation:

$$L_o(p, \omega_o) = \int_{S^2} f(p, \omega_o, \omega_i) L_i(p, \omega_i) |\cos \theta_i| \mathrm{d}\theta_i \tag{2.4}$$

I will not describe the equation nor the Monte-Carlo methods here (they are more than extensively described in [31]). All that you need to know, is that to efficiently evaluate the integral using Monte-Carlo random samples need to be generated on the light surface according to the emissivity (areas with higher

emissivity should receive more samples than areas with lower emissivity in order to decrease the variance of the integral estimate). That is exactly what my sampling algorithm does. Each generated sample must be also weighted by its probability. Note that the PDF p' used for sampling (Equation 2.2) will not, in general, be identical to the desired PDF (Equation 2.1). In order for Monte-Carlo to work correctly I need to use the probability from the probability distribution described by Equation 2.2.

Of course it is also possible to generate samples uniformly on the light surfaces without taking the emissivity into account. If I weigh the samples by correct probability, Monte-Carlo will yield mathematically correct results, however the variance of the integral estimate can be very large for highly varying emissivity textures. Despite this pitfall, uniform sampling is used very widely in currently existing rendering systems, because it is usually very fast and easy to implement.

(a) Uniform sampling

(b) My sampling algorithm

(c) Reference

Figure 2.7: Sampling illumination from a complex luminaire using uniform and my sampling algorithm.

Figure 2.7 shows three images of the same scene which has been only enlightened by the wall on which an HDR texture has been projected. This HDR texture determines the emissivity of the light. All of the images were rendered using path-tracer, however the first two images took an hour to render and the last (i.e. the reference image) took 16 hours to compute. Uniform light source sampling produces a highly noisy image and fails to reproduce the correct reddish tone of the illumination from a HDR map (which is due to tiny high-intensity

street lights that uniform sampling almost never samples). Light source sampling using my algorithm, on the other hand, reduces the noise and reproduces the correct tint of the image early in the progressive computation. One might object why I did not include rejection sampling of complex lights in the test. The reason is that rejection sampling would only accept 1 in about 2500 samples in this case and therefore it would took days to render anything reasonable.

2.2 Hair interpolation and procedural generation

The goal of this section is to describe all necessary theory which is required for the procedural hair generation. The procedural hair generation may be divided to several steps; each of them will be described closely in the following sections. A diagram of the whole procedural generation can be found in Figure 2.12. Before I describe the individual steps of the procedural hair generation in detail, I need to explain how is each hair represented during the hair generation which is goal of the next Section 2.2.1.

2.2.1 Hair representation

In order to select the best representation of hair, it is for the best to write down what is expected from the representation. First of all the internal representation of hair should be easily convertible to some geometric primitive which is easily renderable by RenderMan (one of thesis goals is to support RenderMan) and most importantly which resembles hair. Single hair fiber is usually rendered in RenderMan as a curve of varying width along the curve length. I will also use a curve in final rendering of hair, but I still have many choices which type of the curve to choose, since RenderMan supports any cubic curve, that can be declared by 4×4 matrix (see [32, p. 68]).

As said in the introduction, I create basic hair geometry by interpolating the hair guides. The hair guides are represented in the Stubble project as simple polylines that are defined by the vertices through which the polyline go. I would like to use a curve which could be defined by the same vertices as the hair guides and still would resemble the same shape as the hair guide polyline. Therefore it is an obvious choice to pick an interpolation curve, which passes through all the vertices that defines it, rather than an approximation curve, which may not pass through the vertices.

Many interpolation curves not only need vertices to define them, they also need curve tangents at these vertices. This is a desirable effect for a 3d artist since it gives her/him more control over the curve shape, however for a procedural hair generation it means that it must supply more parameters for the curve. To avoid generating the tangents I use the Kochanek-Bartels spline (see [2, p. 589–593] for more information), which only needs the vertices and three other values (tension, bias, continuity). The three values then control the behavior of the curve tangents. To further simplify problem of defining the curve I use a special type of the Kochanek-Bartels spline which is called the Catmull-Rom spline. The Catmull-Rom spline is the Kochanek-Bartels spline with all three values set to

zero, therefore vertices are all that is need to define the Catmull-Rom spline (sometimes referred as the Catmull-Rom curve). Because of its simplicity, the Catmull-Rom spline is one of the most used curves in Computer Graphics (along with Bezier and NURBS curves).

As I have said, the Catmull-Rom spline is defined only by its vertices and it passes through all of them. But this is only partly true. The first and the last vertex are only reached by the curve if they are equal to the second and to the one before the last, respectively. Position of the first and the last vertex only influences the Catmull-Rom spline shape. Why they are exactly needed will be more obvious in Section 2.2.8, where I will discuss how the Catmull-Rom spline computes the curve tangents from the supplied vertices. For the sake of simplicity, I will duplicate the first and the last vertex (see Figure 2.8), so the Catmull-Rom spline will behave as a polyline, only it will be smoother. During the hair generation I will usually ignore the duplicated first and last vertex (P_0 and P_6 in Figure 2.8), unless I need to compute directly with the Catmull-Rom spline.

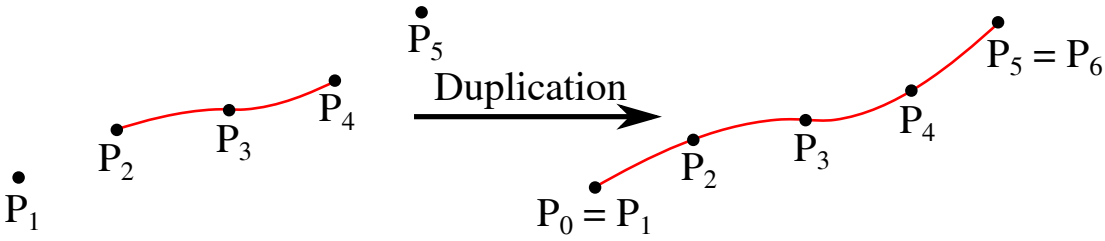


Figure 2.8: The duplication of vertices P_1 and P_5 ensures the desired Catmull-Rom spline behavior.

Of course defining a curve shape is not enough to create realistic hair. I also need to select hair color, opacity and width. Since these parameters may change along the curve length, I will define them for every vertex except the duplicated ones and interpolate them between every two vertices (see Figure 2.9).

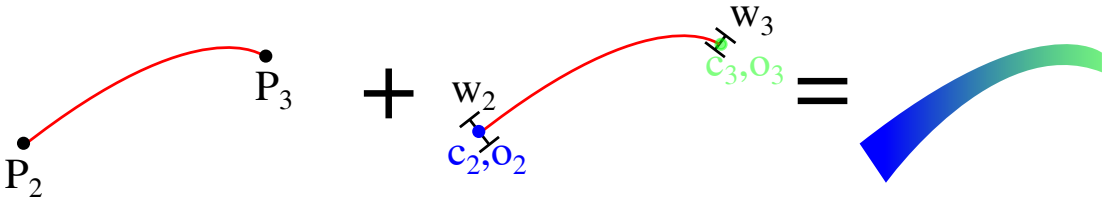


Figure 2.9: Color c_i, opacity o_i and width w_i are defined at the curve vertices and interpolated in between.

Sometimes I will have to use a curve coordinate frame (i.e. s curve normal, s tangent and s binormal at a defined position, see Figure 2.10) for some computations or as output to a renderer. I will also store the curve frame for the same vertices as color, opacity and width.

The last parameter of the generated hair is its position on the model surface. Not only is a root 3D position needed, but also a hair local coordinate frame is required during hair generation. Do not confuse the hair local coordinate frame (see Figure 2.11), which is one for each hair, with the curve coordinate frame (see Figure 2.10), which is defined for each hair curve vertex separately.

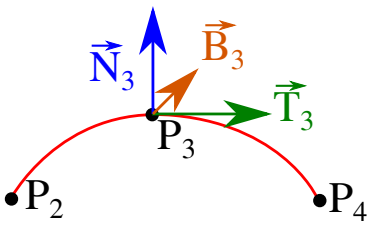


Figure 2.10: The curve frame defined at the vertex P_3. The normal is denoted as $\vec{N}_3$, the tangent as $\vec{T}_3$ and the binormal as $\vec{B}_3$.

Furthermore, texture coordinates at the hair root position are stored. Finally I also store the hair root position and the hair local coordinate frame for a mesh rest position. The rest position of the model is the state of the model without any transformations or deformations applied. Why all these parameters are necessary will become clear in the following sections.

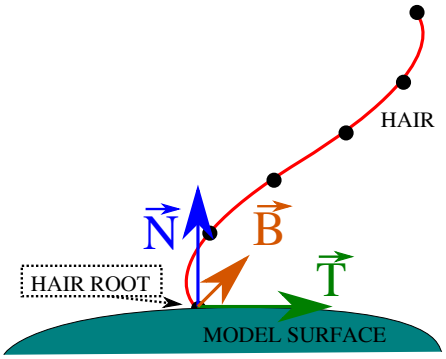


Figure 2.11: The hair local coordinate frame. The frame is defined by 3 perpendicular vectors: the model surface normal $\vec{N}$, the tangent $\vec{T}$ and the binormal $\vec{B}$.

Note, that not all of these hair parameters are defined during a whole time of hair generation, Figure 2.12 in Section 2.2.2 shows the lifetime of different hair parameters.

2.2.2 Procedural hair generation pipeline

In this section I will describe the pipeline of the procedural hair generation I have implemented. Figure 2.12 shows a diagram of the procedural hair generation pipeline. The boxes represents individual steps that are responsible for creating and changing the hair parameters described in the previous section. Furthermore, Figure 2.12 shows a lifetime of the hair parameters and which pipeline steps influence them.

Before I describe individual hair generation steps, I would like to mention here, how user can influence them. As was said in the introduction, the generated hair is interpolated from the hair guides, which are modeled directly by the user, however this is not the only way the user can control the hair generation. Several hair properties may be set that influence hair color, opacity, width and the shape of hair curves (to avoid confusion I will address the properties that the user can set as the hair properties and the hair vertices, normals etc. that are generated as

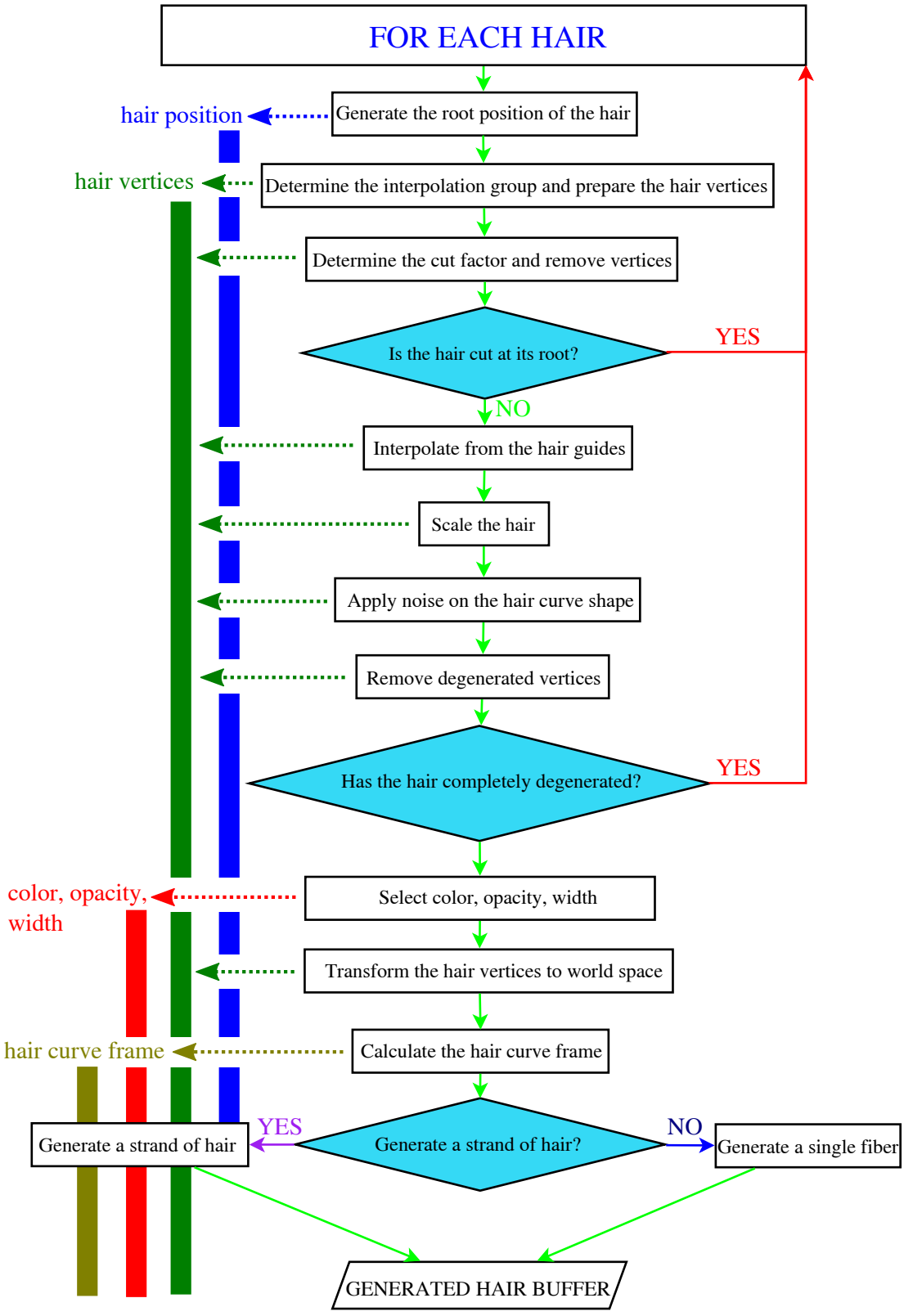


Figure 2.12: The procedural hair generation pipeline.

the hair parameters). These properties can be defined by a 2-dimensional texture which is mapped on the surface model and therefore each individual hair can have different properties. This answers the question why I need to remember texture

coordinates of each hair root – I will use them to access the hair properties from a texture. Furthermore, these textures may be modulated by a global value, which enables the user to quickly change a hair property for all hairs. I will use the following unified notification for hair properties:

- $\mathbf{property_T}$ will denote a texture property and $\mathbf{property_T}(u, v)$ the same property evaluated at the hair root texture coordinates u, v.
- $\mathbf{property_G}$ will denote a global property value (usually a texture modulator).

I will mention each hair property when I will be talking about individual steps of the hair generation pipeline.

Now I will only quickly describe the function of each step in the pipeline (the boxes in Figure 2.12), but most of them will be mentioned again in more detail in the following sections.

Generate the root position of the hair The root position of hair is generated on the surface using the previously described sampling algorithm. As I have mentioned in Section 2.2.1, not only is the 3D position of the hair root generated, but also the hair local coordinate frame and more. See Section 2.2.3 for more information.

Determine the interpolation group and prepare the hair vertices The generator selects the interpolation group in which the currently generated hair belongs. What are the interpolation groups will be discussed in Section 2.2.4, however I can already mention here that each interpolation group tells among other things how many vertices will represent every single hair from that group.

Determine the cut factor and remove vertices The user can define via the hair property texture $\mathbf{Cut\_Map_T}$ how much will each hair be cut. The value of $\mathbf{Cut\_Map_T}(u, v)$ for the currently generated hair is stored as the hair cut factor. The hair cut factor value can go from 0, which means completely cut hair, to 1, which means that the current hair will not be cut at all. The hair cut factor can also tell me how many hair vertices can be discarded from following pipeline steps, since they do not influence the hair that remained after the cut. Cutting the generated hair will be more discussed in Section 2.2.8.

Is the hair cut at its root? If the hair cut factor (described in the previous paragraph) is equal to 0 I can discard whole hair and start with a generation of next hair.

Interpolate from the hair guides This is one of the most important steps in the pipeline. As its name suggests, it is responsible for the hair interpolation, which is the basic element in determining a hair curve shape. See Section 2.2.4 for more details. One detail I have to mention here is that the hair interpolation calculates the hair vertices in the hair local coordinate system. I will use them in that coordinate system for several steps in the pipeline, until I convert them to final world space coordinates.

Scale the hair After the hair interpolation has given hair its primary shape, I can apply some transformations on hair vertices. First of them is scaling, since it is quite simple operation I will describe it here. Each of the hair vertices is scaled by the $scale_{total}$ factor (by a scalar multiplication) The $scale_{total}$ factor has two parts, a static and a random. The static part is calculated from the hair properties as follows: $scale_{static} = \mathbf{Scale_G} \cdot \mathbf{Scale_T}(u, v)$ and the random scale is: $scale_{random} = 1 - \mathbf{Random\_Scale_G} \cdot \mathbf{Random\_Scale_T}(u, v) \cdot \xi$, where ξ is a random number $\in [0, 1]$. $scale_{static}$ can increase/decrease hair scale unlimitedly, since a value of $\mathbf{Scale_G}$ is not limited at all. On the other hand, $\mathbf{Random\_Scale_G}$ is limited to the interval $[0, 1]$, therefore $scale_{random}$ can only decrease hair size by randomly downscaling it. The total $scale_{total}$ factor is then calculated as a multiplication of $scale_{static}$ and $scale_{random}$. As I have said before, hair vertices are currently stored in the local hair coordinates, so the root of hair in the hair local coordinates is $(0, 0, 0)$ and therefore it is not influenced by scaling.

Apply noise on the hair curve shape Another step that influences the hair vertices uses noise and several hair properties to create more realistically looking hair. How exactly it is done is said in Section 2.2.5.

Remove degenerated vertices Some of the previous steps may have caused that two vertices are equal (i.e. their position is the same). This is undesirable for next steps and it could lead to wrong calculations, therefore I remove one vertex from each two equal vertices.

Has the hair completely degenerated? If the previous step has removed all hair vertices except one, the current hair has completely degenerated and can be removed. The generator then immediately continues with a generation of next hair.

Select color, opacity, width For this moment the generator has determined the shape of the hair curve, so it focuses on selecting color, opacity and width of the hair. See Section 2.2.6 for more details.

Transform the hair vertices to world space Before the last hair parameters are calculated, the generator transforms the hair vertices from the hair local coordinate system to a world space coordinate system. I do this because it is much easier and efficient to output a bunch of hair curves to a renderer without the need to tell the renderer how to transform them. However if the user wishes to generate strands of hair (see following paragraphs), the transform is not applied at this moment. The reason for this will become clear in Section 2.2.7.

Calculate the hair curve frame The generator still has to calculate the hair curve frame (i.e. a normal, a tangent and a binormal at each hair vertex). Tangents are generated first and are used to define the curve normals, which will be later output to the renderer. The binormals are calculated only if the user wishes to generate strands of hair (see next paragraph). The hair curve frame calculation is described in more detail in Section 2.2.8.

Generate a strand of hair? The user can specify whether the hair generator will generate only single fibers of hair or whole strands. Depending on this, one of the two following steps will be executed. What exactly I mean by hair strand is described in Section 2.2.7.

Generate a single fiber The user wants to generate only single fibers, so the generator takes the already calculated hair parameters, executes some final computations and outputs the hair to the buffer where it awaits for rendering. For more information about the final stage see Section 2.2.9.

Generate a strand of hair At this step the generator takes the so far generated single hair and creates from it a whole strand of hair. This process is described in Section 2.2.7. After that for each single fiber from the strand, the generator transforms its vertices to world space coordinates, calculates the hair curve frame and executes the final stage mentioned in the previous paragraph.

Please note, that the actual implementation of the hair generator has some minor changes from the pipeline described here (mostly for performance issues). I don't mention these details here to keep the pipeline description as simple as it can be. The following sections describe details that were omitted from this brief description of the hair generation.

2.2.3 Determining hair root positions

Determining hair root positions can be divided to two parts. In the first part, the mesh sampling algorithm is executed and it yields a sample. The sample consists of a triangle identifier and the barycentric coordinates of the sample in the triangle. In the second part, the generator calculates from the sample the hair root position along with the hair coordinate frame (see Figure 2.11, texture coordinates and on the current state of the mesh and on the rest position mesh. To ensure that the hair roots are generated on the same surface positions for every animation frame (the hair generation must be executed again for every frame), the mesh sampling algorithm uses the rest position mesh as input along with $\mathbf{Density\_Map_T}$ which defines hair distribution on the mesh surface. The hair roots will of course undergo same transformations and deformations as the animated mesh; however each of them will always be in the same triangle and at the same barycentric coordinates during the whole animation.

The first part, generating the sample, is quite clear, so let's move to discuss the second part. There are two possible ways how the second part of this mesh generation step will be executed. Which will be used depends on whether the user wants to use mesh displacement. Mesh displacement is a method to add additional details to model geometry during render time. These details are defined by a single channel texture, which is mapped on the surface (see Figure 2.13). Each point of the model is then displaced by a factor given by the texture along the surface normal at the point[1]. So if mesh displacement is used, the generator needs

[1]Note that there is also vector displacement which can displace the point along an arbitrary vector given by any three channel texture, however my generator does not support this type of displacement yet.

to recalculate a hair root position according to it as described in Section 2.2.3, otherwise it can use more simple computations described in Section 2.2.3.

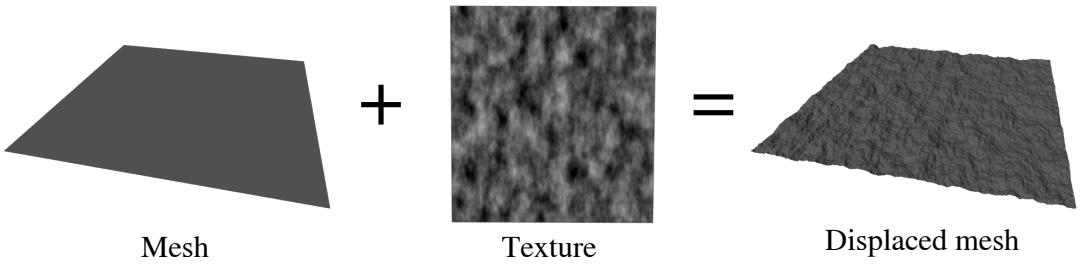


Figure 2.13: The simple example of mesh displacement. The mesh on the left is displaced by the texture in the middle. The displaced mesh can then be seen on the right.

Standard root position calculation

The sample generated by the mesh sampling algorithm contains an identifier of the triangle on which the sample lies. The generator first selects the triangle vertices from the current state of the mesh. For each triangle vertex its 3D world space position, surface normal, surface tangent and texture coordinates are selected. Then I can easily calculate the root position, the normal etc. using following formula:

$$root_{var} = sample_u \cdot v^0_{var} + sample_v \cdot v^1_{var} + (1 - sample_u - sample_v) \cdot v^2_{var}$$

where v^i_{var} represents ith vertex position, normal etc. and $sample_u$, $sample_v$ are the barycentric coordinates of the generated sample.

To get the hair local coordinate system, the binormal must be calculated and all three vectors (the normal, the binormal, the tangent) must be perpendicular and have unit size. To achieve this, the normal is normalized first and then the tangent is made orthogonal to the normal using formula:

$$tangent = tangent - (tangent \cdot normal)normal$$

where $\cdot$ represents a scalar multiplication. After the tangent is normalized, the binormal is calculated as $binormal = tangent \times normal$, where $\times$ represents a vector multiplication.

Finally the root position and the hair local coordinate system based on the rest position mesh are calculated the same way as the properties based on the current state of the mesh.

Mesh displacement

Before a displaced mesh is handled, the standard root position calculation as described in the previous section is executed. The non-displaced position and the coordinate frame of hair will be used to calculate the displaced position. The hair root position on the rest position mesh is never displaced, since it is only required for some internal calculations independent on mesh displacement.

There are two things that must be done when hair is generated on the displaced mesh:

1. First, the root position P' of hair in 3D space is calculated. This is quite easy, all that is required is to get the position P and the normal N on the non-displaced mesh and evaluate the texture, which defines the displacement at the texture coordinates corresponding to the position P. Let's assume the displacement texture D returns float value $D(u, v)$, then P' is calculated as: $P' = P + D(u, v) \cdot N$.

2. The hair local coordinate frame on the displaced mesh, defined by the normal N', the tangent T' and the binormal B', must also be calculated. Let's assume N' is already computed, then the calculation of B' and T' is the same as in the standard root position calculation: $T' = \widehat{(T \cdot N') \cdot (T - N')}$ and $B' = T' \times N'$, where T is the tangent on the non-displaced mesh and $\widehat{V}$ represents normalization of a vector. All that is needed now is to calculate N'. The calculation of a shading normal on a mesh surface with applied bump-mapping described in [31] may be used for this purpose.

Before the normal N' is calculated, the generator must compute partial derivatives of a normal and a position on each mesh triangle with respect to texture coordinates.

Formula from [31, p. 142–143] describes the calculation of the position partial derivatives:

$$\begin{bmatrix} \partial P/\partial u \\ \partial P/\partial v \end{bmatrix} = \begin{bmatrix} u_1 - u_3 & v_1 - v_3 \\ u_2 - u_3 & v_2 - v_3 \end{bmatrix} \cdot \begin{bmatrix} p_1 - p_3 \\ p_2 - p_3 \end{bmatrix}$$

where p_i is the ith vertex position, u_i and v_i are the texture coordinates at p_i. The calculation of the normal partial derivations is similar.

Since N' can be defined as $N' = \partial P'/\partial u \times \partial P'/\partial v$ (u, v are texture coordinates), therefore $\partial P'/\partial u$ and $\partial P'/\partial v$ must be calculated. Using the formula $P' = P + D \cdot N$ and the chain rule, $\partial P'/\partial u$ can be computed (and $\partial P'/\partial v$ likewise) as:

$$\frac{\partial P'}{\partial u} = \frac{\partial P}{\partial u} + \frac{\partial D(u,v)}{\partial u} N + D(u,v) \frac{\partial N}{\partial u} \approx \frac{\partial P}{\partial u} + \frac{D(u + \triangle_u) - D(u,v)}{\triangle_u} N + D \frac{\partial N}{\partial u}$$

The explanation of these formulas can be found in [31, p. 494–495]. To compute derivatives of $D(u, v)$ I use the following formula:

$$\frac{D(u + \triangle_u) - D(u,v)}{\triangle_u} = \frac{D(u + width^{-1}) - D(u,v)}{width^{-1}}$$

where $width$ is the width of the displacement texture D in texels ($\frac{D(u,v+\triangle_v)-D(u,v)}{\triangle_v}$ is calculated likewise).

2.2.4 Hair interpolation

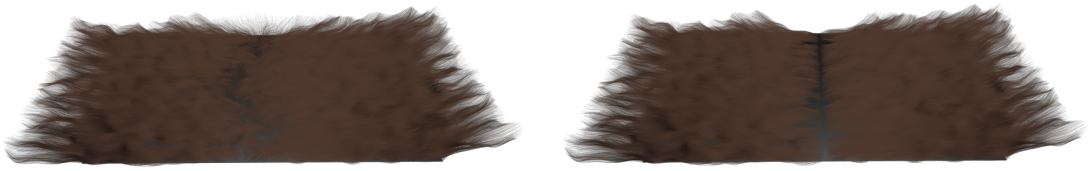
As I have already mentioned several times, the most important part of the hair generation is interpolation from the hair guides. Since the generated hair and each hair guide are defined by vertices, the generator can interpolate the hair by interpolating the hair vertices from the vertices of the hair guides. If the hair has the same number of the vertices as each hair guide, the generator can easily interpolate only the corresponding vertices: the first hair vertex will be

interpolated only from the first hair guides vertices, the second hair vertex from the second hair guides vertices and so on. This makes the hair interpolation lot easier; however it limits the user in selecting the number of the hair vertices. To remove this limitation and gain other advantages, the generator can use the interpolation groups.

Interpolation groups

Both the hair and the hair guides can be divided to several interpolation groups. The hair from one interpolation group will then be interpolated only from the hair guides from the same interpolation group. The first advantage of using the interpolation groups is that the user can specify the number of the vertices for the hair and the hair guides per each interpolation group. This comes handy when for example the user wants to model a horse with millions of short hairs over the horse body, but with long hair on its tail and head. To represent short hairs, few vertices per hair are enough, but for the long hair it is necessary to use many vertices for smoother control over hair curve shapes. Using many vertices for both short and long hair would unnecessarily increase the time spend on hair generation.

Another huge advantage of using the interpolation groups is creating discontinuities in otherwise smooth interpolation. Now imagine the user wants to generate hair on human head and he would like to create a 'hair parting'. He will use the tools from the Stubble software to brush the hair guides on the left side of the head to left and the hair guides on the right side to right. However if the user does not specify the interpolation groups, the hair growing between the left and the right guides will be interpolated from both and therefore the 'hair parting' will not be created. To create the 'hair parting', the user must create discontinuity by specifying two interpolation groups, one for the left side of the head and one for the right side. See Figure 2.14 for a demonstration of interpolation groups.



(a) Without the interpolation groups (b) With the interpolation groups

Figure 2.14: On the left image the 'hair parting' was created without the use of the interpolation groups and in the right image the same 'hair parting' was made with the use of two the interpolation groups.

The most easier and flexible way to define the interpolation groups is to use a texture, let's denote it as $\mathbf{InterpolationGroups_T}$. The $\mathbf{InterpolationGroups_T}$ texture is mapped on the model surface and each color from this texture then defines one interpolation group. The user interface of Stubble also gives the user the ability to choose the number of the vertices for each interpolation group. When the hair generator wants to know, in which interpolation group the hair is, it only has to evaluate $\mathbf{InterpolationGroups_T}$ at the texture coordinates corresponding to the hair root position.

Interpolation method

Now I have to determine how exactly I will interpolate the hair vertices. Let's assume that I have a function $hair_i(x)$. This function tells me a position of the ith hair vertex for hair positioned at the point x (remember that I interpolate each ith vertex of hair alone, so I have a special function for each vertex index i). I will define the hair position x as the hair root position (or the hair guide root position) on the mesh surface in its rest position state, therefore the hair position x will remain the same during animation. Now my function $hair_i(x)$ is defined only for x where any hair guide root lies, for easier notation let's put all x for which the function $hair_i(x)$ is defined to the set G_i. It is easy to observe that $\forall i, j : G_i = G_j$, therefore I will denote $G = G_i$. For $x \notin G$ I will have to use interpolation. Since the positions $x \in G$ are more or less randomly scattered through the 3-dimensional space (in fact they all are from the model surface, however it is hard to use such coherency, so I will just assume they are randomly scattered) and so are the positions at which I want to evaluate the function, I need to use a special type of interpolation called *Scattered Data Interpolation*. *Scattered Data Interpolation* is able to give me a value of my $hair_i(x)$ at any x based on interpolation from the few defined $hair_i(x)$ values.

There are several ways to interpolate from scattered data, some of them are described in [3] and [16]. However most of these methods are very slow and also hard to implement in three dimensions. Here I will only describe three methods that could be possible used for my hair generator.

Nearest neighbor The simplest method for interpolating scattered data is called nearest neighbor. It is not even interpolation in correct sense, since for any $x \notin G$ it only picks the closest $x' \in G$ and defines the $hair_i(x)$ value as $hair_i(x) = hair_i(x')$. This method creates horrible discontinuities everywhere, not only where the user has specified by the interpolation groups.

Inverse distance weighting This method was first published by Shepard in [36]. To evaluate $hair_i(x)$ for $x \notin G$ it first takes all $x' \in G$ and calculate for each of them weight based on the distance to x. Then it interpolates $hair_i(x)$ from all $hair_i(x'), x' \in G$ weighted by the calculated weights. The basic version then looks like this:

$$\forall x' \in G : w(x') = \frac{1}{dist(x, x')^p}$$

$$hair_i(x) = \sum_{x' \in G} \frac{w(x') hair_i(x')}{\sum_{x' \in G} w(x')}$$

The symbol $dist(x, x')$ represents a distance between x and x' and p is an user parameter. Greater values of p assign greater influence to x' closest to x and for p approaching infinity, this method will behave as the aforementioned nearest neighbor method.

The inverse distance weighting works quite well, however it can be rather slow if I would calculate the value of $hair_i(x)$ from all $hair_i(x'), x' \in G$. Therefore it is better to use an modified version (see [3]), which incorporates only few closest

$x' \in G$ that are within a sphere $S(x, r)$ of a radius r with a center at x:

$$\forall x' \in G : \widetilde{w}(x') = \left(\frac{r - dist(x, x')}{r dist(x, x')} \right)^2$$

$$hair_i(x) = \sum_{x' \in S(x,r)} \frac{\widetilde{w}(x') hair_i(x')}{\sum_{x' \in S(x,r)} \widetilde{w}(x')}$$

This modification makes inverse distance weighting very fast and still retains its quality.

Natural neighbor The last method I describe first selects the domain D for which I want to define $hair_i(x)$ and splits it to the Voronoi cells $V(x_j)$ according to the vertices $x_j \in G$. The Voronoi cell $V(x_j)$ is defined as:

$$\forall x_j \in G : V(x_j) = \{x \in D | d(x, x_j) \leq d(x, x_k) \forall x_k \in G\}$$

See [4] for more information about Voronoi diagrams (the Voronoi diagram is a set of Voronoi cells). After the Voronoi cells $V(x_j)$ were calculated, I can evaluate $hair_i(x)$ for $x \notin G$. First I calculate another Voronoi cell $V(x)$ for the point x:

$$V(x) = \{y \in D | d(y, x) \leq d(y, x_j) \forall x_j \in G\}$$

Then I select all $x_j \in G$ for which $V(x_j) \cap V(x) \neq \emptyset$ and put them to the set N. Finally I calculate weights for $x_j \in N$ and evaluate $hair_i(x)$:

$$\forall x_j \in N : w(x_j) = \frac{vol(V(x_j) \cap V(x))}{vol(V(x))}$$

$$hair_i(x) = \sum_{x_j \in N} w(x_j) hair_i(x_j)$$

where the function $vol(z)$ returns a volume of z. Since $\sum_{x_j \in N} w(x_j) = 1$, normalization is not required. This method has several improvements which can be found in [3]. This method gives great results, however is slow and hard to implement (especially in higher dimensions than 2) compared to the previous two mentioned.

Figure 2.15 shows a comparison of interpolation methods mentioned here. It is obvious that the best results are from natural neighbor method, however since it is hard to implement and very slow compared to the other two methods, I choose inverse distance weighting method, which still gives good results and is also very fast.

Interpolation details

I have picked inverse distance weighting method to interpolate hair; however there are still some unresolved issues. First of all, inverse distance weighting method selects several closest hair guides in a sphere of a radius r to interpolate hair. In order for this to work, the user would have to choose the radius r. This radius would vary for each scene and hair guides density. Therefore I will rather have the user to pick the integer number $n = \mathbf{Interpolation\_Samples_G}$, which will tell

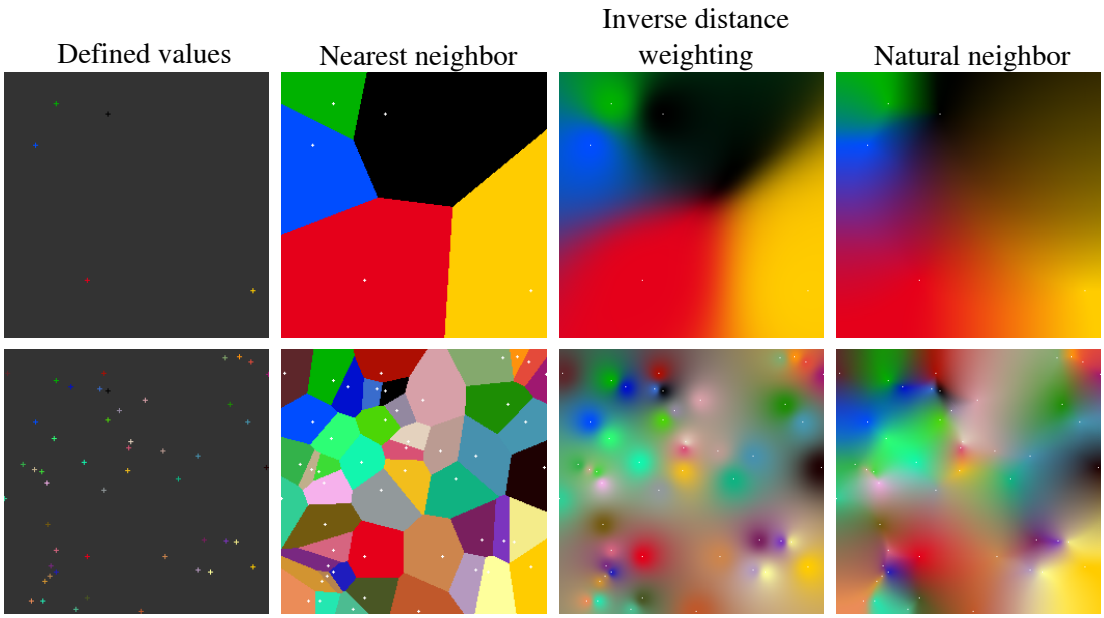


Figure 2.15: The comparison of nearest neighbor, inverse distance weighting and natural neighbor methods for colors on a 2 dimensional plane. The two left-most images shows defined values for the interpolated function (top has 5 values, bottom 50 values) and the images right of them shows different methods results.

the generator how many hair guides it should use to interpolate a single hair. The generator then selects the n closest hair guides (let's denote them as $x' \in C_n(x)$) to the currently interpolated hair and calculate the radius r for inverse distance weighting method as:

$$r = max_{x' \in C_n(x)} dist(x, x')$$

Since setting the radius r like this will result in zero contribution of the farthest hair guide, I rather select the $n + 1$ closest hair guides (although this does not ensure that n hair guides will have non-zero contribution).

Since x used as an input to evaluate the function $hair_i(x)$ is actually the hair root position on the model surface, it would be preferable to calculate $dist(x, x')$ as an geodesic distance between x and x' on the model surface. However that would be complicated and very time consuming, so instead I use Euclidean distance in a 3-dimensional space.

During the interpolation I will need to quickly search for the n closest hair guides. To do that I build a KD-Tree over the hair guides roots. KD-Tree is a spatial data structure which recursively divides space to two nodes by a 2-dimensional plane aligned with one of the three coordinates. The division continues until leaf nodes in a hierarchy contains limited number of elements (in this case the hair guides roots) or the recursion has reached a certain level. KD-Tree then can be quired for the n closest points (the hair guides) from some arbitrary point x. See [19] for more information about KD-Trees and for a description of the query algorithm. Since I use the interpolation groups, I will have a different KD-Tree for each group, so I can easily search for the closest hair guides from a desired interpolation group.

One final problem that I have to discuss here is whether to use the world coordinates of the hair vertices or the hair local coordinates of the hair vertices

during the hair interpolation. It would make sense to use the world coordinates, since the hair and the hair guides local coordinates systems will be different, however there are at least two major problems with using the world coordinates during the interpolation:

- First, when I have for example an ear of an animal, it has hair on both of its sides. Since an ear may be quite thin, the hair on the one side may be interpolated from the hair guides located on both sides of the ear. The hair guides from the other side will surely point in nearly opposite direction than the guides on the same side as the hair and therefore interpolated hair geometry will not be correct.
- The other problem is that when the mesh on which hair grows is deformed, it will change the world coordinates of the hair guides vertices and therefore the interpolated hair will be also deformed, which is not correct behavior. The interpolated hair should retain its shape, when the mesh underneath it is deformed.

These two problems can be easily solved by interpolating the hair vertices in the local coordinates systems. Furthermore if the hair interpolation is more considered as determining the shape of hair curves rather than interpolating individual vertices, using different hair and hair guides local coordinates systems is actually correct.

Figure 2.16 shows the final algorithm for the hair interpolation.

---

1. Select the closest hair guides from hair with the same interpolation group.
2. Among the closest hair guides choose the farthest one and use its distance from the hair root as the sphere radius.
3. Calculate weights for the selected hair guides.
4. Select the vertices of the selected hair guides.
5. **For** $i = 0, .., (\text{the number of hair vertices}) - 1$
   (a) Calculate the hair ith vertex using inverse distance weighting from the ith vertices of the selected hair guides.

Figure 2.16: The final algorithm for the hair interpolation.

2.2.5 Influencing the hair curve by noise

After the generator has determined initial geometry of the currently processed hair by interpolating from the nearest hair guides, it scales the hair vertices according to the user parameters (as described in Section 2.2.2) and then it applies noise to the hair vertices, which is the topic of this section.

Before the noise is applied there is nearly no random element present in hair geometry (only random scaling) and therefore the hair may look unnatural. Of course the hair guides can be modeled is such way that they appear realistic, however there is usually much more final hairs interpolated from the hair guides than the hair guides themselves, so the realism of the hair must be delivered by another step of the hair generation pipeline other than the hair interpolation. I use two types of deformations, which use noise, to improve hair realism: *Frizz* and *Kink*. Compare the picture of the generated hair with no noise and the image of the hair with added the *Frizz* and *Kink* deformations in Figure 2.17.

(a) Hair with no noise (b) Hair with noise

Figure 2.17: The generated hair with no noise applied (left) and the same rendered hair with the *Frizz* and *Kink* deformations applied (right).

Before I describe these two deformations, I will first talk about noise itself.

Noise

In general, noise is usually a smoothly varying function $\mathbb{R}^n \to [-1, 1]$. The function values appear random and have no obvious repetition. One huge advantage of the noise function compared to a random generator, is that it is smooth, so for two close input values the noise function returns two close output values. In my case it means that two hairs that are close to each other will be deformed in similar manner.

In my generator I use the noise function introduced by Ken Perlin (see [30]). This function is known as *Perlin noise*. The function uses lattice build over $\mathbb{R}^n$ (I will always use $n = 3$) and for each integer value $(x_0, \ldots, x_{n-1})$ it returns a gradient vector (these vectors are found by indexing to a precomputed table of integer values). When the user supplies a parameter to *Perlin noise*, it selects 2^n nearest lattice points and the corresponding gradient vectors. After that it calculates weights of the 2^n nearest lattice points as the scalar multiplication of the gradient vector and the vectors from the corresponding lattice point to the user input parameter (see Figure 2.18).

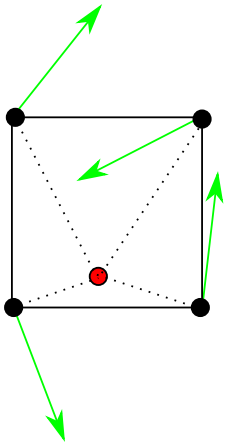


Figure 2.18: The scalar multiplication of the gradient vectors (green lines) with the vectors from the lattice points to the user input parameter (dotted lines) gives the influence of each gradient to a noise value at the user input parameter (red dot).

Finally a value of the noise function is interpolated from these 2^n weights based on the user parameter position among the 2^n nearest lattice points. For more details see the two aforementioned sources. Figure 2.19 shows an example of *Perlin noise* on a sphere.

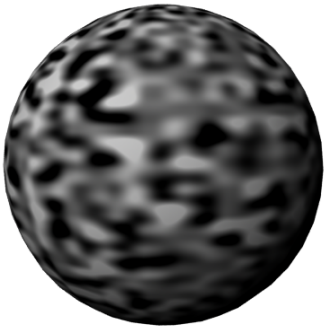


Figure 2.19: A rendered sphere with a color defined by a 2-dimensional variant of *Perlin noise* (texture coordinates were used as an input).

In the hair generator I use a special type of *Perlin noise* function which returns a 3-dimensional point instead of one value. It is defined as:

$$noise3D(\vec{x}) = [noise(\vec{a_0} + m_0\vec{x}),\ noise(\vec{a_1} + m_1\vec{x}),\ noise(\vec{a_2} + m_2\vec{x})]$$

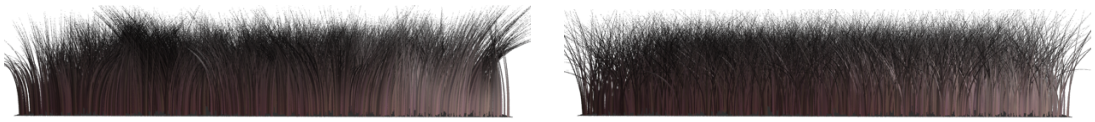
where *noise* is the original *Perlin noise* function and $\vec{a_i}, m_i$ are some values that shifts and scales the original user input. $\vec{a_i}, m_i$ depends on an implementation of the $noise3D$ function, in my generator I use the $noise3D$ from the 3Delight library.

Frizz

Now that I have talked about noise functions, I can begin to describe the first deformation that uses it. As the name of *Frizz* suggests, it causes the hair curve

to bend in randomly chosen directions. The hair bending is defined for a whole hair fiber, so the generator needs to ask the $noise3D$ function once for every hair. In nature hair close to each other is frizzed in nearly the same way. I get this for free from the Perlin noise function, I only have to make sure that I generate a similar input for neighboring hair. Since I also want the noise to be consistent with frame animation, I choose as input to the Perlin noise function the position of the hair root on the rest position mesh.

Sometimes the user wants to change *Frizz* much slowly over the mesh, so the hair looks all bend in the same direction, other time she/he wants more noisy hair. This can be simply controlled by a frequency of the $noise3D$ function (see Figure 2.20). Since I use the hair 3D root position as the input, I can control the frequency separately in 3 dimensions. The user can specify the frequency for all three dimensions using both the global property $\textbf{Frizz\_?\_Frequency}_{\textbf{G}}$ and the texture property $\textbf{Frizz\_?\_Frequency}_{\textbf{T}}$, where the ? can be X,Y or Z.



(a) **Frizz_X_Frequency** = 0.2 (b) **Frizz_X_Frequency** = 2

Figure 2.20: A demonstration of **Frizz_X_Frequency** property influence on the hair.

The final noise function input X component is calculated as:

$$input.x = \textbf{Frizz\_X\_Frequency}_{\textbf{G}} \cdot \textbf{Frizz\_X\_Frequency}_{\textbf{T}}(u, v) \cdot hairRoot.x$$

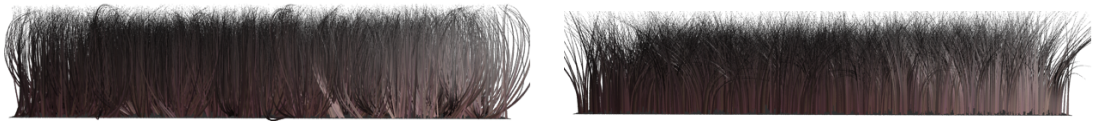
where $hairRoot.x$ is the coordinate X of the hair root position on the rest position mesh in the world space coordinates. The input Y and Z components are computed likewise. The $noise3D$ then returns the 3-dimensional point $P \in [-1, 1]^3$ which I will use as the vector $\vec{F}$ and bend the hair in its direction (in fact, the implementation of $noise3D$ I use returns $P \in [0, 1]^3$, so I have to scale it like this : $P' = 2(P - 0.5)$ in order to get $P' \in [-1, 1]^3$ and therefore F will go in all possible directions). It would make sense to normalize $\vec{F}$, however since normalization is costly and it made nearly no difference in my tests, I don't use it.

The user can also choose how much is each hair influenced by the vector $\vec{F}$ at a hair root and at a hair tip (see Figure 2.21). The corresponding properties are $\textbf{Root\_Frizz}_{\textbf{G}}$, $\textbf{Root\_Frizz}_{\textbf{T}}$, $\textbf{Tip\_Frizz}_{\textbf{G}}$, $\textbf{Tip\_Frizz}_{\textbf{T}}$. From these I compute two scalar values that influence the hair bending:

$$frizz_{root} = \textbf{Root\_Frizz}_{\textbf{G}} \cdot \textbf{Root\_Frizz}_{\textbf{T}}(u, v)$$

$$frizz_{tip} = \textbf{Tip\_Frizz}_{\textbf{G}} \cdot \textbf{Tip\_Frizz}_{\textbf{T}}(u, v)$$

Now I want to influence each hair vertex (except the hair root vertex) by $\vec{F}$, $frizz_{root}$ and $frizz_{tip}$, so the resulting hair curve looks smoothly frizzed. That could be achieved by creating the vector function $frizz(t)$, that excepts the t parameter of the hair curve and returns scaled vector $\vec{F}$. The t parameter has value 0 at the hair root and value 1 at the hair tip and between them is smoothly changing along the curve length. More about curve parameters can be found in



(a) **Root_Frizz** = 3 (b) **Tip_Frizz** = 3

Figure 2.21: **Root_Frizz** and **Tip_Frizz** control the amplitude of *Frizz* at the hair root and at the hair tip, respectively.

Section 2.2.8 The scaling of the vector $\vec{F}$ depends on how close is the t parameter to the tip or the root of the hair and on the $frizz_{root}$ and the $frizz_{tip}$ number. I can't use simple linear function, because then the generated hair would not look natural. By trial and error I have determined the function $frizz(t)$ as follows:

$$frizz(t) = \vec{F} \cdot \min\left(frizz_{root} \cdot 4(t - t^2) + frizz_{tip} \cdot t^2, \max\left(frizz_{root}, frizz_{tip}\right)\right)$$

where the minimum function makes sure that non of the hair vertices will be frizzed more than is the maximum frizz defined by $\max\left(frizz_{root}, frizz_{tip}\right)$.

Frizzing the hair can be used for more than just a simple random bending of hair. If the hair is frizzed with slowly changing noise, it looks like breeze playing with hair or grass. Therefore I will upgrade the *Frizz* deformation noise to have a dynamic part. I will keep the function $frizz(t)$ as it is, but instead of F I will use the new vector $F_{combined}$, that will be the combination of F and the vector $F_{dynamic}$, which will dynamically change during an animation.

As I have said, I would like to slowly change *Frizz* for all hair. Since I use a noise function that receives a 3D point (or a vector) as input, the user needs to supply a vector through property $\mathbf{Frizz\_Anim\_Direction_G}$, which will define a direction of a noise change. Also I would like to know how fast noise is changing during an animation time. That is defined by the hair property $\mathbf{Frizz\_Anim\_Speed_G}$ and its texture variant $\mathbf{Frizz\_Anim\_Speed_T}$. To compute $F_{dynamic}$ I use the noise function $noise3D$, which input component X is:

$$\begin{aligned} input.x = &\mathbf{Frizz\_X\_Frequency_G} \cdot \mathbf{Frizz\_X\_Frequency_T}(u, v) \cdot \\ &\cdot dynamic.x \cdot hairRoot.x \end{aligned}$$

which is similar to the input used to compute F except *dynamic*. *dynamic* is a 3-dimensional vector and represents the noise change in time. It is calculated as:

$$\begin{aligned} dynamic = &\mathbf{Frizz\_Anim\_Speed_G} \cdot \mathbf{Frizz\_Anim\_Speed_T}(u, v) \cdot \\ &\cdot time \cdot \mathbf{Frizz\_Anim\_Direction_G} \end{aligned}$$

where *time* represents the current animation frame time.

Now I only need to merge F and $F_{dynamic}$ together. For this purpose I have another two hair properties associated with frizz, $\mathbf{Frizz\_Anim_G}$ and $\mathbf{Frizz\_Anim_T}$. From these two I calculate $animationFactor$:

$$animationFactor = \mathbf{Frizz\_Anim_G} \cdot \mathbf{Frizz\_Anim_T}(u, v)$$

Finally I can compute the vector $F_{combined}$ as:

$$F_{combined} = (1 - animationFactor) \cdot F + animationFactor \cdot F_{dynamic}.$$

It can be noticed that *Frizz* can make the hair grow larger. This is quite annoying in the direction along a surface normal at the hair root position, so I have made a little fix. I keep the X,Y components of $F_{combined}$ as they are, but I will change the Z coordinate like this: $F_{combined}.z = -\|F_{combined}.z\|$, where $\|a\|$ represents an absolute value of a. Since the Z coordinate base vector corresponds to the surface normal, it will cause that the hair will bend more to the surface.

So now that I have completely defined *Frizz* as the vector function $frizz(t)$, the generator will influence each hair vertex p (with the exception of the hair root) as $p = p + frizz(t)$, where t is the curve parameter at the hair vertex p (curve parameters are explained in Section 2.2.8.

Kink

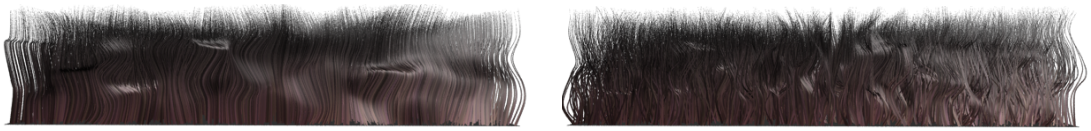
Kink also adds some noise to hair geometry, however in a different way than *Frizz*. The major difference in the computation of *Kink* and *Frizz*, is that in *Kink* the generator calculates the noise vector for each hair vertex separately, therefore the resulting hair looks more crimped or kinkier. Another difference is, that *Kink* does not have the dynamic part (i.e. does not change during animation), since changing *Kink* would look more like each hair vibrating (which is really hard to assign to some effect in reality).

Since I want a different noise vector for every hair vertex, I will not use the hair root position as a noise input, but I will use the current value of each hair vertex position in the world coordinates as the input. The hair vertices will be transformed to the world coordinates using the local coordinate system based on the rest position mesh. The transformed vertices will only be used as the input for the noise function, the noise vectors will be applied on the non-transformed hair vertices.

As with *Frizz* I will use the hair properties to control noise frequency (see Figure 2.22). These properties are **Kink_?_Frequency$_\mathbf{G}$** and the texture property **Kink_?_Frequency$_\mathbf{T}$**, where the ? can be X,Y or Z. So the final input for the ith hair vertex noise vector K_i is calculated as (only a calculation of the X coordinate is shown, Y and Z are similar):

$$input_i.x = \mathbf{Kink\_X\_Frequency_G} \cdot \mathbf{Kink\_X\_Frequency_T}(u, v) \cdot hairVertex_i.x$$

where $hairVertex_i.x$ is the coordinate X of the ith hair vertex in the world coordinates.



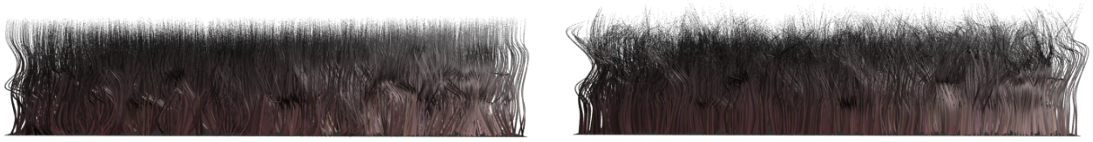
(a) **Kink_X_Frequency** = 0.2 (b) **Kink_X_Frequency** = 2

Figure 2.22: A demonstration of **Kink_X_Frequency** property influence on the hair.

The same way I control the amplitude of the noise (a scaling of a random vector) in *Frizz* deformation, I do it in *Kink* (see Figure 2.23). The hair properties **Root_Kink$_\mathbf{G}$**, **Root_Kink$_\mathbf{T}$**, **Tip_Kink$_\mathbf{G}$**, **Tip_Kink$_\mathbf{T}$** tell me how much should

be the random vectors scaled. By combining the texture and the scalar properties two scalar values are derived: $kink_{tip}$ and $kink_{root}$. Since I use a different noise vector for each vertex, I will have a different function that generates the noise vector for each hair vertex. This also means I can use a linear function for scaling the noise vectors, which I could not use in *Frizz*. The final vector function for the ith vertex is:

$$kink_i(t) = K_i \cdot ((1-t) \cdot kink_{root} + t \cdot kink_{tip})$$



(a) **Root_Kink** = 3 (b) **Tip_Kink** = 3

Figure 2.23: **Root_Kink** and **Tip_Kink** control an amplitude of *Kink* at a hair root and at a hair tip, respectively.

Finally I influence each hair vertex p by the corresponding vector function: $p_i = p_i + kink_i(t)$, where t is the curve parameter at the vertex p_i position (see Section 2.2.8). In Figure 2.24 you can see the difference between the *Kink* and *Frizz* deformation.

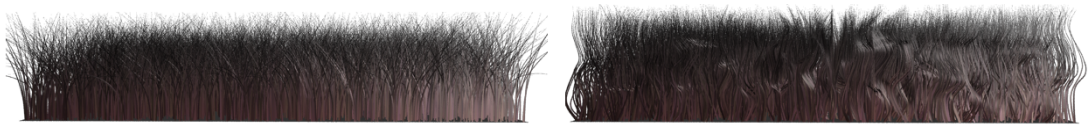


Figure 2.24: In the left image you can see the hair influenced only by *Frizz* and in the right one I have the hair with only applied *Kink*.

2.2.6 Hair color and other parameters

This section talks about selecting color, opacity and width of a single hair. As I have said in Section 2.2.1 about hair representation, these hair parameters may be defined for each hair vertex; however to make realistically looking hair, it is enough to set these three hair parameters at the hair root and at the hair tip and linearly interpolate them to each hair vertex. The interpolation is done just before the generator output hair to a rendering buffer.

First I will talk about a color parameter. The generator receives the hair color from the user in RGB color space and a renderer (all of the currently supported ones) also expects RGB color as input. It would seem logical to keep the color in RGB, but then I would have to interpolate in RGB color space, which is not very good. For example if I interpolate between yellow color and blue color in RGB space, I am going to lose saturation as I get closer to a middle and then start to gain it again, when I get closer to yellow (see Figure 2.25).This is unnatural and therefore I convert the color to HSV color space, in which I interpolate.

The selected colors for the hair root and the hair tip are stored in HSV color space and converted back in RGB after they are interpolated to each hair vertex.

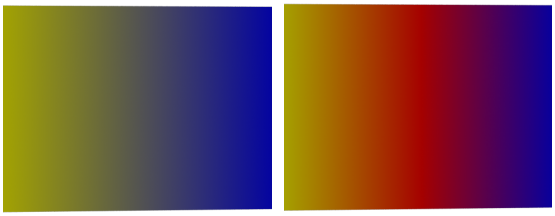


Figure 2.25: An example of a color interpolation in RGB color space and HSV color space.

Interpolating in HSV color space is of course not an ideal solution, but I think it is better than interpolating in RGB space. To convert colors from HSV to RGB color space and back I use methods described in [38]. Both RGB and HSV colors are stored as three scalar values, each RGB component and also the saturation and the value component of HSV is from $[0, 1]$, but hue is an angle so its value is from $[0°, 360°]$. More about RGB and HSV color spaces may be found in [38].

Since hue is an angle, I have to be careful while interpolating it. I have to choose the shortest route on a circle between two hues I am interpolating. From now on I declare, that a counter-clockwise direction represents a decreasing of the hue angle and clockwise represents an increasing of the hue angle. The counter-clockwise distance ccw of hue_1 and hue_2 is calculated by:

$$ccw = hue_1 \geq hue_2 \ ? \ hue_1 - hue_2 : 360 + hue_1 - hue_2$$

and the clockwise distance cw:

$$cw = hue_1 \geq hue_2 \ ? \ 360 + hue_2 - hue_1 : hue_2 - hue_1$$

where ? and : represents the standard c language ternary operator. When I have calculated ccw and cw, I choose which is shorter and interpolate hue:

- $cw \leq ccw$: $hue = hue_1 + t \cdot cw$
- $cw > ccw$: $hue = hue_1 - t \cdot ccw$

where t is the interpolation parameter. After this calculation it may not be true that $hue \in [0, 360]$, but I can fix that by adding or subtracting 360 from hue. Since the only difference between two possible interpolation calculations is that cw is added and ccw is subtracted, I can easily interpolate hue like this:

$$hue_{dist} = cw \leq ccw \ ? \ cw : -ccw$$

$$hue = hue_1 + t \cdot hue_{dist}.$$

Because I use HSV color space, I can apply some perturbation of hue, saturation and value to a hair color. I calculate a random shift in hue ($hueShift$) like this:

$$hueShift = \mathbf{Hue\_Variation_G} \cdot \mathbf{Hue\_Variation_T}(u, v) \cdot 2(\xi - 0.5)$$

where $\mathbf{Hue\_Variation_G}$ and $\mathbf{Hue\_Variation_T}$ are the hair properties supplied by the user and ξ is a random number from $[0, 1]$.

The hue variation property $\mathbf{Hue\_Variation_G}$ can have value from $[0, 180]$, however formula $2(\xi - 0.5)$ ensures that $hueShift \in [-180, 180]$ and therefore hue can perturbate in both directions. The same way I calculate a random shift in value ($valueShift$) and in ($saturationShift$), with use of the $\mathbf{Value\_Variation_G}$, $\mathbf{Value\_Variation_T}$, $\mathbf{Saturation\_Variation_G}$ and $\mathbf{Saturation\_Variation_T}$ hair properties. The resulting $valueShift$ and $saturationShift$ can be any value from $[-1, 1]$.

Before I can apply these shifts, I have to first select non-shifted hair colors. There can be two types of the hair, which differ by the hair color. First is the normal hair, which color is defined at the root and at the tip. But there is also the mutant hair, which is defined by a single color all over the hair curve. The mutant hair may represent for example gray human hair.

To choose whether the hair is normal or mutant I will use a random generator and some hair properties, that define a probability of the mutant hair occurrence. The formula to determine whether the hair is mutant is:

$$\xi < (\mathbf{Percent\_Mutant\_Hair_G} \cdot \mathbf{Percent\_Mutant\_Hair_T}(u, v) \cdot \frac{1}{100})$$

where ξ is a random number ($\xi \in [0, 1]$) and $\mathbf{Percent\_Mutant\_Hair_G}$ and $\mathbf{Percent\_Mutant\_Hair_T}$ are the hair properties defined by the user. The hair property $\mathbf{Percent\_Mutant\_Hair_G}$ can have any value from $[0, 100]$, so I have to transform it to $[0, 1]$ by dividing with 100. From this formula it is immediately clear, that hair is mutant with the probability of

$$\mathbf{Percent\_Mutant\_Hair_G} \cdot \mathbf{Percent\_Mutant\_Hair_T}(u, v)\ \%$$

Now if the hair is mutant, the root and the tip of the hair will have the same color. The color is calculated as a mix of two colors supplied by the user $\mathbf{Mutant\_Hair\_Color_G}$ and $\mathbf{Mutant\_Hair\_Color_T}(u, v)$. The two colors are mixed simply by a component-wise addition. After I have selected the hair color, I can convert it to HSV color space and apply the hue, value and saturation shifts.

The hue shift is applied as follows (and likewise for value and saturation shifts):

$$hue = hue - hueShift$$

After the shifts I have to make sure, that *hue* stays in $[0, 360]$ by adding or subtracting 360 and I also have to clamp *value* and *saturation* to stay in $[0, 1]$. Since the root color and the tip color are the same, no interpolation of color will be done.

If the hair was not selected as mutant, I pick different colors for the tip and the root of the hair. These colors are defined by the self-explanatory properties $\mathbf{Root\_Color_G}$, $\mathbf{Root\_Color_T}$, $\mathbf{Tip\_Color_G}$ and $\mathbf{Tip\_Color_T}$. Then I apply the same steps as in the case with the mutant color with the difference, that root and tip colors are mixed, converted and shifted separately. Since the root color and the tip color may differ, I have to calculate hue_{dist} as specified above and store it until I use it during the color interpolation.

Finally I am done with the hair colors and I can select the opacity and width of the hair. Both the opacity and the width are defined by the user properties that define the value at the hair root and at the hair tip. The opacity of the root

is calculated as:

$$\mathbf{Root\_Opacity_G} \cdot \mathbf{Root\_Opacity_G}(u, v)$$

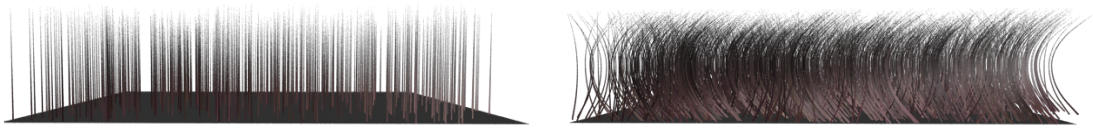
The tip opacity is calculated likewise. The width of root is then:

$$\mathbf{Root\_Thickness_G} \cdot \mathbf{Root\_Thickness_G}(u, v)$$

and the value for the hair tip is calculated likewise.

The selected color, width and opacity for both the tip and the root of the current hair and hue_{dist} are stored for later use. The interpolation of these parameters to the hair vertices is described in Section 2.2.9.

2.2.7 Hair strands



(a) Single hairs (b) Hair strands

Figure 2.26: The difference between single hairs and hair strands consisting of 5 fibers.

This section describes the generation of the hair vertices of a single fiber in a hair strand (see Figure 2.26). The number of fibers in each hair strand is defined by the user property $\mathbf{Multi\_Strand\_Count_G}$. The hair I have generated so far (let us refer to it as the *main hair*) will not be thrown away, since I will need its hair vertices and its curve frame (see Section 2.2.8) to create each single fiber of the hair strand. In other words, I want the main hair to influence the shape of the hair strand. In order to achieve this, I put each fiber vertex P_i on a plane defined by the main hair vertex P_i^M and by the corresponding main hair curve normal N_i^M and the binormal B_i^M as shown in Figure 2.27. I will denote this plane as ρ_i. For this to work, each fiber has to have the same number of the vertices as the corresponding main hair. For each fiber vertex P_i I need to determine its position on ρ_i. Instead of using N_i^M and B_i^M as a position coordinate frame, I will use the distance d_i between P_i and P_i^M and the angle φ_i between N_i^M and the vector $P_i - P_i^M$ (see Figure 2.28), since it will allow me better control over the fiber shape. Each fiber will be generated with the initial distance $d \in [0, 1]$ and the angle $\varphi \in [0, 2\pi]$, from which I will calculate the distance d_i and the angle φ_i of every fiber vertex P_i on the plane ρ_i. The initial d and φ differ for each fiber and are generated by uniformly sampling a unit disc (this unit disc might be deformed into ellipse, as will be discussed later). To uniformly sample a unit disc I use the concentric sampling described in [31, p.665–667].

The hair properties defined by the user will influence the overall shape of the hair strand by indirectly controlling a computation of the fiber parameters d_i and φ_i. Texture hair properties will be always evaluated at texture coordinates of the main hair root, since texture coordinates at the fiber root position are not defined. First I will start with a description of those hair properties that control the distance parameter d_i.

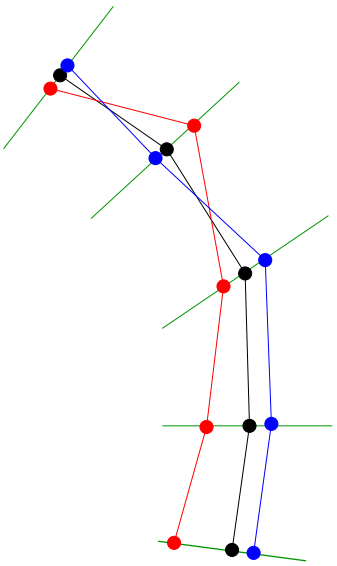


Figure 2.27: The black polyline represents the main hair curve, the green lines are the planes ρ_i. The red and the blue polylines represents fibers from the hair strand.

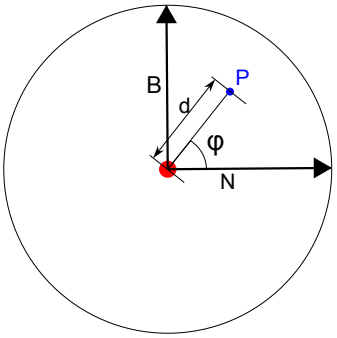


Figure 2.28: This image represents one ρ_i plane. B and N are the binormal and the normal of the main hair curve. The blue point P is the generated fiber vertex, with its distance d and its angle φ.

The hair properties controlling the distance parameter d_i are $\mathbf{Tip\_Splay_G}$, $\mathbf{Center\_Splay_G}$, $\mathbf{Root\_Splay_G}$ and their texture versions (see Figure 2.29). These properties specify the distance d_i on the planes ρ_i defined by the main hair tip vertex, the center vertex and the root vertex. I will define the function $splay(t)$ (t is the hair curve parameter), that will calculate the distance d_i on the remaining planes as follows:

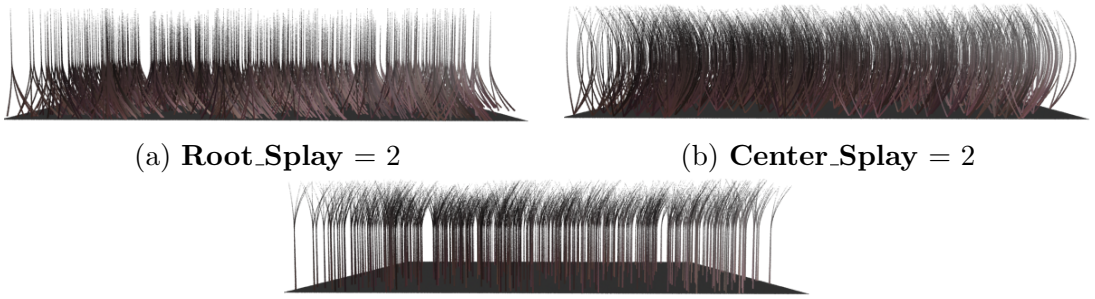
$$splay(t) = 4(t-0.5)^2 \cdot borderSplay(t) + (1 - 4(t-0.5)^2) \cdot centerSplay$$

where $borderSplay(t)$ and $centerSplay$ are defined as:

$$borderSplay(t) = \begin{cases} \mathbf{Tip\_Splay_G} \cdot \mathbf{Tip\_Splay_T}(u,v), & \text{if } t > 0.5 \\ \mathbf{Root\_Splay_G} \cdot \mathbf{Root\_Splay_T}(u,v), & \text{if } t \leq 0.5 \end{cases}$$
$$centerSplay = \mathbf{Center\_Splay_G} \cdot \mathbf{Center\_Splay_T}(u,v)$$

The function $splay(t)$ was defined by error and trial to enable the user good control of a fiber distance from the main hair and to interpolate the fiber distance

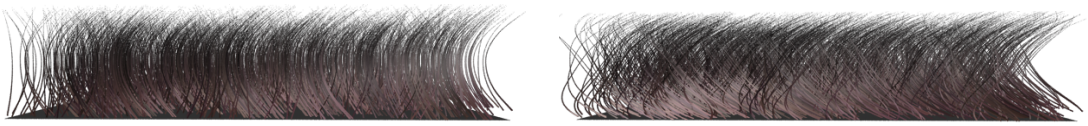
smoothly over the fiber length. Once I have defined $splay(t)$ I can calculate the fiber vertex P_i distance d_i on the plane ρ_i as: $d_i = d \cdot splay(t)$.



(a) **Root_Splay** = 2 (b) **Center_Splay** = 2

(c) **Tip_Splay** = 2

Figure 2.29: A demonstration of an influence of **Tip_Splay**, **Center_Splay** and **Root_Splay** properties on the hair strand.



(a) **Twist** = 0 (b) **Twist** = 5

Figure 2.30: A demonstration of the **Twist** property.

The hair property that only influences the angle φ_i is $\mathbf{Twist_G}$ and its texture version $\mathbf{Twist_T}$ (see Figure 2.30). For easier explanation I will denote the combined twist property as:

$$twist = \mathbf{Twist_G} \cdot \mathbf{Twist_T}(u, v)$$

As this property's name suggests, it defines a twisting of the fiber around the main hair. I achieve this twisting by adding a small angle to the initial angle φ on each plane ρ_i. So each φ_i is calculated from the previous φ_{i-1} by adding some angle. I will have to eventually calculate the fiber vertex position P_i from the distance d_i and the angle φ_i (see Figure 2.28). To do that I need to know $\sin\varphi_i$ and $\cos\varphi_i$. Because calculating these functions takes a lot of time, I will only calculate them for the initial angle φ and then use fast formulas below to get the sine and cosine of φ_i from the sine and cosine of φ_{i-1}.

The *twist* gives a difference between an angle at the fiber root and an angle at the fiber tip. To calculate φ_i, I need to know how much should I add to φ_{i-1}. I will denote that value as $twist_i$:

$$twist_i = \varphi_i - \varphi_{i-1}$$

To calculate $twist_i$ I simply uniformly divide the *twist*, so:

$$\forall i : twist_i = \frac{twist}{N_{vertices}}$$

where $N_{vertices}$ is the number of vertices of each fiber. Since $twist_i$ is same for all i I will denote it as $twist_*$. Finally I can calculate the desired sine and cosine value of φ_i from the sine and cosine value of φ_{i-1} using following formula:

$$\sin\varphi_i = \sin\varphi_{i-1} \cdot \cos twist_* + \cos\varphi_{i-1} \cdot \sin twist_*$$

$$\cos\varphi_i = \cos\varphi_{i-1} \cdot \cos twist_* - \sin\varphi_{i-1} \cdot \sin twist_*$$

where φ_0 is defined as the initial angle φ.

There are three more hair properties that influence fibers in the hair strand. They all have a scalar and a texture variant.

1. **Aspect** (see Figure 2.31): On each plane ρ_i a vertex of every hair in the strand lies on a circle of radius d_i (where it lies on this circle is determined by the angle φ_i). The property **Aspect** can deform this circle into an ellipse. This ellipse will have the same radius as the circle in the normal N_i^M direction and the radius $d_i \cdot \mathbf{Aspect_G} \cdot \mathbf{Aspect_T}(u, v)$ in the binormal B_i^M direction (N_i^M and B_i^M are from the curve frame of the main hair).

2. **Offset** (see Figure 2.32): This property will be applied after the fiber vertex P_i position was calculated and it will offset P_i in the normal N_i^M direction. P_i closer to the fiber tip will be displaced more:

   $$P_i = P_i + \mathbf{Offset_G} \cdot \mathbf{Offset_T}(u, v) \cdot t^3 \cdot N_i^M$$

   where t is the fiber curve parameter.

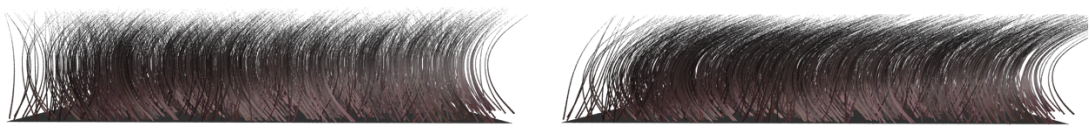
3. **Randomize_Strand** (see Figure 2.33): Finally I scale each fiber vertex in the strand by the factor:

   $$scale = (1 - \mathbf{Randomize\_Strand_G} \cdot \mathbf{Randomize\_Strand_T}(u, v)) \cdot \xi$$

   where ξ is a uniform random number ($\xi \in [0, 1]$). For each fiber in the hair strand I generate a new ξ.

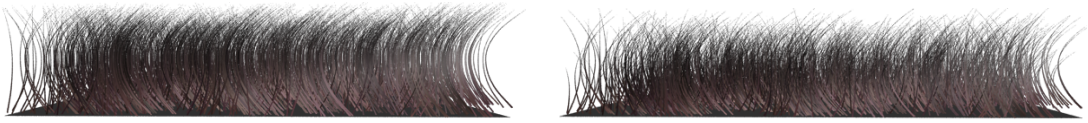
(a) **Aspect** = 0 (b) **Aspect** = 1

Figure 2.31: The **Aspect** property



(a) **Offset** = 0 (b) **Offset** = 5

Figure 2.32: The **Offset** property



(a) **Randomize_Strand** = 0 (b) **Randomize_Strand** = 5

Figure 2.33: The **Randomize_Strand** property

Now that I have described every hair property that influences the vertices of each fiber, I can show the complete equation for the calculation of each fiber vertex:

$$P_i = scale \cdot \left[\begin{array}{c} P_i^M + d_i \cdot (\cos\varphi_i \cdot N_i^M + \sin\varphi_i \cdot \mathbf{Aspect_G} \cdot \mathbf{Aspect_T}(u,v) \cdot B_i^M) \\ + \mathbf{Offset_G} \cdot \mathbf{Offset_T}(u,v) \cdot t^3 \cdot N_i^M \end{array} \right] \tag{2.5}$$

where *scale* has been defined during the property **Randomize_Strand** description and t is the fiber curve parameter.

Figure 2.34 shows a flowchart of a fiber generation.

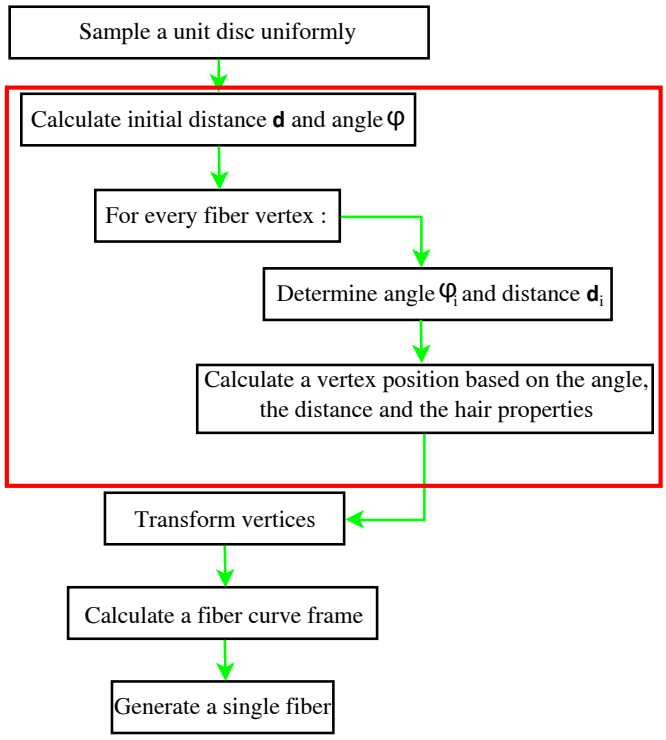


Figure 2.34: A flowchart of the hair strand fiber generation. The red rectangle groups the steps that generate the fiber vertices. These steps were closely described in Section 2.2.7.

After the fiber vertices have been generated, I need to execute several more steps before the whole strand is ready for rendering. First of all I set the color, width and opacity of hair strand fibers from the corresponding hair parameters of the main hair. Afterwards I transform the fiber vertices from the main hair local coordinate frame to the world space coordinates. Then for each fiber I calculate its curve frame. Finally I finalize and output each fiber exactly as I do it with a single hair in non-strand generation (see Section 2.2.9).

2.2.8 Handling the Catmull-Rom spline

As I have said before, hairs generated by my generator are in fact the Catmull-Rom curves. In the following text I talk about some important features of these curves that I use during the hair generation.

Curve parametrization

Several parts of my generator need the hair curve to be parametrized. That means that the curve (let's denote it as C) will behave as a function of the parameter t:

$$C(t) \in \mathbb{R}^3, t \in [0,1]$$

$$C(0) = P_1,\ C(1) = P_N$$

where P_1 is the curve start in $\mathbb{R}^3$ (in my case the first hair vertex) and P_N is the curve end (the last hair vertex). Generally t does not have to be from $[0,1]$, but in my generator it will be. Inside my generator I use the curve parametrization when I want to interpolate any value smoothly along a curve length. For example, the hair color is defined for the hair root and the hair tip and I have to interpolate it for the rest of the curve. If I have the curve parametrized, I can easily define the function *color* which will return the color of the curve at any point:

$$color(0) = \text{hair root color},\ color(1) = \text{hair tip color}$$

$$\forall t \in (0,1) : color(t) = t \cdot color(1) + (1-t) \cdot color(0)$$

Then the point $C(t)$ of the curve will have the color $color(t)$. Since the hair curve is defined by several vertices $P_0, P_1, \ldots, P_N, P_{N+1}$, sometimes during the hair generation I need to know the curve parameter t value that corresponds to the given hair vertex P_i:

$$t : C(t) = P_i$$

This is influenced by how I choose to parametrize the hair curve. The most common and easiest way to parametrize a curve is to use uniform parametrization, which means that the curve parameter change dt between two neighbor vertices defining the curve C is constant:

$$C(0) = P_1,\ C(dt) = P_2,\ C(2dt) = P_3,\ \ldots,\ C((N-1)dt) = C(t) = P_N$$

$$\forall i = \{1, \ldots, N\} : C\left(\frac{i-1}{N-1}\right) = P_i$$

Now it might be confusing what $C(t)$ corresponds to P_0 and P_{N+1} ? The answer is simple: none. As explained in Section 2.2.1, the first and the last vertex are there only to define the curve shape, but the curve does not need go through them. Furthermore, for my hair curves the following statement is always true:

$$P_0 = P_1, P_N = P_{N+1}$$

The uniform parametrization is very simple, however it has one pitfall, it may not be consistent along a curve length. Meaning that the $color(t)$ function, I have defined earlier ,may change faster for some parts of the curve C. Figure 2.35 shows an example of the uniform parametrization which is not consistent along a curve length. More about Catmull-Rom parametrization can be found in [49].

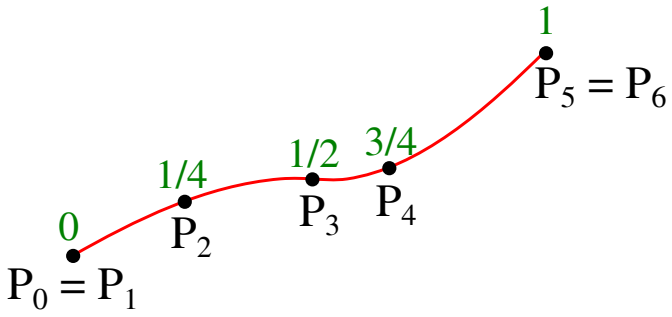


Figure 2.35: The uniform parametrization of the hair curve. The green numbers represent the parameter t value at a given hair vertex.

Cutting the curve

One of the hair properties, that can be set by the user, tells me how much shall each hair be cut. The property value can be between 0 and 1 and corresponds to the curve parametrization. Therefore if user says the curve should be cut at 0.5, the curve will end at $C(0.5)$ (C was defined in the previous section). To be able to cut the hair curve C at any parameter value t_{cut}, I have to be able to recalculate some of the hair vertices positions in a such way that the hair curve $C'(t)$ after the vertices recalculation has the following properties:

$$C'(1) = C(t_{cut}),\ C'([0,1]) = C([0,t_{cut}])$$

where $C([a,b]) = \{C(t)|t \in [a,b]\}$ (and the same for C') and C is the curve before the vertices recalculation.

Before I discuss how to recalculate the hair vertices, I first need to explain how is $C(t)$ calculated for any t. Since I use the cubic Catmull-Rom curve, the curve may be divided to segments $C([a,b])$, where a, b are curve parameters for two neighboring hair vertices: $C(a) = P_i$ and $C(b) = P_{i+1}$. Each segment $C([a,b])$ is then only influenced by the 4 closest hair vertices $P_{i-1}, P_i, P_{i+1}, P_{i+2}$ (see Figure 2.36), where again $C(a) = P_i$ and $C(b) = P_{i+1}$. To calculate the

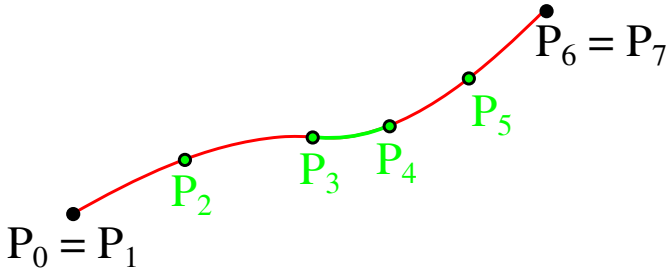


Figure 2.36: The green curve segment is only influenced by the 4 closest (green) vertices.

Catmull-Rom curve $C(t)$ for $t \in [a,b]$ I have to first recalculate the parameter t in such a way, that the new parameter t' will be 0 for $t = a$ and 1 for $t = b$:

$$t' = \frac{t-a}{b-a}$$

and then I can compute $C(t)$ as follows:

$$C(t) = [t'^3, t'^2, t'^1, 1] \cdot M \cdot \begin{bmatrix} P_{i-1} \\ P_i \\ P_{i+1} \\ P_{i+2} \end{bmatrix}$$

where M is the 4×4 matrix which defines the Catmull-Rom basis.

Now that I know how is the Catmull-Rom curve calculated, I can easily calculate were was the curve C cut ($C(t_{cut})$) and use it as the last vertex of the new curve C'. If $C(t_{cut}) \in C[(a, b)]$ then I will define the curve C' after cut using the hair vertices $P_0, P_1, \ldots, P_i = C(a), C(t_{cut}), P_{i+1} = C(b)$ (see Figure 2.37). The curve C' will end at $C(t_{cut})$ however as [12, p. 425] suggests, it may differ from C, so $C'([0, 1]) \neq C([0, t_{cut}])$. This is of course an undesired effect, but it will be hardly noticed since the only two effected segments of the curve are those defined by $P_{i-2}, \ldots, C(t_{cut})$ and $P_{i-1}, \ldots, P_{i+1}$, and their difference from the corresponding C segments will be only subtle (the curve still has to pass through all of its defining vertices).

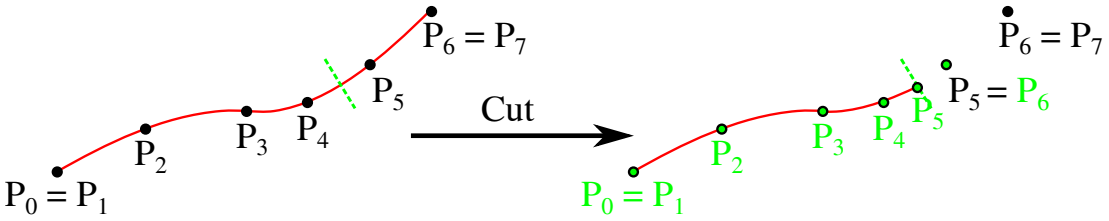


Figure 2.37: The curve is cut between the vertices P_4 and P_5, the curve after the cut is defined by the green vertices

Since cutting the hair curve may result in throwing away several vertices (e.g. P_6, P_7 in Figure 2.37), I may use that to speed up the hair generation and never calculate these vertices. If N is the number of the hair vertices and the curve is cut at t_{cut}, I only need to calculate:

$$N' = \max\left(\lceil t_{cut} \cdot N \rceil + 2, N\right)$$

I have to be careful with one thing though, I can't calculate the curve parameter t corresponding to P_i using the new number of vertices N', since it would result in a wrong calculation of some hair parameters. For example, if I have a curve with a color which is smoothly interpolated from yellow at the curve root to red at the curve tip and I cut this curve in a half, I want the color to go only to something between yellow and red. Therefore I need to calculate the curve parametrization using the original number of the hair vertices N.

Calculating bounding box

For rendering purposes I sometimes need to calculate the smallest possible bounding box (see Section 2.3.2), that contains all generated hair. The most simple way to this is to calculate a bounding box of each hair and join these bounding boxes together.

Calculating a bounding box of the Catmull-Rom curve is not easy, since it does not lie inside a bounding box of the vertices that defines it (see [12, p. 425]). However there are some other curves that have this property, one of them is Bézier curve (see [2, p. 578–584]). This is great, since I can convert the hair vertices to Bézier curve control vertices in a way that will make the resulting Bézier curve identical to the Catmull-Rom hair curve.

Both the Catmull-Rom and Bézier cubic curves are evaluated by the following formula (M is the matrix representing a curve type):

$$C(t) = [t^3, t^2, t^1, 1] \cdot M \cdot \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

where P_i are the vertices defining one segment of the curve C and t is the curve parameter.

If CR is the Catmull-Rom curve matrix, P_i are the original hair vertices, B is the Bézier curve matrix and P_i^B are the Bézier curve vertices, the following calculations can be used to convert P_i to P_i^B:

$$[t^3, t^2, t^1, 1] \cdot CR \cdot \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} = [t^3, t^2, t^1, 1] \cdot M \cdot \begin{bmatrix} P_0^B \\ P_1^B \\ P_2^B \\ P_3^B \end{bmatrix}$$

$$M^{-1} \cdot CR \cdot \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} = \begin{bmatrix} P_0^B \\ P_1^B \\ P_2^B \\ P_3^B \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ -\frac{1}{6} & 1 & \frac{1}{6} & 0 \\ 0 & \frac{1}{6} & 1 & -\frac{1}{6} \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} = \begin{bmatrix} P_0^B \\ P_1^B \\ P_2^B \\ P_3^B \end{bmatrix}$$

which directly leads to this:

$$\begin{aligned} P_0^B &= P_0 \\ P_1^B &= P_1 + \tfrac{P_2 - P_0}{6} \\ P_2^B &= P_2 + \tfrac{P_1 - P_3}{6} \\ P_3^B &= P_3 \end{aligned}$$

I can then easily calculate the hair curve bounding box by converting the vertices defining each hair curve segment and using the converted vertices to create the bounding box.

Curve frame calculation

As I have already mention several times, I need to calculate the hair curve frame at each hair vertex (see Figure 2.38).

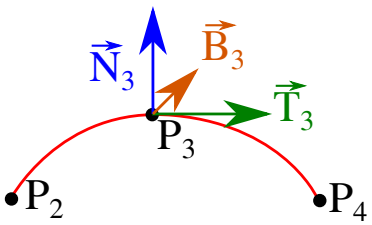


Figure 2.38: The curve frame defined at vertex P_3. The normal is denoted as $\vec{N}_3$, the tangent as $\vec{T}_3$ and the binormal as $\vec{B}_3$.

The calculation of the Catmull-Rom curve tangents at defining vertices is quite easy (see [2, p. 590]). The computation of the tangent $\vec{T}_i$ at the vertex P_i is as follows:

$$\vec{T}_i = \frac{P_{i+1} - P_{i-1}}{2}$$

I will also define the tangents for the first and the last vertex as:

$$\vec{T}_0 = \vec{T}_1,\ \vec{T}_{N+1} = \vec{T}_N$$

Now I have to calculate the curve normals. The normals will define how much a hair fiber surface twists along the curve length (remember that the hair has defined width, so this twist will be visible). To make the hair look realistic I will use the *Rotation Minimizing Frame* method, which makes the twist as minimal as it is possible. To compute *Rotation Minimizing Frame* I will use the fast approximation proposed in [41].

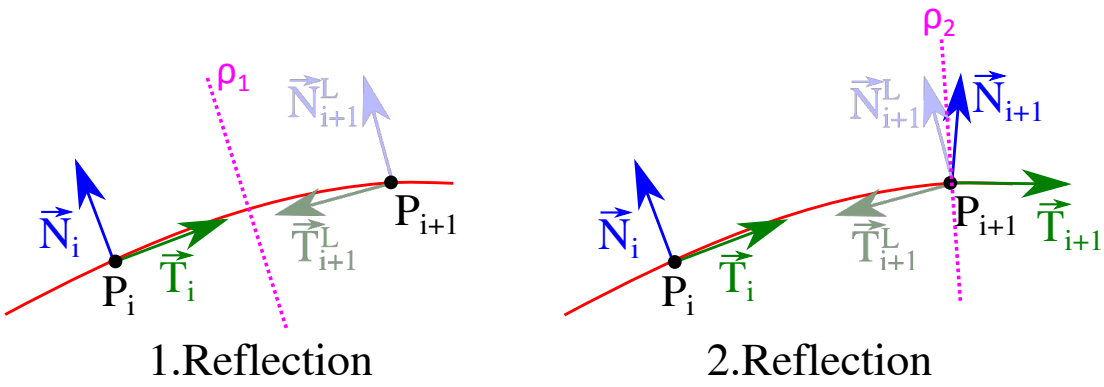


Figure 2.39: The *Double reflection* method used for N_{i+1} computation.

For the computation of the normal $\vec{N}_i$ at the vertex P_i I need to provide the curve vertices P_i, P_{i-1} and the corresponding tangents $\vec{T}_i, \vec{T}_{i-1}$. Furthermore, I also need to know the normal $\vec{N}_{i-1}$, therefore I have to choose the first normal $\vec{N}_0$ myself. I simply choose $\vec{N}_0$ to be equal to a surface tangent at the hair root position. To calculate $\vec{N}_i$ I need to perform two following steps:

1. I define the plane ρ_1, which lies at a middle of points P_i, P_{i-1} and the normal of ρ_1 is vector $(P_i - P_{i-1})$. I use this plane to reflect vectors $\vec{N}_{i-1}$ to $\vec{N}_i^L$ and $\vec{T}_{i-1}$ to $\vec{T}_i^L$.

2. I define the plane ρ_2 in similar manner as ρ_1, however I will use the vertices $P_i + \vec{T}_i$ and $P_i + \vec{T}_i^L$ to define it. I use ρ_2 to reflect the vector $\vec{N}_i^L$ to $\vec{N}_i$.

Because this approximation of *Rotation Minimizing Frame* uses two reflections it is called *Double reflection* method. Figure 2.39 shows the two reflections described above.

Finally when I have both the tangents and the normals of the curve calculated, I can simply calculate the binormal $\vec{B}_i$ at the vertex P_i as:

$$\vec{B}_i = \vec{T}_i \times \vec{N}_i$$

Before I calculate $\vec{B}_i$, I normalize tangents and normals, so $\vec{B}_i$ will be too normalized. The curve frame vectors must be normalized in order for some of the hair generation pipeline steps to work properly (e.g. hair strand generation, see Section 2.2.7).

2.2.9 Finalizing hair

This section describes the last step of my hair generator, before the hair is sent to a rendering buffer. Not all of the hair parameters that have been generated are actually sent to the buffer. Some of the parameters have just served for calculation of other parameters. Each hair is sent to the rendering buffer as the hair vertices, for each of these vertices I sent only these additional parameters: the color, the opacity, the width and the hair curve normal at a hair vertex. I have mentioned in Section 2.2.1, that I need two additional vertices (the first and the last) to define the Catmull-Rom curve. These vertices are not displayed; therefore I will not define for them any of the aforementioned parameters.

Before I can output the hair, I have to perform these three last operations:

1. Remember that in Section 2.2.6 I have said, that I only store the color, opacity and width of the hair root and the hair tip. So here I need to interpolate these parameters to the remaining hair vertices.

2. I also need to cut the hair as the user has specified. How is this exactly done is the topic of Section 2.2.8.

3. Finally there is one more task, I have never mentioned before. I remove some of the generated hair vertices along with their parameters. This helps reduce the rendering time and memory spent by a renderer, since both of these depends on the number of hair vertices. Of course I want the hair look nearly the same as it would look without removing vertices, therefore I need to choose carefully which vertices will be discarded.

Figure 2.40 shows a flowchart of the last hair generation step. Most of the flowchart should be self-explanatory; however I should mention some parts in more detail. The determination of which vertex should be removed will be explained in Section 2.2.9. The interpolation of hair parameters is quite straightforward, I use the hair curve parameter t at the current vertex as the interpolation parameter:

$$value_{current} = t \cdot value_{tip} + (1 - t) value_{root}$$

Where *value* is opacity or width. Remember that the hair color is stored in HSV and its interpolation is more complicated (see Section 2.2.6). I also have to transform the current vertex color to RGB before I send it to the rendering buffer.

Finally I should emphasize, that if I cut the curve, the last vertex that is duplicated and send to the rendering buffer, is not the last vertex of the original curve, but the vertex instead of which I have used the cut position (see Section 2.2.8). As the last, additional hair parameters (such as hair unique index) are sent to the rendering buffer. They are trivially selected by the hair generator and I mention them in Section 2.3.5.

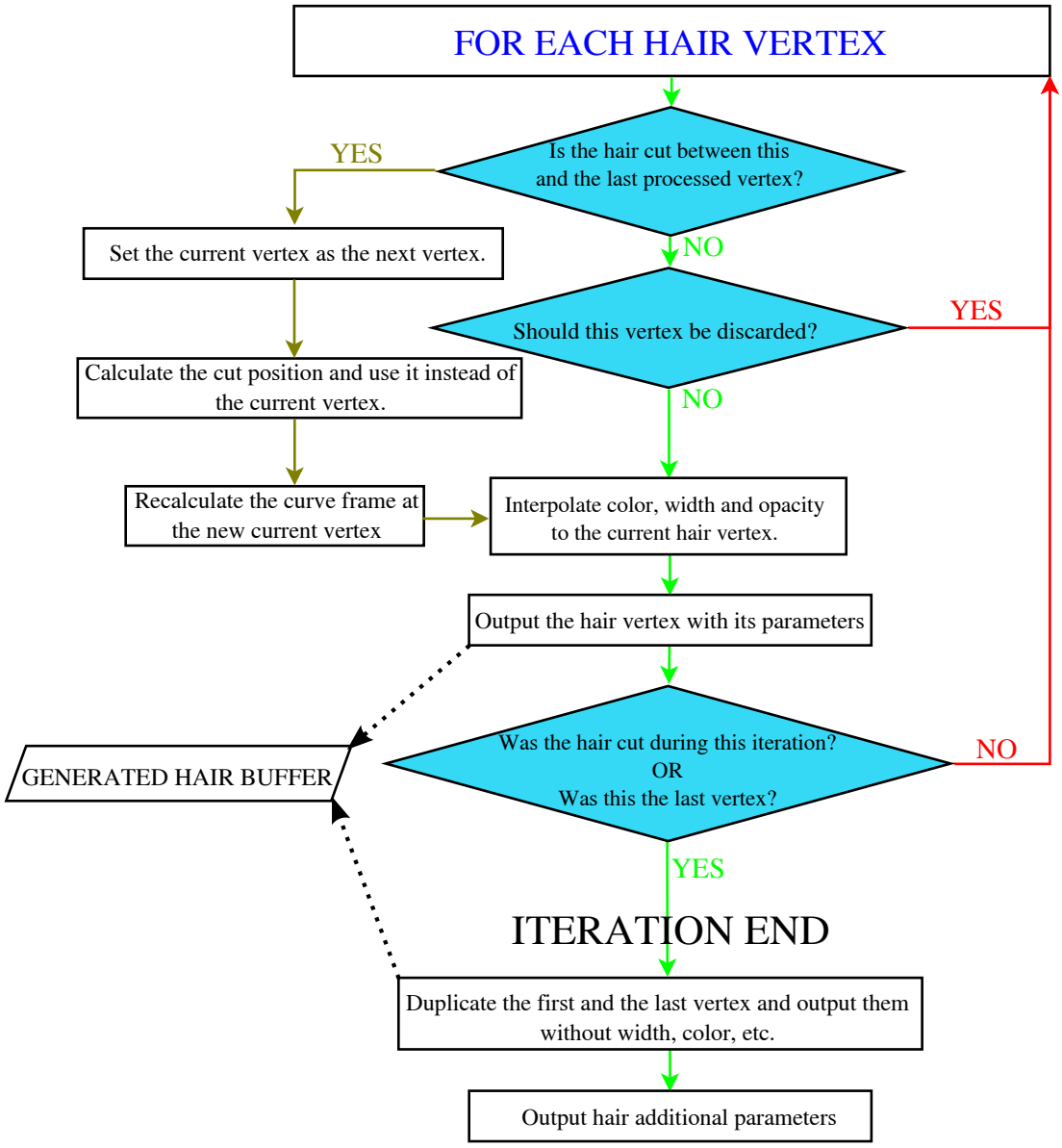


Figure 2.40: The flowchart of the last hair generation step. The last step cut hair, removes unnecessary vertices and outputs hair parameters to the rendering buffer. The nontrivial sub steps are described in text.

Discarding unnecessary vertices

As I have said before, I want to reduce the number of the hair vertices. Of course I want to remove only those vertices, that do not affect the hair curve shape too much, so the curve won't change rapidly after the removal. Therefore I never

remove the first and the last vertex (this also explains why I avoid testing vertex removal if I process the vertex at the cut position – it is the last vertex).

To determine whether the vertex P_i should be removed, I use this simple criterion: if the tangent $\vec{T}_i$ at the vertex P_i is similar to $\vec{T}_{i-1}$ and also to $\vec{T}_{i+1}$, the vertex P_i can be removed since it influence on the hair curve shape is minimal (see Figure 2.41).

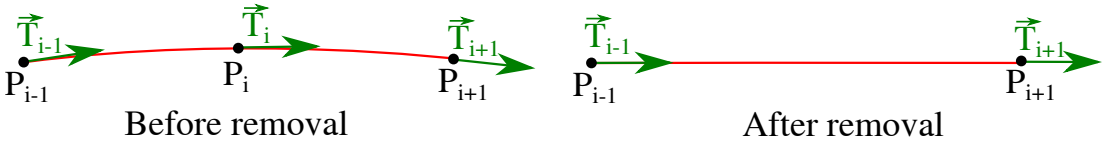


Figure 2.41: The removal of a vertex that has been determined as unnecessary.

I still need to define, what similar tangents means. There is a plenty ways to compare tangents, but I found measuring an angle between two compared tangents as the best. The cosine of the angle between two vectors is calculated easily by a dot product (tangents are normalized), so the vertex removal criterion looks like this:

$$\vec{T}_i \cdot \vec{T}_j > \mathbf{Skip\_Threshold_G}$$

where $\mathbf{Skip\_Threshold_G}$ is the hair property set by the user. The higher the threshold is, the smaller the angle of two tangents must be in order to determine them as similar (remember that the smaller the angle between two vectors is, the higher will be the cosine of that angle).

2.3 Hair rendering

In this section I will describe how is the generated hair rendered by two different rendering systems: RenderMan and OpenGL. RenderMan is used for rendering quality images of scenes and OpenGL is used to display hair interactively during modeling in Autodesk Maya.

2.3.1 RenderMan

I will first talk about what exactly is RenderMan and how can I generate hair in it during render time (see Section 2.3.2), then I will discuss how I move hair data to RenderMan (see Section 2.3.3). How RenderMan renders the generated hair curves is the topic of Section 2.3.5 and finally I will describe how I generate hair for RenderMan in Section 2.3.4.

2.3.2 Rendering with RenderMan

RenderMan is an API defined by Pixar for rendering of 3D scenes. I have picked this API for hair rendering of its wide acceptance and because UPP uses one of its implementations: 3Delight.

The RenderMan interface technical specification [32] allows two ways, how to tell the renderer, what should be rendered. The first is by supplying a text *.RIB* file, which includes RenderMan commands defined in the specification. The

second way is to call c methods from the RenderMan implementation library. These c methods correspond to the commands used inside the *.RIB* file. For my purposes the second way will be more useful.

As I have said before, to render hair I will use the Catmull-Rom curves. The details of how to render such curves with RenderMan are discussed in Section 2.3.5. All you need to know for now is that there can be up to 150 000 hairs on a human head and even tens of millions hairs on an animal body. Rendering such a vast number of the hair curves consumes a lot of computational time and even more memory, therefore a typical way to do that, is to write all needed rendering commands to a *.RIB* file and then send it along with textures and other data files from a computer used by a graphics artist to a "rendering farm"(a cluster of powerful computers used for rendering), where the scene with hair inside the *.RIB* file will be rendered. As told to me by workers from UPP, all the *.RIB* files used to render the whole movie are stored. Since each of these files can reach up to several gigabytes of data, storing them all could be hard to accomplish. The other problem is that hard drives speeds are quite low, so generating and moving these big files is very time consuming.

There is a way to overcome these problems. The original idea is taken from the article [35]. There is a special RenderMan command *Procedural*, that can be written inside any *.RIB* file. Once the scene defined by this file is rendered, the command executes a Dynamic-link library specified with the *Procedural* command, which can generate more scene data during the render by calling c methods corresponding to the *.RIB* commands. These commands are immediately executed and they are never stored to hard drive. This has two advantages: it keeps the *.RIB* files small and it also moves the hair curve generation (which can take a lot of time) from a graphics artist computer to the rendering farm.

There is one issue with the usage of *Procedural*. I need to know the bounding box of the geometry generated by the executed dynamic-link library during the *.RIB* file creation (before the geometry gets generated during the render time). I could of course use some huge bounding box, but as suggested in the previously mentioned article, the smaller the bounding box is, the less memory will be consumed by a RenderMan compliant renderer. The authors of the article also explain that additional computational time is worth it, because insufficient memory is a bigger issue than slower rendering. Their idea how to generate the best bounding box is quite simple: they generate exact hair curves geometry before rendering to calculate the bounding box. Therefore hair curves geometry will be generated twice: before the hair rendering for the calculation of the bounding box and during the rendering. It is important to mention, that each *Procedural* call only generates hair for a single frame of an animation.

To decrease memory requirements even further, I can divide hair to groups called voxels (see Section 2.3.3) and for each voxel call the *Procedural* command. This has another huge advantage and that is the fact, that independent *Procedural* calls are executed in parallel, therefore I can use only one thread inside my Dynamic-link library and leave parallelization to the renderer, which is definitely for the best.

The data needed for the hair generation inside the dynamic-link library will be loaded from temporary files, which are described in Section 2.3.3.

2.3.3 Exporting hair data

Before I can render my hair with RenderMan, I need to export data that will be used by my generator to generate the final shair during rendering. The data are exported for each animation frame separately. To generate the hair I need to export:

- The triangular mesh from which the hair grows. Remember that in Section 2.2.1 I have said, that I need both the hair position on the current mesh (the mesh is usually animated, so by the current mesh I mean the mesh in the currently rendered frame) and the hair position on the rest position mesh (the mesh without any animation applied). Therefore I need to export the current mesh and the rest position mesh.

- Generated hair geometry is mainly influenced by interpolation from the closest hair guides, therefore I need to export hair guides geometry and their positions. As said in Section 2.2.4, I use the rest position to search for the closest guides, so I only export a 3D rest position of each hair guide root. The hair guides may also be animated, so I need to export the hair guides geometry for the currently rendered frame.

- Finally I need all the hair properties, which has been set by the user.

As I have said in Section 2.3.2, I will divide the hair into voxels and generate the hair from each voxel independently on the other voxels. I have mentioned that I do this to help the renderer to decrease memory consumption. However this is achieved only, if the hair is split to the voxels according to hair position in a scene, so the voxels have minimal overlap and the renderer can more easily manage which voxel must be generated and which voxel can already be discarded. Furthermore I will need my random sampling algorithm to generate hair positions only for the currently processed voxel. Finally I also need to make sure, that the hair stays in the same voxel during whole animation, because the generator may generate the hair differently for each voxel in which it lies during an animation.

In order to achieve all of these goals, I calculate a bounding box of the rest position mesh. I create an uniform grid inside the bounding box and I store each triangle of the rest position mesh in one cell of the uniform grid in which the triangle barycenter lies. A uniform grid resolution is based on the user property $\mathbf{Voxels\_Dimensions_G}$. The user has to experiment with the resolution, since for each scene a different resolution is for the best. Each cell of the uniform grid now contains a part of the rest position mesh. From now on I will call these cells the voxels and the hair inside each voxel will be generated only on the triangles stored in the voxel. I also have to store for each voxel triangles of the current mesh. A voxel will contain a triangle of the current mesh only if it contains a corresponding triangle of the rest position mesh.

I have mentioned in the previous section that each *Procedural* call requires a bounding box, which will contain all geometry that will be generated by the *Procedural* call. Therefore before exporting the data I generate for each voxel the hair that grows in it and calculate the bounding box as described in Section 2.2.8. Since I don't need to know other hair parameters than the hair vertices, I don't need to generate them and save a lot of computational time. Furthermore I can execute this procedure in parallel (each voxel is calculated individually).

There is one last thing I need to solve. The user defines the number of hairs for a whole model (property $\mathbf{Interpolated\_Hair\_Count_G}$), not for each divided part. So I need to calculate the number of hairs for each voxel. Back in Section 2.1 I have described the sampling algorithm and determined the probability that one sample will be generated on a single sub-triangle. From these probabilities I can easily calculate the probability of hair being generated in each voxel V:

$$P_V = \sum_{\triangle_i \in V} P(\triangle_i)$$

where $\triangle_i$ is a sub-triangle and $P(\triangle_i)$ its probability. To determine whether $\triangle_i$ lies in V I simply look if its parent triangle lies in V. When I have calculated P_V I can compute the number of hair generated in the voxel V as:

$$P_V \cdot \mathbf{Interpolated\_Hair\_Count_G}$$

So now I have described what data I export, but I still need to tell how I export them. The easiest and most flexible way to export these data is to store them to temporary files, from which my Dynamic-library that executes the hair generation will read them. For each voxel I will store one file with a corresponding part of the rest position mesh and the current mesh. I will also store there how many hairs should be generated inside the voxel. These files are called *Voxel files*. The other data (the hair properties, the hair guides) required by the hair generator are same for all voxels, so I store them inside one file called *Frame file*.

The whole point of generating hair during rendering is that I don't create huge files with hair geometry. *Voxel files* and *Frame file* are very small compared to a file with complete hair geometry; however it is still for the best to compress them. I use zlib[2] library to do so.

Of course I need to tell my Dynamic-library, which files it should use. This is discussed in Section 2.3.4.

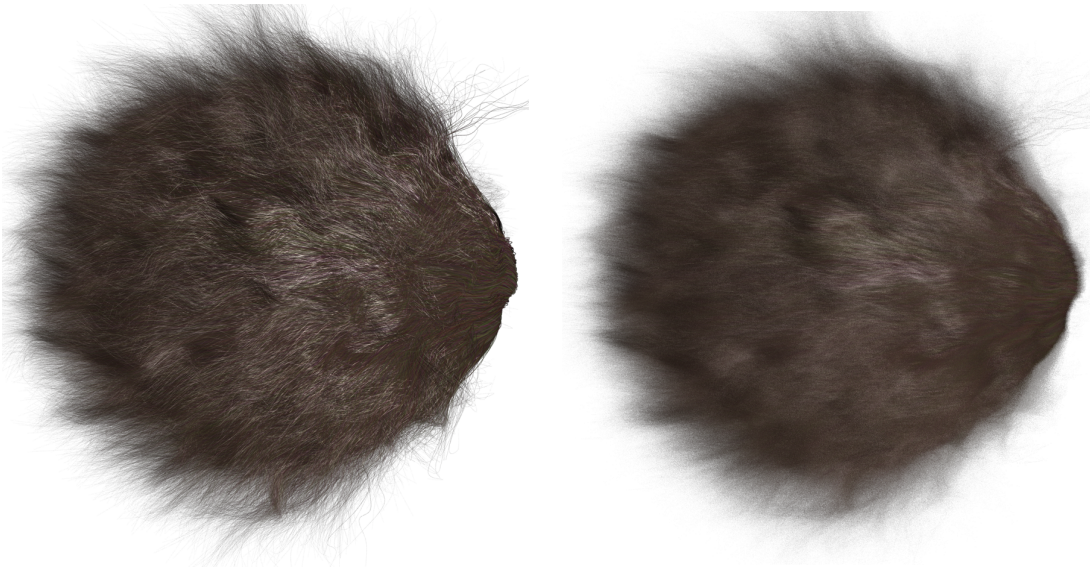
2.3.4 Hair generator for RenderMan

The hair generator for RenderMan is executed inside a Dynamic-library. This library is called by the *Procedural* command from a *.RIB* file and may receive any number of attributes. However these attributes are stored in text form and 3Delight implementation of RenderMan seems to have problem if these attributes exceed certain range. Therefore I use them mainly to tell the library, which data files it should load and use as input for the hair generator.

As I have explained before, each call of the library will generate the hair inside one voxel and only for one frame of an animation. However, there is one exception: if the user specifies that rendering should use deformation motion blur (see Figure 2.42). Any geometry that wants to have deformation motion blur applied in RenderMan must be created in between RenderMan commands: *RiMotionBeginV* and *RiMotionEnd*. Only certain types of geometry are allowed in between those two commands and *Procedural* is not one of them, therefore I have to handle motion blur inside my library. *RiMotionBeginV* receives a number of samples and their relative time compared to the current frame. There

[2] `http://zlib.net/`

must be one and only one RenderMan command (geometry command, translation command etc.) for each of these samples. For more information about motion blur in RenderMan see [32, p. 96–98].



(a) Without motion blur (b) With motion blur

Figure 2.42: A demonstration of motion blur applied on a ball with fur.

My library needs information, how to handle motion blur and from where it should load the hair data. This info is delivered via custom *Procedural* parameters: the number of time samples, their relative time compared to the currently rendered frame and for each of these samples one *Frame file* and one *Voxel file* with the exported hair data.

When the input attributes are handled and the date files are loaded, my library only executes the hair generator and render the hair curves (see Section 2.3.5) for every time sample. Figure 2.43 shows a flowchart of my library.

2.3.5 Rendering curves

The last thing I need to talk about in connection with RenderMan is how can I render the hair curves using RenderMan commands. First I have to tell RenderMan that I want to render the Catmull-Rom curves using the RenderMan command *RiBasis*. *RiBasis* requires four attributes: two matrices and two step sizes. One matrix and step size are used to define a curve type and an interpolation step in the direction of a U coordinate, the other two are used for the same purpose in the direction of a V coordinate. The parameter V changes across a length of the curve and the parameter U across a width of the curve. I will always use the *RiCatmullRomBasis* matrix and *RI_CATMULLROMSTEP* as the step size in both directions.

After I have defined which curves I want to render, I can render them by calling the function in RenderMan API called *RiCurves* (see [32, p. 84–85] for function full specification).

The parameters of the function *RiCurves* are:

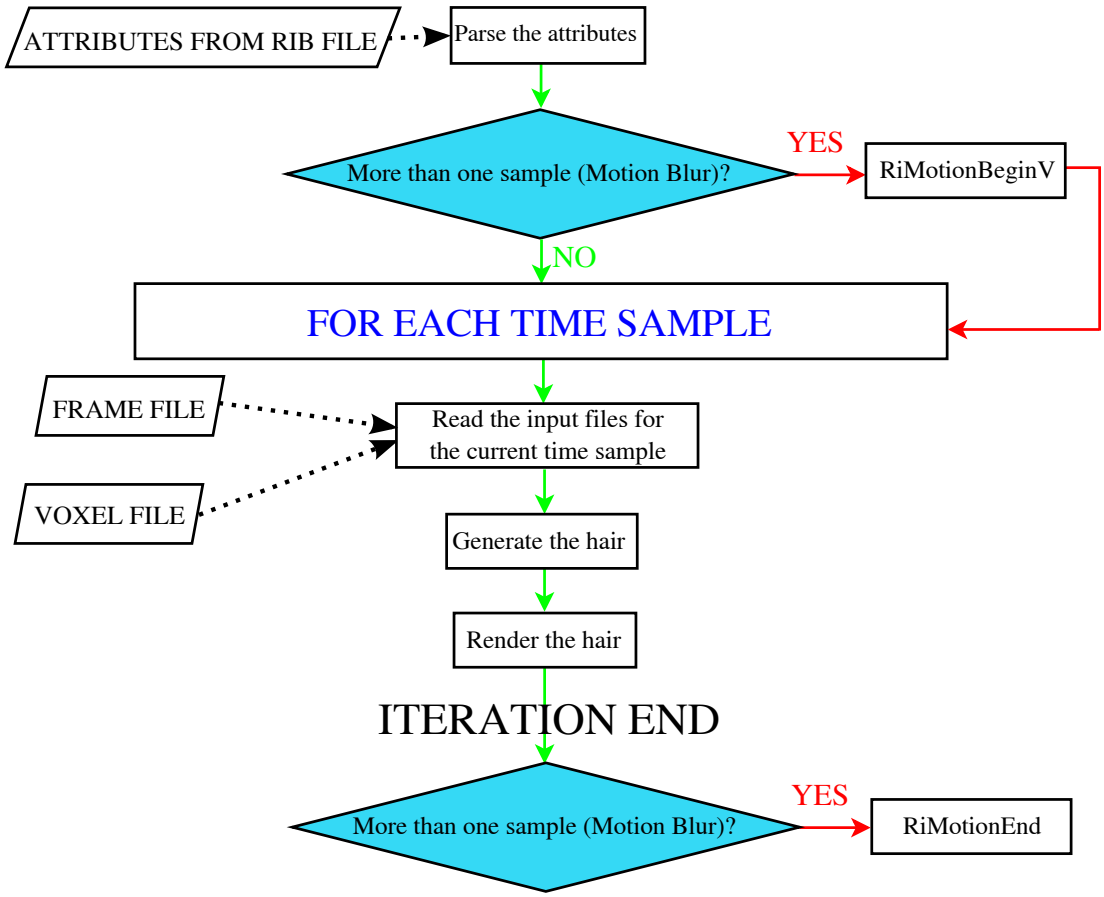


Figure 2.43: A flowchart of my library for hair generation inside RenderMan.

- *type*: Specifies whether linear (*RI_LINEAR*) or cubic (*RI_CUBIC*) curve interpolation will be used. I will use cubic interpolation, since it gives a smoother curve.
- *ncurves*: The number of curves drawn by one call of this function.
- *nvertices*: An array which each element holds a number of vertices defining one curve geometry.
- *wrap*: This parameter says if the curves will be periodic (*RI_PERIODIC*) or non-periodic (*RI_NONPERIODIC*). A periodic curve end is connected with its start. This is an unwanted behavior for hair, therefore I will use the non-peridiodic curves.
- other parameters: The function *RiCurves* may have an unlimited number of parameters. These parameters must come in pairs, where the first one is called a token and specifies what the second parameter is. The second parameter is an array of one or more elements. There are three types of these additional parameters, which distinguish the number of the elements in the second parameter that must be supplied to cover all curves.
  - *constant*: Only one value is needed for all curves.
  - *uniform*: One value for every rendered curve.

– *varying*: There must $nsegs_i + 1$ values per each curve, where $nsegs_i$ is the number of the segments of the ith curve. How is the number of the segments calculated is specified in the *RiCurves* documentation. Since I use a non-periodic cubic curve, the number of the segments of the curve is:

$$nsegs_i = \frac{nvertices_i - 4}{vstep} + 1$$

where $nvertices_i$ is the number of the vertices of the ith curve and $vstep$ is the step size along the V coordinate (which changes across the curve length). Since I use *RI_CATMULLROMSTEP* as the step size, $vstep$ is 1. Finally I can get $nsegs_i = nvertices_i - 3$, so there must be $nvertices_i - 2$ values per each curve.

Among these additional parameters there is one that must be always specified and it represents curve vertices positions. This parameter is *varying*, but it has an exception to the number of values, because there must be $nvertices_i$ vertices specified for each curve. The parameter is announced by the token *RI_P* and the second part of the parameter is *Rt_Float* array, where three consecutive elements represent one vertex position.

- As the last parameter the value *RI_NULL* must be input.

I can use these additional parameters predefined by RenderMan specification. All of them are *varying*.

- *RI_CS*: The parameter that specifies RGB colors of the curve vertices. Each color is represented as three consecutive float values.
- *RI_OS*: Opacities of the curve vertices. Like the color, the opacity is defined by three float values.
- *RI_N*: The curve normals specified at the curve vertices, again each normal is input as three consecutive float values.
- *RI_WIDTH*: A width of a curve at a curve vertex. Each width is represented by a single float value.

These additional properties and the curve vertices directly correspond to the hair parameters that are generated by my hair generator. Furthermore I define additional custom variables that might be used by some specialized hair shaders that compute hair final color (such as shaders that are used in UPP). The variables are all *uniform*, therefore defined once per each hair:

- *HAIR_UV_COORDINATE_TOKEN*: Parameter specifying texture coordinates on the mesh where hair grows. One texture coordinate is represented by two float values.
- *STRAND_UV_COORDINATE_TOKEN*: Texture coordinates on the mesh, where the main hair of a hair strand is located (see Section 2.2.7).
- *HAIR_INDEX_TOKEN*: The unique hair index represented by a single integer.

- *STRAND_INDEX_TOKEN*: The unique hair strand index.

It is important to mention that the user has the ability via the hair property **Calculate_Normals$_G$** to tell, whether the curve normals should be output by the hair generator. If they are not sent to RenderMan, the renderer will calculate them itself. However the normals calculated by the renderer may not be consistent during animation, since they tend to point towards a camera.

2.3.6 Interactive rendering

This section will discuss interactive rendering of hair during modeling in Maya Software. First I will describe in Section 2.3.6 how the interactive hair generation and rendering fits into the Stubble project and then I will talk about rendering the generated hair using OpenGL in Section 2.3.6.

Interactive generation and rendering overview

The interactive hair generation and rendering must react on changes done by the user of the Stubble project. There are many different events that trigger 4 various reactions of the interactive hair generation:

First generation I first generate enough mesh samples for a selected number of the hairs, then I calculate the hair root positions and finally I run the hair generation (including a preparation of the generated hair for OpenGL rendering).

Mesh update I have to recalculate the hair root positions from the already generated mesh samples (the samples are stored from the first generation as barycentric coordinates and triangle identifiers, so they ignore mesh deformations and transformation) and run the hair generation.

Hair properties update I only need to run the hair generation.

Draw I draw the generated hair using OpenGL. The hair has been already generated by any of the previous actions.

To improve performance, the hair generation and the hair root positions calculation can be executed in parallel. The number of the hairs generated in the interactive rendering is set by the user via the hair property **Displayed_Hair_Count$_G$** and is limited to 10000 for performance issues. Which events from the Stubble project triggers which reaction is shown in Table 2.1.

Rendering hair with OpenGL

Rendering a curve in OpenGL is done by tessellating it to a number of lines which are then displayed. This has however two pitfalls. First, tessellating the curve is very expensive and second, lines can only have constant width. So instead of rendering the hair as curves, I will connect the hair vertices with a quad created by two triangles as displayed in Figure 2.44. This should be good enough for the interactive rendering. As mentioned in Section 2.2.1, the hair curves are defined

Event	Reaction
Number of hair changed New model	**First generation**
Model deformation Model transformation	**Mesh update**
Hair properties update Hair guides update	**Hair properties update**
Draw	**Draw**

Table 2.1: The interactive hair generation reaction on different events from the Stubble project. By model I mean the triangular mesh from which hair grows.

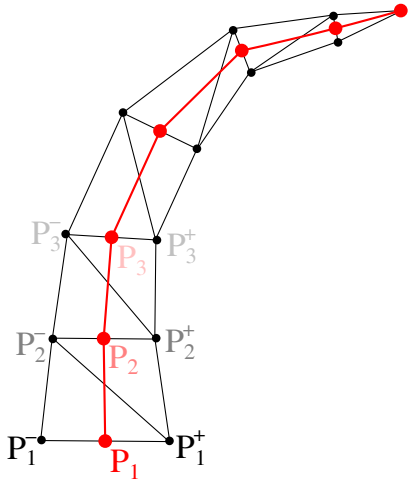


Figure 2.44: The representation of the hair in OpenGL. The red points represent hair vertices and the black triangles are the final representation of the hair in OpenGL.

by set of vertices and the first and the last of them are duplicated. Since I will not render the hair as curves in OpenGL, I will ignore the duplicated vertices.

The size of the quad sides perpendicular to an imaginary line connecting the hair vertices (i.e. a red line in Figure 2.44) will represent the hair width and therefore the width will be linearly interpolated. The only problem I face with this hair representation is that I need to create from one hair vertex P_i two vertices P_i^+, P_i^- of the quad. The logical solution of this problem would be to calculate them as follows:

$$P_i^+ = P_i + \vec{B}_i \cdot W_i, \;\; P_i^- = P_i - \vec{B}_i \cdot W_i$$

where $\vec{B}_i$ is the curve binormal at P_i and W_i is the hair width (again at p). This solution would correspond to RenderMan, where a curve width spans in a curve binormal direction. However my hair generator does not output the curve binormals, therefore I will use the curve normal $\vec{N}_i$ at P_i instead (remember that the interactive rendering is here only to give the user some idea how will the hair look like).

Each quad vertex will also have its RGB color and opacity represented together as RGBA color. I shall get them from the corresponding hair vertex color and opacity.

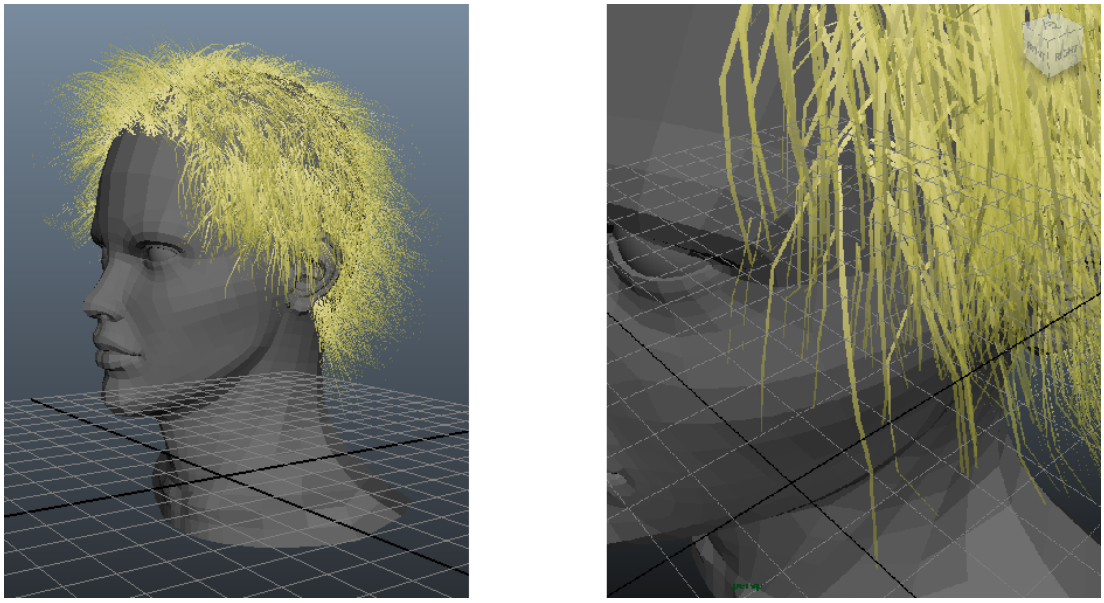


Figure 2.45: The generated hair displayed by OpenGL in the Maya viewport.

Now that I have defined how I transform the generated hair data to colored quads, I need to define how I will handle these quads, so they can be easily rendered by OpenGL. As I have said before, each quad consists of two triangles. I will store all their vertex positions and RGBA colors to a single array of float numbers. Because I need to store 3 component position and 4 component color for each vertex and there are two vertices for every hair vertex, I need to store 14 float numbers per hair vertex. Every time the hair is generated I will convert it to this single array and send it to a graphics card memory through OpenGL Vertex Buffer Objects (see [37, p. 93–100]). Then I can easily render the hair via few commands of OpenGL without the need to call the hair generation again or move any new data to the graphics card. This significantly speeds up the interactive rendering time. Figure 2.45 shows the hair rendered by OpenGL during modeling inside Maya.

3. Implementation

As has been already said, my hair generator is a part of the Stubble project. The Stubble project is written in C++ and was developed in Visual Studio 2010 Professional and compiled for the x86-64 platform using the supplied Microsoft compiler. The Stubble project consists of two dynamic libraries: the first is the plugin to Maya for hair editing and the second is the hair generator library for 3Delight RenderMan.

In this chapter I will only describe those C++ classes of the Stubble project that are used during the procedural hair generation. I will divide this chapter into four sections, first I will shortly describe supplementary classes that are used by the hair generator (Section 3.1), then I will talk about the class that encapsulates the hair generator (Section 3.2) and how is it made to be renderer-independent. Finally I will discuss the implementation of the hair generator library for 3Delight RenderMan (Section 3.3) and the interactive hair generation used in the Maya plugin (Section 3.4)

To make the description of the hair generator implementation clearer, I will always use different text formats for `methods`, *parameters* and CLASSES.

3.1 Supplementary classes

As I have already said, in this section I will talk about different supplementary classes used by my hair generator. Some of these classes were written by other members of the Stubble team, so I will mention here only some of their methods that are used during the hair generation. I will not describe here how these classes are connected to the hair generator that is the topic of the following sections.

3.1.1 Random generators

In this subsection I will describe classes for random numbers generation.

RandomGenerator

RANDOMGENERATOR is a class that serves as an interface for any random number generator. It defines two methods that must be implemented by any random number generator:

- `uniformNumber`() Returns a random float number from $[0, 1]$.
- `randomFloat`(*aMin*,*aMax*) Returns a random float number from the interval [*aMin*,*aMax*].
- `reset`() Resets random numbers generation.

None of these methods is virtual, since that would make generation of random numbers slow. When I need somewhere to use different random number generators, I create a template class or a method with a random generator as a template attribute. The template class or the method will then expect the supplied random generator to have an interface same as RANDOMGENERATOR.

JamesRandomGenerator

JamesRandomGenerator implements the random number generator described by F. James (see [18]), which passes all tests for random number generators and has a period of 2^{144}. The actual implementation has been taken from the JaGrLib project [29]. JamesRandomGenerator only implements the methods defined by RandomGenerator.

HaltonRandomGenerator

HaltonRandomGenerator is a template class that is able to generate numbers from the Halton sequence (see [15]). The used base of the Halton sequence is a template attribute. HaltonRandomGenerator implements the methods defined by
RandomGenerator.

3.1.2 Triangular mesh storage

In this subsection I will describe classes that are used to store a whole triangular mesh and also classes that store a single position on a triangular mesh.

UVPoint

UVPoint is a small class for storing a sample position on a triangular mesh. The sample is stored as an identifier of a triangle, in which the sample lies, and barycentric coordinates U, V of the sample inside the triangle. These three parameter are stored in via the constructor of UVPoint and returned via simple methods `getTriangleID`, `getU` and `getV`.

MeshPoint

MeshPoint is a class for storing a position on a triangular mesh and a local coordinate frame defined by a normal, a tangent and a binormal of the mesh surface at the stored position. All of the mentioned vectors and the position are stored as Vector3D, which is a class for storing a vector or a position and it is able to perform many operations defined on vectors. MeshPoint also stores texture coordinates mapped on the surface at the stored position. It is mainly used to store the hair root position (see Section 2.2.1). All stored parameters are received via a constructor of MeshPoint (with the exception of the binormal, which is calculated from the normal and the tangent) and are accessible by the methods: `getPosition`, `getNormal` etc (see the source for the full list).

Furthermore, MeshPoint can be written and read to/from a binary stream via operators `>>` and `<<` (If MeshPoint is to be read from the binary stream, it can be created using its default constructor with no parameters). MeshPoint is sometimes used to store only the position without the normal, the tangent etc. For that purpose it has a special constructor and also `importPosition`(*aStreamIn*) which reads the 3D position from the binary stream *aStreamIn*.

MeshPoint has several additional methods that can come in handy when dealing with the local coordinate frame stored in MeshPoint:

- `toWorld`(*aLocalVector*) Transforms the vector *aLocalVector* from the local coordinates to the world coordinates. The local coordinate frame is defined by the vectors and positions stored by MeshPoint.

- `getLocalTransformMatrix`(*aLocalTransformMatrix*) Receives the matrix *aLocalTransformMatrix* as a reference and makes it to be a transformation matrix from the world coordinates to the local coordinates.

- `getWorldTransformMatrix`(*aWorldTransformMatrix*) It is nearly same as the previous method, however it creates an opposite transformation: from the local coordinates to the world coordinates.

Triangle

Triangle is a class for storing one triangle. Triangle is represented by its three vertices, which are in fact MeshPoint classes, therefore for each vertex I store more than just its position. The vertices are received via a constructor or can be read from a binary stream using the specialized constructor `Triangle`(*aInStream*, *aCalculateDerivatives*). *aInStream* is the binary stream and *aCalculateDerivatives* tells the constructor, whether it should calculate derivatives (the derivatives will be explained shortly). Triangle can also exports its data to the binary stream via the method *exportTriangle*(*aOutputStream*). The three vertices can be accessed via methods *getVertex?*(), where ? can be 1, 2 or 3. `getBarycenter`() calculates a barycenter of the stored triangle.

I have already mentioned that Triangle is able to calculate the derivatives. This is done by `recalculateDerivatives`() method. By the derivatives I mean the derivative of the position and the normal on the stored triangle according to the texture coordinates U and V. These derivatives are useful during mesh displacement (see Section 2.2.3). The normal derivatives are accessible via methods `getDNDU`() (the derivate according to U) and `getDNDV`() and the position derivatives via `getDPDU`() and `getDPDV`().

TriangleConstIterator

TriangleConstIterator serves as a simple iterator over mesh triangles. Each triangle is returned as a constant reference to Triangle.
TriangleConstIterator is build over the standard C++ vector iterator and has some specialized methods. Since it is very primitive I will not describe it any further.

Mesh

Finally I will describe the class Mesh, which stores a triangular mesh. Mesh can be initialized in 3 different ways:

- Via the constructor `Mesh`(*aInStream*, *aCalculateDerivatives*), which loads triangles from the binary stream *aInStream* and calculate the derivatives if *aCalculateDerivatives* is set to true (see Section 3.1.2 to know which derivatives I mean).

- Via the constructor `Mesh`(*aTriangles*, *aCalculateDerivatives*), which is the same as the previous constructor, but it loads triangles from the list *aTriangles*.
- Mesh can be initialized by its friend class MayaMesh, which is a proxy class for handling any triangular mesh stored inside Maya.

Mesh can export/import triangles to/from a binary stream via methods `exportMesh` and `importMesh`. Using `importMesh` will throw away triangles already stored in Mesh.

Triangles stored in Mesh can be accessed by several methods:

- `getTriangleConstIterator`() Returns an instance of the iterator TriangleConstIterator, which iterates over all triangles.
- `getTriangle`(*aID*) Returns a triangle as Triangle with the specific id *aID*.
- `getRequestedTriangles`(*aTrianglesIds*, *aResult*) Returns the list of triangles *aResult* whose identifiers are specified in the list *aTrianglesIds*.
- `getTriangleCount`() Returns the number of triangles stored in Mesh.

Mesh can also be used to convert a sample stored in UVPoint to a full position MeshPoint:

- `getMeshPoint`(*aPoint*) Converts a sample *aPoint* to a full position MeshPoint as described in Section 2.2.3.
- `getDisplacedMeshPoint`(*aPoint*, *aDisplacementTexture*, *aDisplacementFactor*) Converts a sample *aPoint* to a full position MeshPoint on a displaced mesh (see Section 2.2.3). The displacement of the mesh is defined by the texture *aDisplacementTexture* scaled by *aDisplacementFactor*.

Furthermore Mesh is able to calculate a bounding box of the triangles it stores and return it via `getBoundingBox`(). I have already mentioned that there is another class for handling meshes called MayaMesh. MayaMesh does not store any triangles; it only works with triangles stored by Maya. MayaMesh defines similar methods as Mesh, but it cannot calculate displaced position (since it is never desired inside Maya) and also it cannot import triangles from a list or a binary stream.

When hair is created on a given model, MayaMesh is connected to this model and copy its triangles from Maya to Mesh. Mesh is then used as the rest position of the model and MayaMesh is used to access the current state of the model (deformed and/or translated) which is stored in Maya. MayaMesh has the method `getRestPose`() which returns an instance of Mesh which holds the mesh rest position.

Inside the RenderMan hair generator library (see Section 2.3.4) Mesh is used to store both the rest position and the current state of the model on which hair grows.

3.1.3 Texture

TEXTURE is used for storing a single image texture. I will describe here only the part that I use during the hair generation. TEXTURE obtains texture data from Maya or it can read it from a binary stream via its constructor `Texture`(*aIsStream*) (`exportToFile`(*aOutStream*) exports data to the binary stream). TEXTURE can hold a single-channel texture or a RGB texture.

The most important methods of TEXTURE for the hair generation are those that return a texture value at a given position:

- `colorAtUV`(aU, aV, *aOutColor*) Returns an interpolated RGB color via its parameter *aOutColor* from the image texture at the position (aU,aV).
- `realAtUV`(aU, aV) Returns an interpolated value from the single-channel texture at the position (aU,aV).
- `derivativeByUAtUV`(aU, aV) Returns a derivation of the single-channel texture at the position (aU,aV) according to the texture coordinate aU. This is useful for mesh displacement (see Section 2.2.3).
- `derivativeByVAtUV`(aU, aV) Same as the previous method, only the returned derivation is according to the texture coordinate aV.

Sometimes I need to access the stored texture data directly. For that reason these methods can be used:

- `getRawData` Returns a pointer to the texture raw data.
- `getHeight` and `getWidth` Returns a resolution of the texture, which is important for raw data handling.
- `getColorCompomentsCount` Returns the number of the texture channels (one or three).

3.1.4 UVPointGenerator

UVPOINTGENERATOR implements the mesh sampling algorithm described in Section 2.1. UVPOINTGENERATOR has three public methods (including the constructor):

- `UVPointGenerator`(*aTexture*, *aMesh*, *aRelativeGridSize*) The constructor executes the preprocessing stage of the sampling algorithm. It first finds the maximum subdivision depth for all triangles from *aMesh* in parallel. Then it calculates the sub-triangle barycentric coordinate look-up table using the private method *computeSubTrianglesPositions*(). After that it subdivides the triangles and calculates their probabilities in parallel. *aTexture* is the required density texture. Finally it calculates the cumulative distribution function and build the uniform grid over it in a single thread. Grid cells count is calculated as a product of *aRelativeGridSize* and a total number of sub-triangles.

- `next`(*aSampleX*, *aSampleY*) Generates a single sample using two random numbers *aSampleX* and *aSampleY*. Supplied random numbers can be generated by any random generator.

- `getDensity`() Returns a sum of sub-triangle probabilities without normalization. This is used during the export of the hair data to calculate how many hairs should be generated in a selected voxel (see Section 2.3.3 for a description of the hair data export and the voxel definition).

3.1.5 InterpolationGroups

Class InterpolationGroups is used to handle the interpolation groups. The interpolation groups are described in Section 2.2.4. The interpolation groups are defined by colors of interpolation groups' texture. The InterpolationGroups constructor receives this texture as an instance of Texture. Inside the constructor the texture raw data are obtained via the Texture method `getRawData` and processed into a different texture representation, which instead of colors stores corresponding interpolation group indices. Color is converted to an interpolation group index by an additional look-up table. This look-up table is filled with different colors as the interpolation groups' texture is processed. Each new color receives a unique index — an interpolation group index.

As have been said in Section 2.2.4, the hair in a different interpolation group can have a different number of the hair vertices. Therefore for each interpolation group I store a number of segments (segments = vertices - 1).

During the hair interpolation I use these methods of InterpolationGroups:

- `getGroupId`(*aU*, *aV*) Returns an interpolation group id from the stored texture at the given texture coordinates (*aU*,*aV*).

- `getGroupSegmentsCount`(*aGroupId*) Returns the number of segments for a selected interpolation group.

- `getMaxSegmentsCount`() Returns the maximum number of segments for all interpolation groups. This is useful when I need to allocate an array for the hair vertices, but I don't know the hair interpolation group yet.

I also use the methods `export/importSegmentsCountToFile` to export/import segments count for every interpolation group to/from a binary stream. Other methods of InterpolationGroups are not used during the hair procedural generation.

3.1.6 RestPositionsDS

RestPositionsDS is used for storing the root positions of the hair guides on the rest position mesh. These positions are split to different groups corresponding to the interpolation groups (each hair guide is in a single interpolation group). Each group is then stored in a KD-Tree. I used these KD-Trees to query for the closest hair guides from a given 3D position during the hair interpolation (see Section 2.2.4). As with any other classes, I will describe here only those methods of RestPositionsDS that are relevant to the procedural hair generation:

- `getNClosestGuides`(*aPosition*, *aInterpolationGroupId*, *aN*, *aClosestGuides*) Returns the *aN* closest guides to the position *aPosition* from the interpolation group with id *aInterpolationGroupId*. The closest guides are returned as a list of guides identifiers and distances to *aPosition*.

- `exportToFile`(*aOutputStream*) Exports RestPositionsDS to the binary stream *aOutputStream*.

- `importFromFile`(*aInputStream*, *aInterpolationGroups*) Imports RestPositionsDS from the binary stream *aOutputStream* and uses the supplied InterpolationGroups instance to rebuild the inner KD-Trees.

Figure 3.1 shows dependencies between described supplementary classes.

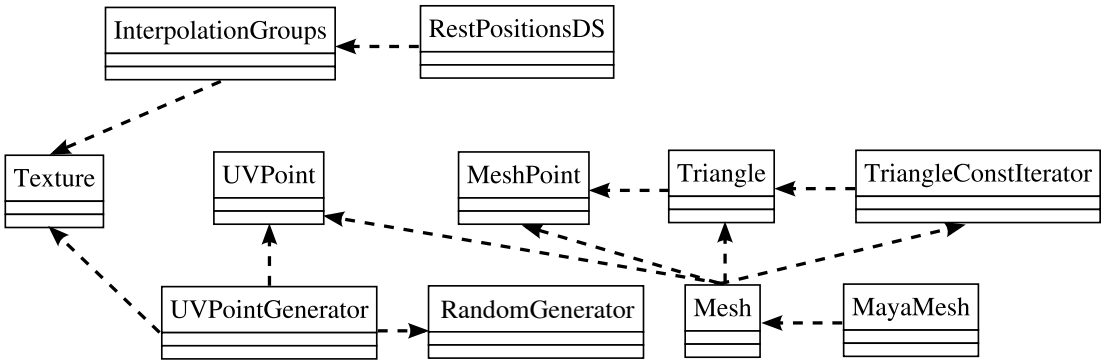


Figure 3.1: Dependencies between supplementary classes. If a class A uses B, an arrow goes from A to B.

3.2 Hair generator

The actual hair procedural generation is done inside HairGenerator (see Section 3.2.4). HairGenerator communicates with the outer world via two other classes. These two classes are different for RenderMan and the interactive rendering and must implement methods defined by two classes (interfaces) PositionGenerator and OutputGenerator (described in the following sections). For performance reasons I want to avoid the usage of virtual functions during the hair generation, therefore the aforementioned classes are never used through the interface, but they are given to HairGenerator as template attributes. The references to existing objects of these classes are then received in a constructor.

In this case PositionGenerator and OutputGenerator are not really needed, but they are useful for showing which methods should be implemented. Furthermore, HairGenerator also requires the HairProperties object (see Section 3.2.3) which stores all properties of the generated hair (scaling, randomization etc.). HairProperties is inherited by classes that handle the hair properties for RenderMan and the interactive rendering. See Figure 3.2 for a scheme diagram.

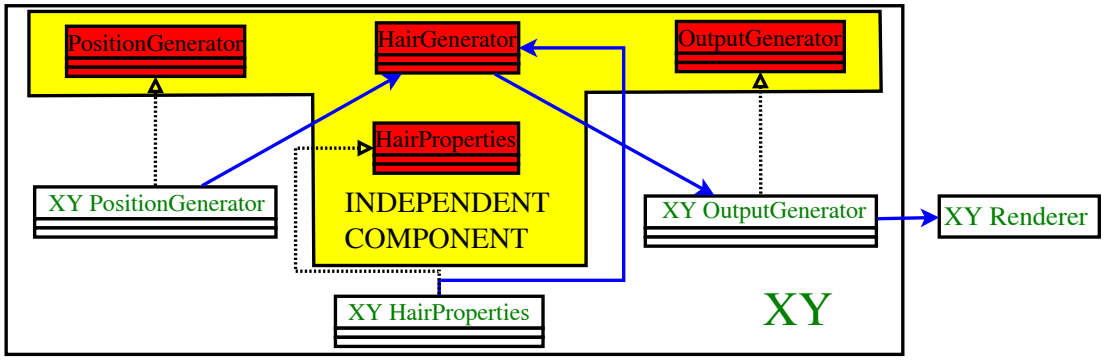


Figure 3.2: A scheme of hair generation interfaces. The blue arrows represent data flow, the dotted arrows point from a derived class to a base class. Independent component represents a part which is same for RenderMan and the interactive rendering, XY represents classes used inside for RenderMan rendering or the interactive rendering.

3.2.1 PositionGenerator

PositionGenerator serves as an interface for classes that generates the hair root positions on a given triangular mesh. A class that implements PositionGenerator will use the hair sampling algorithm defined by the class UVPointGenerator (see Section 3.1.4) to generate samples (UVPoint) on the mesh. Furthermore it needs a mesh storage class like Mesh or MayaMesh (see Section 3.1.2) to calculate MeshPoint from the generated samples. See Figure 3.3 for the PositionGenerator class diagram.

PositionGenerator

```
+generate(out aCurrentPosition: MeshPoint &,
          out aRestPosition: MeshPoint &): void
+generate(out aCurrentPosition: MeshPoint &,
          out aRestPosition: MeshPoint &,
          in aDisplacementTexture:const Texture &,
          in aDisplacementFactor:Real): void
+getHairCount(): uint
+getHairStartIndex(): uint
#PositionGenerator()
```

Figure 3.3: The PositionGenerator class diagram.

PositionGenerator defines these methods:

- `generate`(*) There are actually two methods `generate`(*), both of them serve for the hair root position generation and return a hair position on the current mesh and also on the rest position mesh. One of them has two more parameters, that control mesh displacement (see Section 2.2.3) and the generated hair root position is changed according to it.
- `getHairCount`() Returns the number of hairs that should be generated by HairGenerator.

- `getHairStartIndex`() Returns an index of the first hair. HairGenerator uses this parameter to give each hair a unique index.

3.2.2 OutputGenerator

OutputGenerator (see Figure 3.4 for its class diagram) defines methods that must be implemented by classes that are used as a proxy between HairGenerator and a renderer. These classes store the generated hair data in their buffers and then send them all at once to the renderer. It was already mentioned (see Section 2.2.1), that each hair is represented by the hair vertices and for each of these vertices (with an exception of the duplicated first and last vertex) I also store color, width, opacity and the hair curve normal (the tangent and the binormal are only used during the hair generation and are not sent to the renderer). Furthermore for each hair, HairGenerator also outputs a hair unique index, a strand unique index, texture coordinates of the hair root and finally texture coordinates of the strand main hair. However, only the output generator class for RenderMan will use all these parameters (see Section 2.3.5).

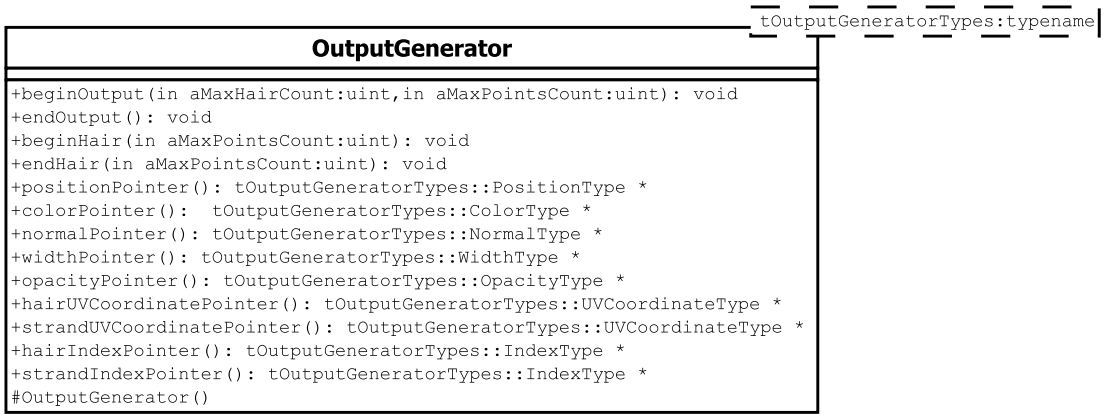


Figure 3.4: The OutputGenerator class diagram.

Since different renderers use different types as input (e.g. a color component may be defined as double or as float), it is for the best if both OutputGenerator and HairGenerator are not dependent on any specific types. To achieve this, OutputGenerator is a template, that receives a single class tOutputGeneratorTypes as a template attribute. This class must define a type for each hair parameter. For example, colors data type must be defined by the type ColorType. Opacities, widths, normals, and positions have also defined separate types like colors, hair/strand indices uses IndexType and the UV coordinates of a single hair or a strand are stored as UVCoordinateType. Any class that implements the interface OutputGenerator, must supply tOutputGeneratorTypes with the types definitions as the OutputGenerator template attribute. HairGenerator will also use these types internally during the hair generation, since it will have access to their definition via OutputGenerator. Figure 3.5 shows a diagram of this special type handling.

For every mentioned hair parameter there is a method in OutputGenerator that returns a pointer to an internal buffer which stores this parameter. These

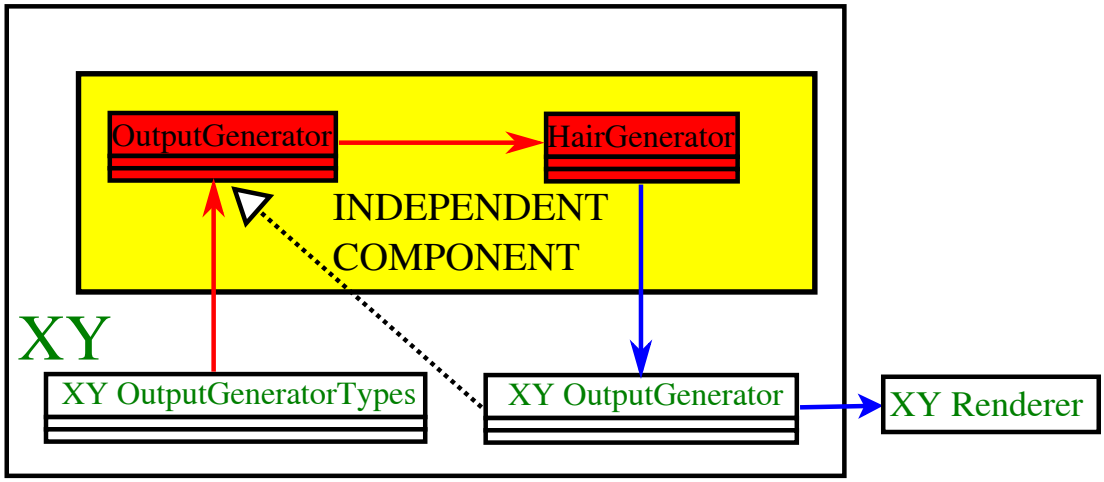


Figure 3.5: The types for XY renderer are defined by class XY OUTPUTGENERATORTYPES which is supplied to OUTPUTGENERATOR and through it to HAIRGENERATOR (red lines). The blue arrows represent data flow and the dotted arrows point from a derived class to a base class.

methods are named `*Pointer`(), where the `*` can be color, opacity, position etc. HAIRGENERATOR will use these methods to output the hair. These methods must be called for each generated hair again, since the returned pointer points at the first element of the buffer that is reserved for the currently generated hair.

The remaining methods defined by OUTPUTGENERATOR control how many hairs will be output, and how many vertices each hair will have. They must be called in the following order (the methods `beginHair` and `endHair` will be called more than once, but a call of the method `beginHair` must always be succeeded by a `endHair` call):

1. `beginOutput`(*aMaxHairCount*,*aMaxPointsCount*) This method announces that the hair generator wants to start outputting the hair. The parameters *aMaxHairCount* and *aMaxPointsCount* are useful for an allocation of the internal buffers. A total count of the outputted hair must never be greater than *aMaxHairCount* and a number of vertices for each hair must not exceed *aMaxPointsCount*. Since HAIRGENERATOR has an unlimited access to the OUTPUTGENERATOR internal buffers, no control of the buffers overflow will be done by OUTPUTGENERATOR.

2. `beginHair`(*aMaxPointsCount*): Announces that the next data incoming through the `*Pointer` methods will be used for a new hair. Similar to the previous method, *aMaxPointsCount* serves only as an upper limit of the hair vertices count.

3. `endHair`(*aPointsCount*): This method must be called when all data for a single hair was sent to the output generator. *aPointsCount* is the number of the output vertices.

4. `endOutput`(): By calling this method, I tell to the output generator that all hair has been output. This usually causes that the hair data are sent from the internal buffers to the renderer.

3.2.3 HairProperties

As I have stated before, the class HairProperties is used to store the hair properties needed for the hair generation, but it is never used on its own, since the hair properties are read-only outside the class HairProperties and their values can only be set or changed by classes deriving from HairProperties. Therefore for RenderMan and the interactive rendering I have special classes for storing the hair properties, which inherit from HairProperties. HairProperties is directly used only by HairGenerator.

Most of the stored properties are textures or scalar/color values and have already been described during the description of the hair generation. I will mention here only those methods that are different from the rest of the properties.

- `getInterpolationGroups`() This method returns InterpolationGroups (see Section 3.1.5) which stores interpolation groups.
- `getGuidesSegments`() This method returns a list of all hair guides. Each hair guide is represented as a list of vertices. These hair guides vertices are used during the interpolation described in Section 2.2.4.
- `getGuidesRestPositionsDS`() This method returns RestPositionsDS (see Section 3.1.6) that is used during the hair interpolation to search for identifiers of the closest hair guides.

3.2.4 HairGenerator

HairGenerator is a class that encapsulates the hair generation as it was described in Section 2.2. HairGenerator receives by its constructor a reference to a class which implements the interface PositionGenerator and to a class which implements the interface OutputGenerator. These references are stored for later use. HairGenerator has only two methods for communication with other classes:

- `generate`(*aHairProperties*) This method executes the hair generation. The generator uses the properties stored inside the HairProperties instance *aHairProperties*. The hair root positions are generated using the stored reference to a descendant of PositionGenerator and the sgenerated hair is sent to a descendant of OutputGenerator. The hair generation is split into several steps, each of these steps is represented by a private HairGenerator method for higher readability. Since these methods closely correspond to the already discussed hair generation steps, I will not discuss them here.
- `calculateBoundingBox`(*aHairProperties*, *aHairGenerateRatio*, *aBoundingBox*) This method is similar to `generate`, it also generates the hair; however it never outputs any hair via OutputGenerator, it only calculates the hair bounding box as it was describe in Section 2.2.8. The calculated bounding box is returned via *aBoundingBox*. *aHairGenerateRatio* influences how many hairs should be generated. PositionGenerator returns via its method `getHairCount`() the number of the hairs to be generated. Here I multiply the returned number by *aHairGenerateRatio*. This

can be used to limit the number of the hairs needed to calculate the bounding box.

3.3 RenderMan support

As I have already said in Section 2.3.2, to enable rendering the hair with RenderMan, I need to do two things: first I need to export the hair properties to the temporary files and I also need to write a library that will create the hair from the temporary files during rendering. Classes that are used to export the hair are discussed in Section 3.3.1 and the library design is the topic of Section 3.3.2.

3.3.1 Exporting hair data

I have already talk about exporting the hair data in Section 2.3.3, so here I will only discuss classes that are used for exporting the data and connections between them.

SimplePositionGenerator and SimpleOutputGenerator

The classes SimplePositionGenerator and SimpleOutputGenerator implement the methods defined by PositionGenerator and OutputGenerator. SimpleOutputGenerator only declare the methods, but leave their implementation empty, since it will never receive any data from HairGenerator. On the other hand, SimplePositionGenerator must generate the hair root positions as it was specified in Section 3.2.1. SimplePositionGenerator receives through its constructor the rest position mesh, the current state of the mesh, the number of the hairs and the first hair index. The implementation of the methods defined by PositionGenerator is similar to RMPositionGenerator (see Section 3.3.2), therefore I will omit its explanation here.

Voxelization

Section 2.3.3 talks about dividing hair into groups called voxels. This is a task of the class Voxelization. I will briefly describe its important methods and constructor, since most of the voxelization itself has already been discussed.

- `Voxelization`(*aRestPoseMesh*, *aDensityTexture*, *aResolution*) A task of the constructor is simple. It divides triangles of the rest position mesh (represented as an instance of Mesh) to the voxels. The voxels are cells of the uniform grid build over the mesh triangles (here comes handy the method `getBoundingBox` of Mesh). The uniform grid resolution is given by the parameter *aResolution*. Furthermore, for each voxel I create an instance of UVPointGenerator (for that I need *aDensityTexture*) and of JamesRandomGenerator. I will use them later for the calculation of a voxel bounding box. I also have to remember identifiers of the triangles that lie inside a voxel. If any of the constructor parameters changes, Voxelization must be reconstructed.

- `UpdateVoxels`(*aCurrentMesh*, *aHairProperties*, *aTotalHairCount*) In this method I calculate a bounding box of every voxel. First, for each voxel I will calculate how many hairs will be generated in it and the first hair index for each voxel, which will be used by the hair generator to give the hairs in the voxels unique indices. The rest of `UpdateVoxels` is parallelized, each voxel is handled individually. Before I calculate the bounding box, I need to split the current state of the mesh *aCurrentMesh* to the voxels. This is done easily, since I have stored the identifiers of the triangles that lie in the currently processed voxel and *aCurrentMesh* is the instance of MayaMesh, which has a handy method `getRequestedTriangles` that returns triangles when supplied a list of identifiers. When I have this, I can construct instances of SimpleOutputGenerator and SimplePositionGenerator and use them to create an instance of HairGenerator. Finally I execute the method `calculateBoundingBox` of HairGenerator (this method requires *aHairProperties*) and return the calculated bounding box. `UpdateVoxels` should be called anytime the current state of the mesh changes or any hair property changes.

- `exportVoxel`(*aOutputStream*, *aVoxelId*) This method must be called after `UpdateVoxels`. It exports the voxel data (the number of hairs, the first hair index, the part of the rest position mesh and the part of the current state of the mesh) to the binary stream *aOutputStream*.

MayaHairProperties

MayaHairProperties is a descendant of HairProperties. It is able to receive user interface messages and changed the stored hair properties according to the user actions. It also holds a pointer to the hair guides, so it can return their vertices via HairProperties interface. MayaHairProperties has one method, which is important for me and that is `exportToFile`(*aOutputStream*). This method exports the hair properties (including the hair guides vertices) to the binary stream *aOutputStream*.

HairShape - sampleTime

HairShape is the main class of the Stubble project and represents a hair node inside Maya (a node is a scene object, such as a triangular mesh, a Bézier curve or hair). It is very complex and has many functions, but here I am only interested in its one particular method `sampleTime`(*aSampleTime*, *aFileName*, *aVoxelBoundingBoxes*). `sampleTime` is responsible for exporting the hair data, that will be used to render the hair at one given animation frame (*aSampleTime*). First it exports the hair properties via the MayaHairProperties method `exportToFile`. `sampleTime` receives the parameter *aFileName*, which is used as the exported files name prefix. The file containing the hair properties has the suffix *.FRM*. After exporting the properties, a Voxelization instance is created (sometimes it already exists from a previous `sampleTime` call, so it can be reused). The Voxelization instance is used to create the voxels and calculate their bounding boxes. Each voxel is then exported to an individual file with the suffix *.VXi*, where *i* is a voxel identifier. All the exported files are compressed

using zip format. As a final step, all the voxel bounding boxes are returned via *aVoxelBoundingBoxes*.

CachedFrame

CachedFrame is used for calling the `sampleTime` method of HairShape at requested time samples and writing the *Procedural* commands to a RIB file (see Section 2.3.2. These commands will cause that the renderer will call my library for the hair generation. CachedFrame can be used to create several time samples of the same HairShape instance, these time samples will be used by to create a motion blur effect (see Section 2.3.4). CachedFrame has these important methods and the constructor:

- `CachedFrame`(*aHairShape*,*aNodeName*,*aSampleTime*) The constructor of CachedFrame calls the `sampleTime` method of the supplied HairShape instance. `sampleTime` needs to know how to name the exported data files. I export them to the directory *STUBBLE_WORKDIR/SCENE_NAME*, where *STUBBLE_WORKDIR* is an environment variable defined during Stubble installation and *SCENE_NAME* is the name of the current Maya scene. The prefix of exported files is *NODE_NAME-TIME*, where *NODE_NAME* is the unique name of HairShape instance (the parameter *aNodeName*) and *TIME* is the time of the sample. I store the sample time, the received voxel bounding boxes and the filename prefix for future use.

- `addTimeSample`(*aHairShape*,*aNodeName*,*aSampleTime*) This method can be used to generate another time sample of *aHairShape*. It behaves the same way as the constructor.

- `emit`() This method is responsible for writing the *Procedural* commands to the RIB file. This is done once for each voxel by calling the command `RiProcedural` (see [32, p. 87–92] for its full specification). `RiProcedural` accepts two important parameters:

  - *args*: The arguments for *Procedural* consists of two strings, the first contains the name of my Dynamic-link library, that will generate the hair. The second string consists of parameters that will be given to my library. As has been said in Section 2.3.4, the parameters contains a number of time samples, a name of frame file and a voxel file for each time sample and relative samples time (e.g. if the sample times are 7.5, 8.0 and 8.5, the relative values will be -0.5, 0.0 and 0.5).
  - *mybound*: A bounding box of a voxel. Since I have one voxel bounding box per every time sample, I need to merge them to a single one.

RenderManCacheCommand

I have already mentioned that I use the 3Delight implementation of the RenderMan API. 3Delight is not only a standalone renderer, but also a plug-in for Maya[1]. This plug-in allows rendering of scenes directly from Maya. To allow any custom

[1] `http://www.3delight.com/en/index.php?page=3DFM_overview`

scene node (such as the Stubble hair) to be rendered, a special cache command with a given interface must be created for the node. RenderManCacheCommand represents this command for the Stubble hair. It implements the Maya interface MPxCommand, which allows it to behave as a Maya command and therefore can be executed from a special Maya language: MEL script. I also need to tell to the 3Delight plug-in, that I have defined such a command. This is done by redefining the simple MEL script command *DL_userGetGeoTypeData*, which returns a list of scene node types with associated commands. Furthermore, I need to define the MEL script command *DL_<custom_type>CanUseObjectInstance*, where <custom_type> is the name of the Stubble hair node type. All this command does, is that it returns 0, which tells 3Delight, that commands used to render the Stubble node (*Procedural*) are not simple geometry and therefore cannot be instanced (see [32, p. 94–95] to learn what instantiation in the RenderMan API means)

Finally I can discuss the implementation of RenderManCacheCommand. There are several tasks that cache command must be able to do. They are all defined in the specification of the 3Delight plug-in for Maya (see [1]) and are distinguished by command arguments. I will discuss each of these tasks when I describe RenderManCacheCommand methods.

To implement the MPxCommand interface I need to define two methods:

- `creator`(): This static method creates and returns an instance of my command.
- `doIt`(*aArgumentsList*): This method is called whenever my command is executed from a MEL script. It receives the command arguments as its parameter. I use these arguments to distinguish what task should be executed. For each task defined by the 3Delight specification I have one method.

The methods for handling the 3Delight tasks (the corresponding command arguments are mentioned after a method name in brackets).

- `sampleTime`(-addstep -sampleTime <double> <node>): This method takes the selected <node> in the time frame <double> and stores it to an internal cache. The cache is represented as the standard C++ map, where key is the <node> unique name and the value is a pointer to the class CachedFrame (see Section 3.3.1). One node can be cached more than once if motion blur is switched on and therefore more time samples of one node are needed. If <node> is not yet in the cache, I create a new instance of CachedFrame and store it to the internal cache (and with it the first time sample), otherwise I find the corresponding CachedFrame instance and call its method `addTimeSample`, which stores an additional time sample.
- `remove` (-remove <node>): Removes the selected node from the internal cache (with its time samples).
- `emit` (-emit <node>): Renders all time samples of the selected node by calling the `emit` method of the node's CachedFrame instance.
- `flush` (-flush): Clears the internal cache.

- `list` (-list): Returns names of all cached nodes.

Figure 3.6 shows connections between classes used to export the hair data.

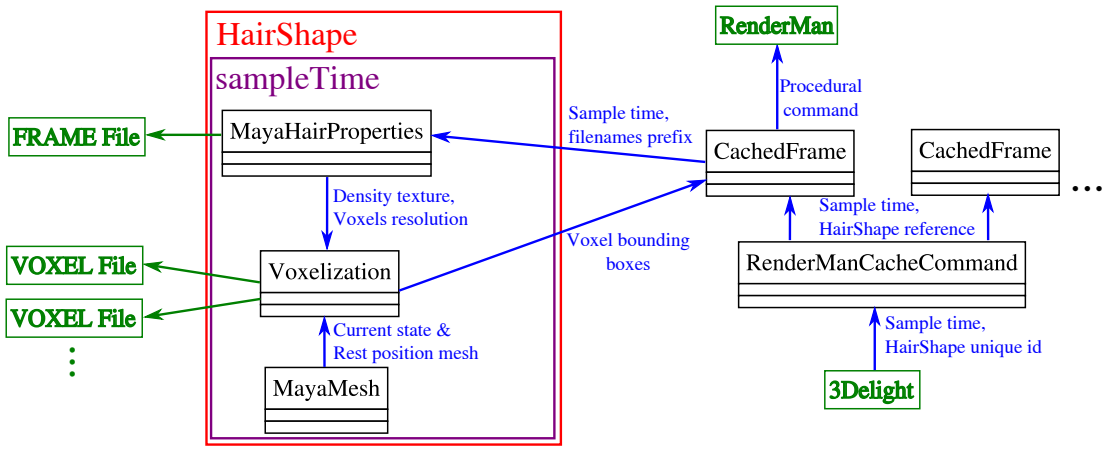


Figure 3.6: A scheme of the hair data export. The blue arrows represent data flow and green arrows represent output to files. Black and red boxes are classes and a purple box represents a method.

3.3.2 Dynamic-Library for RenderMan

In this section I will describe important classes and methods used in the dynamic-link library which generates all hair geometry by calling RenderMan API functions. I need to define three functions that will be used by RenderMan to handle the *Procedural* command.

- `ConvertParameters`(*aParamString*) This method receives *Procedural* parameters in the string *aParamString* and converts these parameters to a binary format. The parameters are mentioned in Section 2.3.4.
- `Subdivide`(*aData*, *aDetailSize*) It is the main method of the library, it receives the binary parameters *aData* created by `ConvertParameters` and it is expected to call RenderMan API functions to generate geometry. The parameter *aDetailSize* is not used by my library.
- `Free`(*aData*) This method has only one task and that is to free memory allocated to store the binary parameters *aData*.

The flowchart of `Subdivide` has already been described in Section 2.3.4, so I will only mention here the implementation of three important classes which are required by HairGenerator and are responsible for most of the work done in `Subdivide`. Figure 3.7 shows connections between classes used inside my Dynamic-Library for RenderMan.

RMHairProperties The class RMHairProperties is used to load all hair data from the compressed *FRAME* file and then distribute them through its base class HairProperties. The contents of the *FRAME* file (and also of the *VOXEL* file) has been described in Section 2.3.3.

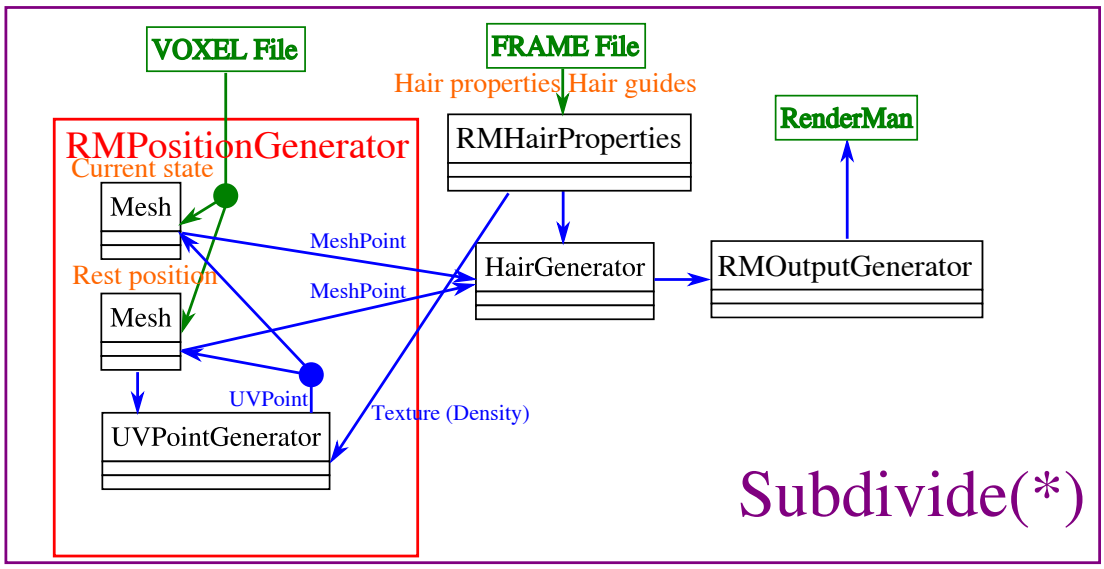


Figure 3.7: A scheme of the hair generation in `Subdivide` method of my DynamicLibrary for RenderMan. The blue arrows represent data flow and green arrows represent input from the files. Black and red boxes are classes and each orange text describes data held in a nearby class.

RMPositionGenerator The constructor of RMPositionGenerator loads the compressed *VOXEL* file that contains all data necessary to generate hair positions on a triangular mesh. From these data and the hair density texture, supplied as a parameter, I construct UVPointGenerator and the current state of the mesh and the rest position mesh (stored as two instances of Mesh).

RMPositionGenerator implements the methods defined by the interface PositionGenerator:

- `generate`(*aCurrentPosition*, *aRestPosition*). This method first generates two random numbers using JamesRandomGenerator, then it uses these samples as input of the UVPointGenerator method `next` to obtain a sample on the triangular mesh. Finally it uses the two instances of Mesh, one for the current state and one for the rest position state of the triangular mesh, to generate the hair root position via the method `getMeshPoint`. `getMeshPoint` receives the generated sample as input and returns the position on the mesh as MeshPoint.

- `generate`(*aCurrentPosition*, *aRestPosition*, *aDisplacementTexture*, *aDisplacementFactor*). This method works the same as the previous method, but it generates a displaced position on the current state of the mesh via method `getDisplacedMeshPoint`,which with addition to the generated sample receives two parameters defining the displacement: *aDisplacementTexture* and *aDisplacementFactor*.

- `getHairCount`() and `getHairStartIndex`() just returns the appropriated values loaded from the *VOXEL* file.

RMOutputGenerator In the class RMOutputGenerator deriving from OutputGenerator I call RenderMan commands that generate the hair. Ren-

dering the generated hair in RenderMan has been already documented in Section 2.3.5. An implementation of the methods defined by OutputGenerator is straightforward and requires no explanation.

3.4 Interactive generation support

In this section I describe classes that are used to generate the hair interactively during modeling in Maya. As it was with the RenderMan rendering, I need three special classes for the interactive rendering that will be used by HairGenerator. One of them, MayaHairProperties has been already mentioned. The whole interactive hair generation and rendering is handled inside InterpolatedHair that will described as the last in this section.

MayaPositionGenerator MayaPositionGenerator implements the methods defined by PositionGenerator (see Section 3.2.1), however it never actually generates positions as RMPositionGenerator do. (see Section 3.3.2) It receives already generated positions (both on the current state mesh and rest position mesh) via its method `set`(*mGeneratedPositions*, *aCount*, *aHairIndex*) along with the hair count (*aCount*) and the first hair index (*aHairIndex*). The `generate`(*) methods then uses one of the already generated positions (that has not been used before) each time they are called. `reset()` can be used to make `generate`(*) methods to start giving away the generated positions from the beginning.

MayaOutputGenerator The MayaOutputGenerator class receives all the hair data (the hair vertex positions, the colors etc.) from HairGenerator through the interface defined by its base class OutputGenerator (see Section 3.2.2). Then it transforms these hair data and displays the hair using OpenGL as it was decribed in Section 2.3.6).

MayaOutputGenerator has the method `draw`(), that causes drawing the received data using OpenGL. `draw` first makes sure that the hair data were send to a graphics card and then renders them. `draw` can be called anytime to draw the hair, without the need to use HairGenerator to generate the hair again.

InterpolatedHair As has been already said, InterpolateHair encapsulates the interactive hair generation and rendering inside Maya. InterpolatedHair class is instanced and used by the Stubble project main class HairShape. Figure 3.8 shows connections between classes used to generate and render the interactive hair.

Since the generation of the hair can be quite slow, I divide the hair into several groups which are generated in parallel (one thread per each group). Each of these groups has its own set of objects for the hair generation stored in the class ThreadData. To generate the hair inside Maya I need to have for each ThreadData an instance of classes MayaPositionGenerator, MayaOutputGenerator and HairGenerator (MayaHairProperties is shared among all hair groups).

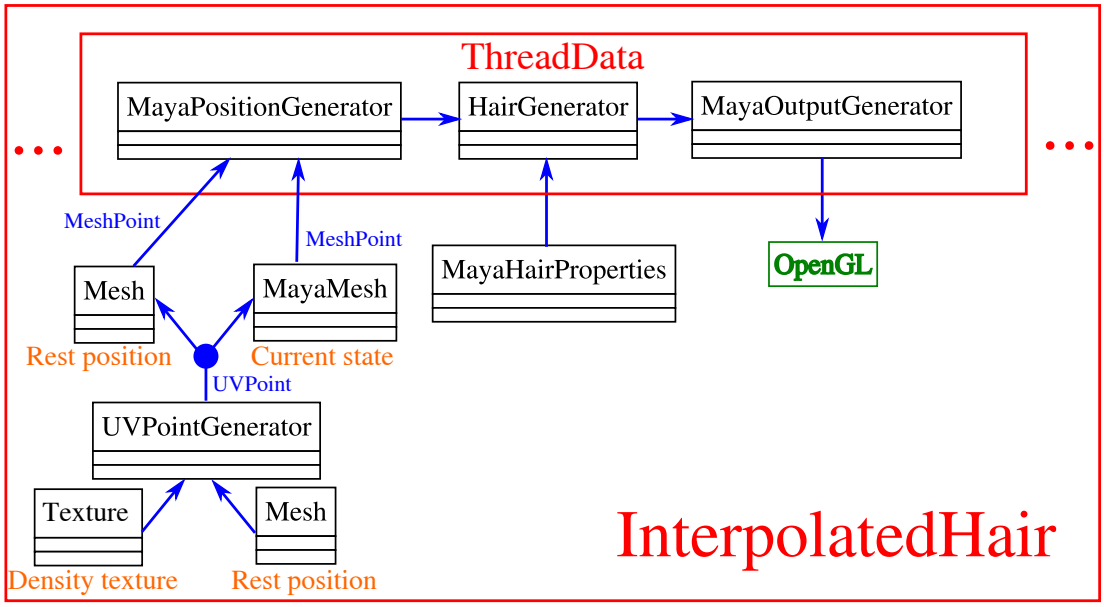


Figure 3.8: A scheme of the interactive hair generation. The blue arrows represent data flow, black and red boxes are classes and each orange text describes data held in a nearby class.

The constructor of InterpolatedHair only creates an empty ThreadData instance for each computing thread. The hair is then generated and displayed as reaction to the user events. These events and reactions on them are described in Section 2.3.6. There are four different reactions, each of them corresponds to one of the following methods:

- `generate`(*aUVPointGenerator*, *aCurrentMesh*, *aHairProperties*, *aCount*): *aUVPointGenerator* is used to generate samples (and I store them for later use) and *aCurrentMesh* makes the hair root positions from them. I split the generated hair root positions uniformly between the hair groups and for each group I send a block of these positions to MayaPositionGenerator via `set`. MayaPositionGenerator will then supply the hair root positions to the hair generator. The number of the generated hairs is defined by *aCount* and is limited to 10,000. Finally I generate the hair by calling the method `propertiesUpdate`.

- `meshUpdate`(*aCurrentMesh*, *aHairProperties*): This method reacts to a deformation or a translation of the mesh surface *aCurrentMesh*. Since I have stored the generated samples, I can recalculate the hair roots position from them using *aCurrentMesh* without the need to use UVPointGenerator. This recalculation may be executed in parallel for each hair group. Finally I call `propertiesUpdate` to generate the hair.

- `propertiesUpdate`(*aHairProperties*): This method generates the hair by calling HairGenerator method `generate` for each hair group in a different thread. I have to supply to this method the hair properties *aHairProperties*.

- `draw`: Renders the previously generated hair. Each group of hair is rendered by calling the `draw` method of MayaOutputGenerator. Because I use

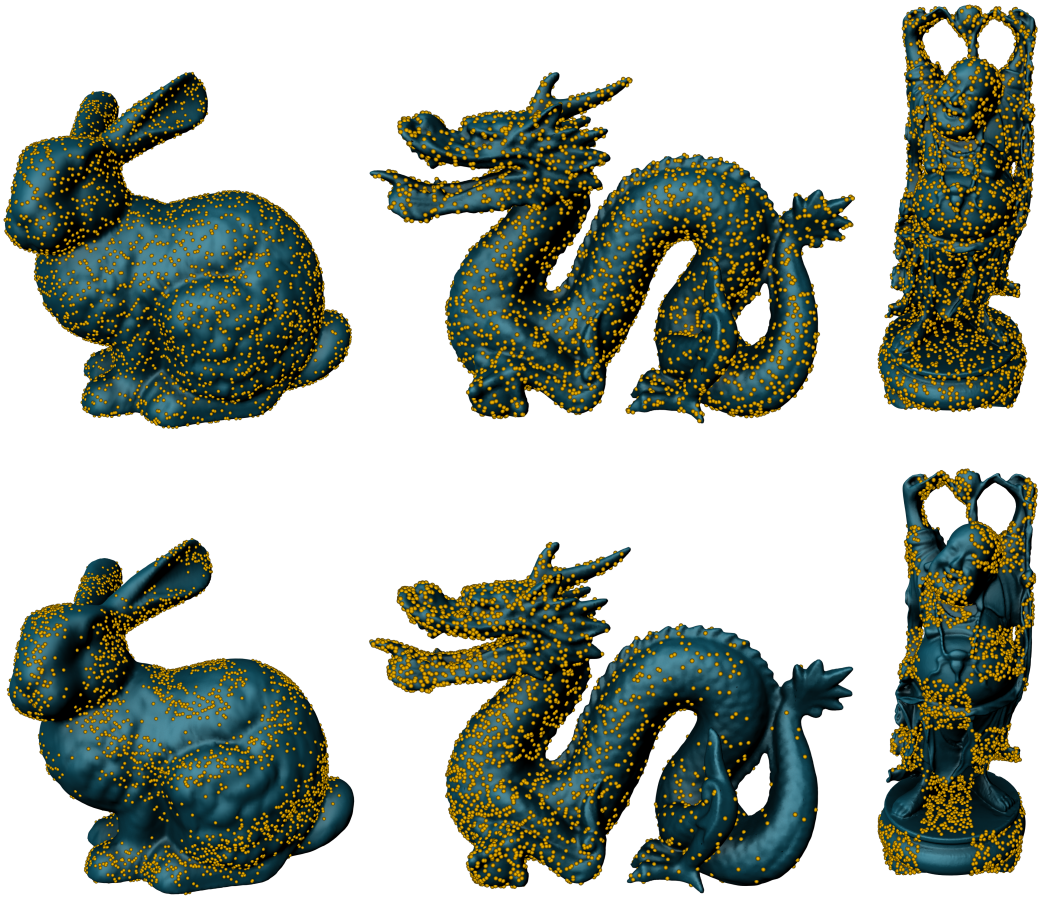
OpenGL to render the hair in Maya, I can't execute rendering of the hair groups in parallel.

4. Results

I will divide the results chapter to two sections. In the first, I will show the results of the sampling algorithm described in Section 2.1 and in the second I will show the results of the procedural hair generation (see Section 2.2).

4.1 Sampling algorithm results

I test my sampling algorithm on 4 cores with 8 threads of a 3.07 GHz Intel Core i7-950 PC with 6 GB RAM running Windows 7 64bit. I use density textures with a resolution of 1024×1024 unless noted otherwise. To make full use of the four available CPU cores, both the triangle subdivision and the sampling are parallelized. Figure 4.1 shows sample distributions generated by my algorithm for the three example models I use in my tests.



Bunny (70k) Dragon (202k) Buddha (1,087k)

Figure 4.1: Three different models with 7,000 point samples distributed uniformly (top row) and according to simple density textures (bottom row). Triangle counts for each model are listed in parentheses. Models are rendered in high resolution.

4.1.1 Uniform Grid Performance

I start the evaluation by investigating the influence of the uniform grid resolution on sampling performance as shown in Figure 4.2. I have used the Bunny model and four different density textures for this test: uniform, checkerboard, Perlin noise, and a HDR environment map. The highest sampling rate is achieved when the number of cells is about four times higher than the CDF domain size (i.e. the number of sub-triangles).

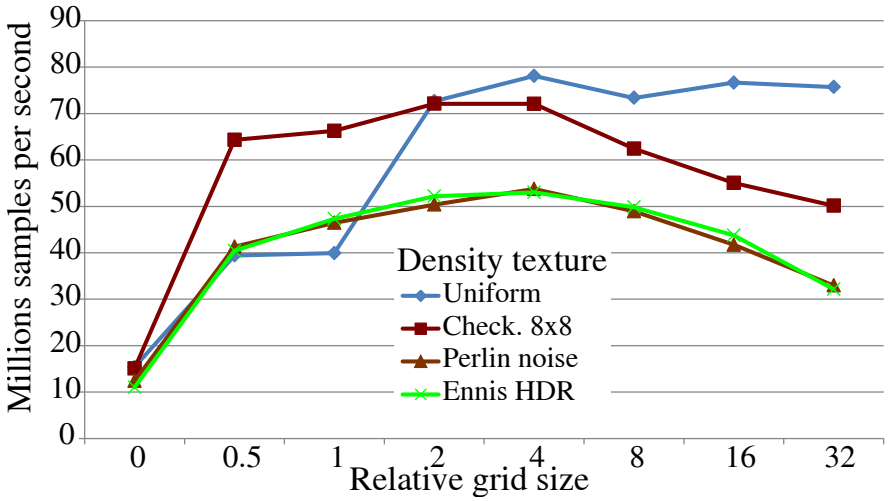


Figure 4.2: Sampling rate of my algorithm for different grid sizes (reported as a ratio of the CDF size).

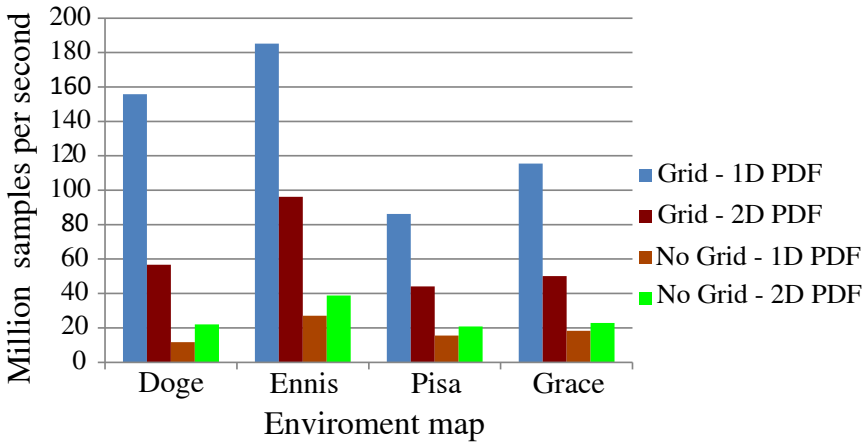
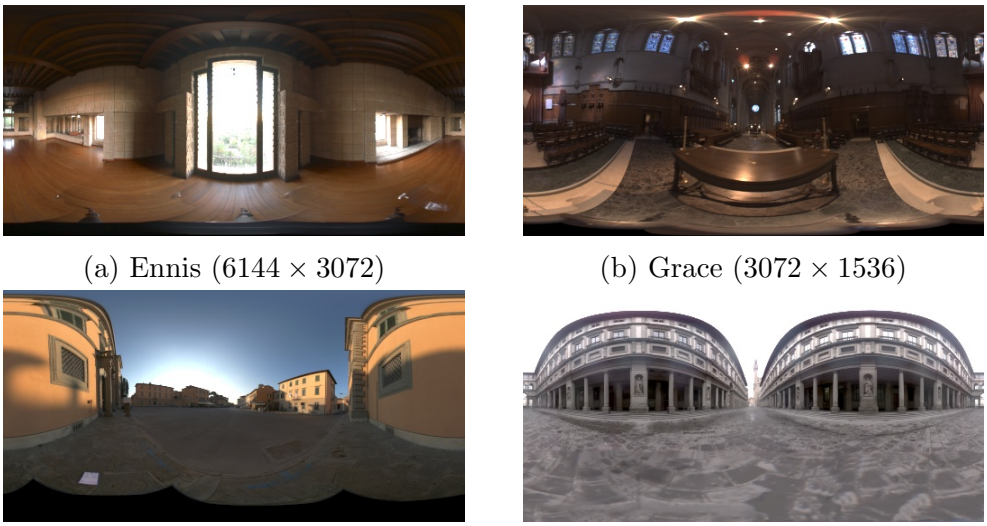


Figure 4.3: Performance of environment map sampling with and without uniform grid-based CDF search.

As an entirely general technique, the grid-based CDF search that I use to speed up the selection of sub-triangles in my mesh sampling algorithm can be used for any other application that requires sampling from a discrete probability distribution. As an example, I could use this technique to speed up computation of lightning from an environment map. Figure 4.3 compares sampling from the environment maps shown in Figure 4.4 with and without the grid. In the tested cases the use of the grid led to a speedup between 5 to 14 when considering the

environment map as a 1-dimensional probability distribution, and between 2 to 3 for a 2D distribution.



(a) Ennis (6144 × 3072)

(b) Grace (3072 × 1536)

(c) Pisa (4096 × 2048)

(d) Uffizi (2400 × 1200)

Figure 4.4: HDR environment maps used in my tests. The resolution of each environment map is listed in parentheses.

4.1.2 Sampling Performance and Memory Consumption

I compare the performance and memory consumption of my mesh sampling algorithm to rejection sampling (as described in Section 2.1.2) because it is the fastest existing alternative. I consider two flavors of my algorithm – with and without the uniform grid optimization described in Section 2.1.6 – and perform the test for a number of different density textures: uniform, checkerboard and 2-dimensional Perlin noise textures with different frequencies, and two HDR environment maps from Figure 4.4 converted to a 1024 × 1024 resolution.

Figure 4.5 plots the performance of the compared algorithms. With uniform density texture, rejection sampling never rejects any samples; therefore the faster sampling rate of my algorithm is only due to the uniform grid. Performance of rejection sampling significantly deteriorates for non-uniform density textures (up to 26 times for tested HDR textures), while my algorithm's sampling rate drops at most by 35% compared to sampling rate with a uniform texture. The drop in performance of my algorithm is mainly caused by the fact, that for uniform texture no triangle need to be subdivided and therefore total number of sub-triangles is much lower than when any highly varying texture is used.

Break-down of the time my algorithm spends on sampling reveals that 2.5% – 51.2% is spent on selecting a sub-triangle by searching the CDF, while random number generation takes 16.8% – 33.5% (using James random generator, see Section 3.1.1), and the rest (i.e. generating sample position in a sub-triangle and transforming it to the parent triangle) takes 32% – 64% of the time. The fact that substantial fraction of the time is spent on random number generation suggests that my algorithm does not leave much space for performance improvement other than micro-optimization.

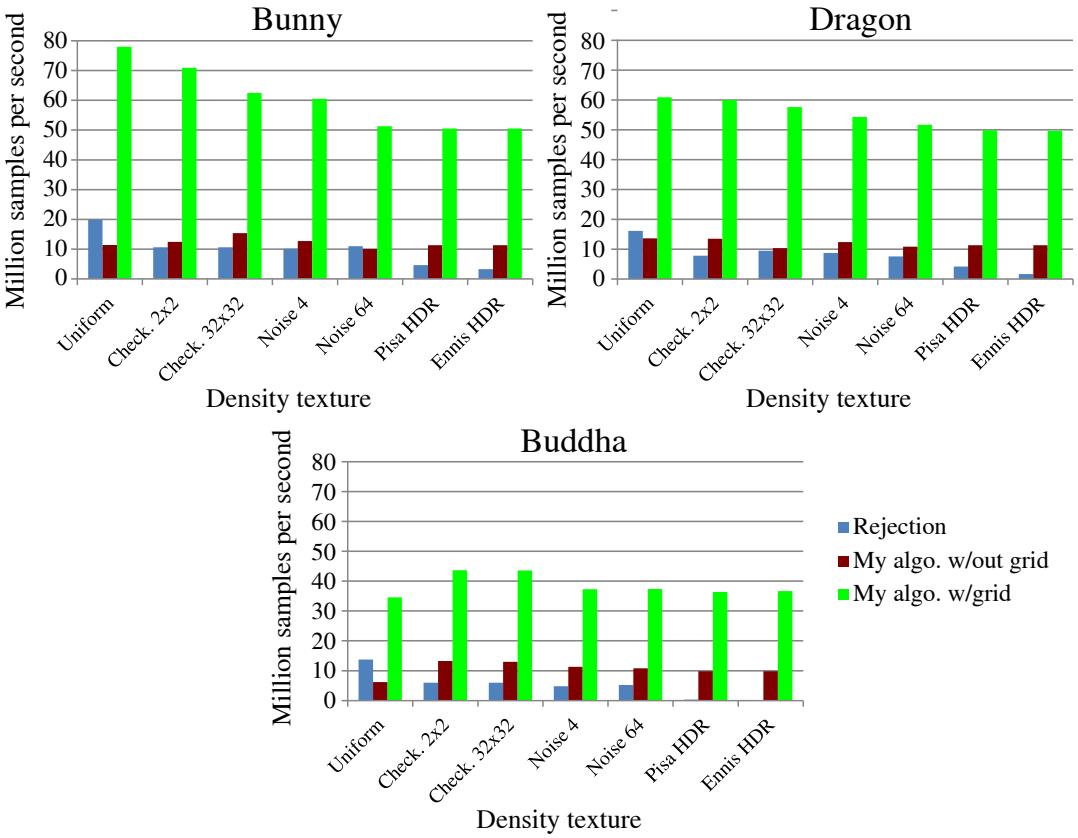


Figure 4.5: Sampling rate (in million samples per second) for rejection sampling and my algorithm (without and with grid). Three different models with a uniform density texture, checkerboard and noise textures with different frequencies, and two HDR textures were used in this test.

Table 4.1 top shows the time spent on the mesh and density texture preprocessing. Rejection sampling has the shortest preprocessing time since it only creates the CDF based on triangle areas. My algorithm needs to subdivide triangles, calculate their probabilities based on the density texture, create the CDF, and build the uniform grid. The grid construction time is negligible $(3 - 4\%)$ compared to the rest of preprocessing. Since triangle count for the Buddha model (1,087k) is higher than the texel count of the density texture (1,024k), only few triangles are subdivided and the preprocessing time for my algorithm is only slightly higher than for rejection sampling.

Memory consumption of rejection sampling and my algorithm is shown in Table 4.1 bottom. Even though my algorithm has higher memory cost in most cases, it is still insignificant with respect to the usual computer's memory size. Both preprocessing time and memory consumption of my algorithm depends on a sub-triangles count, therefore the measured values are lowest for the checkerboard 2×2 texture (where most of the triangles with dark texture part are discarded completely) and highest for highly varying HDR textures.

Finally I test the influence of a density texture resolution on my sampling algorithm. Figure 4.2 shows preprocessing time, memory consumption and sampling speed for the bunny model with an HDR density texture of different resolutions. Increasing the tested density texture resolution two times causes four times

Preprocess [ms]	Bunny	Dragon	Buddha
Rejection sampling	6	11	67
My algorithm w/out grid	$65 - 68$	$67 - 69$	$68 - 71$
My algorithm w/ grid	$67 - 70$	$68 - 72$	$71 - 74$

Memory [MB]	Bunny	Dragon	Buddha
Rejection sampling	4.3	4.8	8.1
My algorithm w/out grid	$2.8 - 17.8$	$4.3 - 17.8$	$14.4 - 17.9$
My algorithm w/ grid	$3.9 - 23.7$	$5.5 - 24.1$	$21 - 25$

Table 4.1: Preprocessing time in milliseconds (top) and memory consumption in megabytes (bottom) for rejection sampling and my algorithm (without and with grid). Same models and textures as in Figure 4.5 were used. The lowest values were measured for the checkerboard 2×2 texture and the highest for the HDR textures.

higher preprocessing time and memory consumption, however sampling speed is decreased at most by 30%. Rejection sampling algorithm does not depend on a density texture resolution at all, but since a density texture resolution sufficient for hair modeling will usually be around 1024×1024, my sampling algorithm dependence on a density texture resolution is only a small disadvantage.

Measured value	512×512	1024×1024	2048×2048
Preprocessing time [ms]	14	70	253
Memory consumption [MB]	5.6	23.7	76.2
Samples per second [10^6]	44.2	39.8	28.2

Table 4.2: Preprocessing time in milliseconds (top), memory consumption in megabytes (middle) and sampling speed (bottom) of my algorithm with uniform grid for the bunny model. The same HDR density texture with different resolutions was used during the testing.

In summary, my mesh sampling algorithm offers a speed-up between 3 and 26 compared to rejection sampling (for the tested density textures and models) and it is up to 200× faster than the Poisson-disk sampling algorithm presented in [44] running on a GPU. The usage of the uniform grid improves the mesh sampling performance up to 7 times.

4.2 Procedural hair generation results

In this section I will analyze the speed of my hair generator (Section 4.2.1) and I will also show some visual results of my hair generator (Section 4.2.2).

4.2.1 Hair generation performance

The time needed for the hair generation compared to overall rendering will be mentioned in the following section. Here I will discuss how much time is taken by individual hair generation steps. To do so, I am using Visual Studio 2010 Profiler

on a fully optimized code with debug symbols. I run the profiler on a single thread of a 3.07 GHz Intel Core i7-950 PC with 6 GB RAM running Windows 7 64bit. The test scene consist of one Bunny model (see Figure 4.1) with a high-frequency Perlin noise density texture of 1024×1024 resolution. In all tests, 1000 hair guides are used and each hair is interpolated from the 3 closest hair guides.

Figure 4.3 shows a breakdown of the hair generation to individual steps for a scene with $1,000,000$ single hairs (without strands) with 5 annd 15 segments. It is easily observable, that one of the most expensive steps are *Frizz* and *Kink*. This is caused by the fact that they use the Perlin noise function, which is quite slow. Since *Kink* calls the Perlin noise function for every hair vertex (see Section 2.2.5), it becomes a severe bottleneck for the 15 segments hair. The interpolation of hair from the closest hair guides is also time consuming; both the interpolation itself and the search of the closest hair guides are responsible for that. Finally, sending the generated hair to RenderMan takes also significant amount of time, especially for the 15 segments hair. Please notice that thanks to my fast sampling algorithm, the generation of samples for the hair root calculation (the calculation of the hair root position from a sample is represented by a different step: Hair root) does not delay the hair generation.

	5 segments		15 segments	
Step	CPU Time	Internal body	CPU Time	Internal body
Generate sample	2.90%	97.87%	1.64%	95.45%
Hair root	8.07%	56.49%	5.73%	64.29%
Interpolate	26.62%	12.50%	17.86%	21.88%
Frizz	10.47%	16.47%	7.14%	28.65%
Kink	18.85%	10.78%	28.01%	8.63%
Other properties	10.41%	13.61%	5.28%	13.38%
Curve frame	2.71%	6.82%	4.50%	9.92%
Finalizing hair	5.98%	38.14%	11.64%	36.42%
Random numbers	0.43%	100.00%	0.30%	100.00%
Output hair	8.26%	0.00%	13.73%	0.00%
Rest	5.30%	N/A	4.17%	N/A

Table 4.3: A breakdown of the hair generation to individual steps for a scene with $1,000,000$ single hairs (without strands) with 5 (left) and 15 (right) segments. Internal body value indicates how much work was done by a hair generation step excluding the work done by external functions that were called by it (e.g. math functions, Perlin noise, RenderMan functions).

Similar to the previous table, Figure 4.4 shows a breakdown of the hair generation to individual steps, however this time I generate $200,000$ hair strands each consisting of 5 fibers with 5 and 15 segments. Remember that each fiber of a single hair strand is computed from one generated hair, so the hair root calculation, *Frizz*, *Kink*, the interpolation and other properties computation are executed only $200,000$ times, therefore they are not so expensive as they were in the previous test. On the other hand, for every fiber I need to execute the curve frame calculation and the finalizing hair step, which use time consuming mathematical operations (e.g. square root calculation), so these steps become more significant than in the previous test. Finally, generation of a hair strand

fiber from a single hair is displayed as the hair strand step. As in the previous test, sending the hair to RenderMan is expensive, especially for the 15 segments hair.

	5 segments		15 segments	
Step	CPU Time	Internal body	CPU Time	Internal body
Generate sample	3.11%	95.4%	0.73%	100%
Hair root	3.70%	68.00%	2.70%	59.46%
Interpolate	14.67%	13.13%	7.14%	34.69%
Frizz	5.04%	8.82%	3.21%	34.09%
Kink	9.93%	5.97%	12.17%	11.98%
Other properties	3.70%	8.00%	2.70%	10.81%
Curve frame	10.07%	8.82%	12.03%	13.33%
Finalizing hair	16.59%	33.04%	23.40%	34.58%
Hair strand	10.07%	8.82%	12.03%	13.33%
Random numbers	0.30%	100.00%	0.07%	100.0%
Output hair	17.63%	0.00%	25.00%	0.00%
Rest	5.19%	N/A	3.72%	N/A

Table 4.4: A breakdown of the hair generation to individual steps for a scene with $200,000$ strands each consisting of 5 fibers with 5 (left) and 15 (right) segments. Internal body value indicates how much work was done by a hair generation step excluding the work done by external functions that were called by it (e.g. math functions, Perlin noise, RenderMan functions).

Generator	Files output	Rendering	Hair interp.	Total	HD space
Mine	3.7s	38.69s	3.28s	45.67s	5.27MB
Shave and...	73.57s	79.12s	?	152.69s	495MB

Table 4.5: Results of rendering a model consisting of 19,800 triangles with million hairs. Faster rendering of the hair generated by my generator is caused by the fact, that no huge scene files with hair geometry need to be read and parsed by the renderer. Instead, the hair is generated on the fly during the render time from the hair guides.

Finally, I also compare the speed of my hair generator to *Shave and a Haircut*. I do it by rendering million hairs on a simple sphere consisting of $19,800$ triangles. Since *Shave and a Haircut* creates a hair guide for each mesh vertex, using meshes with higher polygon count causes performance issues for *Shave and a Haircut*. For my hair generator I use the same number of the hair guides as *Shave and a Haircut* generates and both mine and *Shave and a Haircut*'s hair guides have 5 segments. I use uniform density texture, since I want to compare hair generation speed, not sampling speed. Finally, I use 5 different textures with resolution 1024×1024 to set 5 hair properties and therefore increasing size of the temporary files used by my hair generator (*Shave and a Haircut*'s temporary files contains hair geometry, so they are not influenced by this). In Table 4.5 you can see the time and the hard drive space consumed during image rendering by *Shave and a Haircut* and by my hair generator. The Files output time is the time spend on the creation

of all files needed for rendering. Since *Shave and a Haircut* generates hair during Files output, I cannot measure the time needed for hair generation. In this test I have used the same machine and renderer as those described in the following section.

4.2.2 Visual results

Here I show several models with the hair (or grass) generated by my hair generator. Images were rendered using a free license of 3Delight RenderMan, which is only able to use 2 threads during rendering. In the description of each image I also mention the time spent on the hair generation and the total rendering time. To make the hair look more realistic, I use the specialized Kajiya-Kay shader (see [21]) to calculate the generated hair final color. All images were rendered on two threads of a 3.07 GHz Intel Core i7-950 PC with 6 GB RAM running Windows 7 64bit. Since I have only 6 GB RAM and also limited computational power, numbers of generated hairs (for fur) are quite low.

For each image I will mention hair generation time, a scene file output time (including hair bounding box calculation) and finally total rendering time, which includes the two aforementioned times. Figure 4.6 shows hair on a human head rendered from two positions and Figure 4.7 shows different hair colors with different hair noisiness. Figures 4.8 and 4.9 show fur generated by my hair generator, notice how hair density changes over the creatures surface. Finally, Figure 4.10 shows that my generator is also able to generate grass.

(a) Front view (b) Side view

Figure 4.6: Blond hair consisting of 100, 000 hair strands (each consisting of 7 fibers, each fiber has 10 segments). The scene files were created in 1.54 seconds and the hair generation took 1.23 seconds. Rendering of the front view took 287.09 seconds and rendering of the side view 313.97 seconds.

(a) Blond hair

(b) Red hair

(c) Brown hair

(d) Black hair

Figure 4.7: Four different hair styles and colors, each of them has $100,000$ hairs (each hair has 10 segments). The time spent on the creation of scene files was between 0.8–0.9, on the hair generation 0.5–0.6 and on rendering 161–409 seconds. Rendering time was mainly influenced by hair fibers length.

Figure 4.8: An imaginary wolf with $200,000$ hair strands (each consisted of 6 fibers and each fiber has 5 segments) was rendered in 1126 seconds, a scene file was created in 1.15 seconds and the hair generation took 1.6 seconds. Higher rendering time is caused by the usage of ambient occlusion.

Figure 4.9: An alien creature with 1.6 million hairs (each hair consisted of 5 segments) was rendered in 281.87 seconds, the scene file was created in 1.05 seconds and the hair generation took 4.5 seconds.

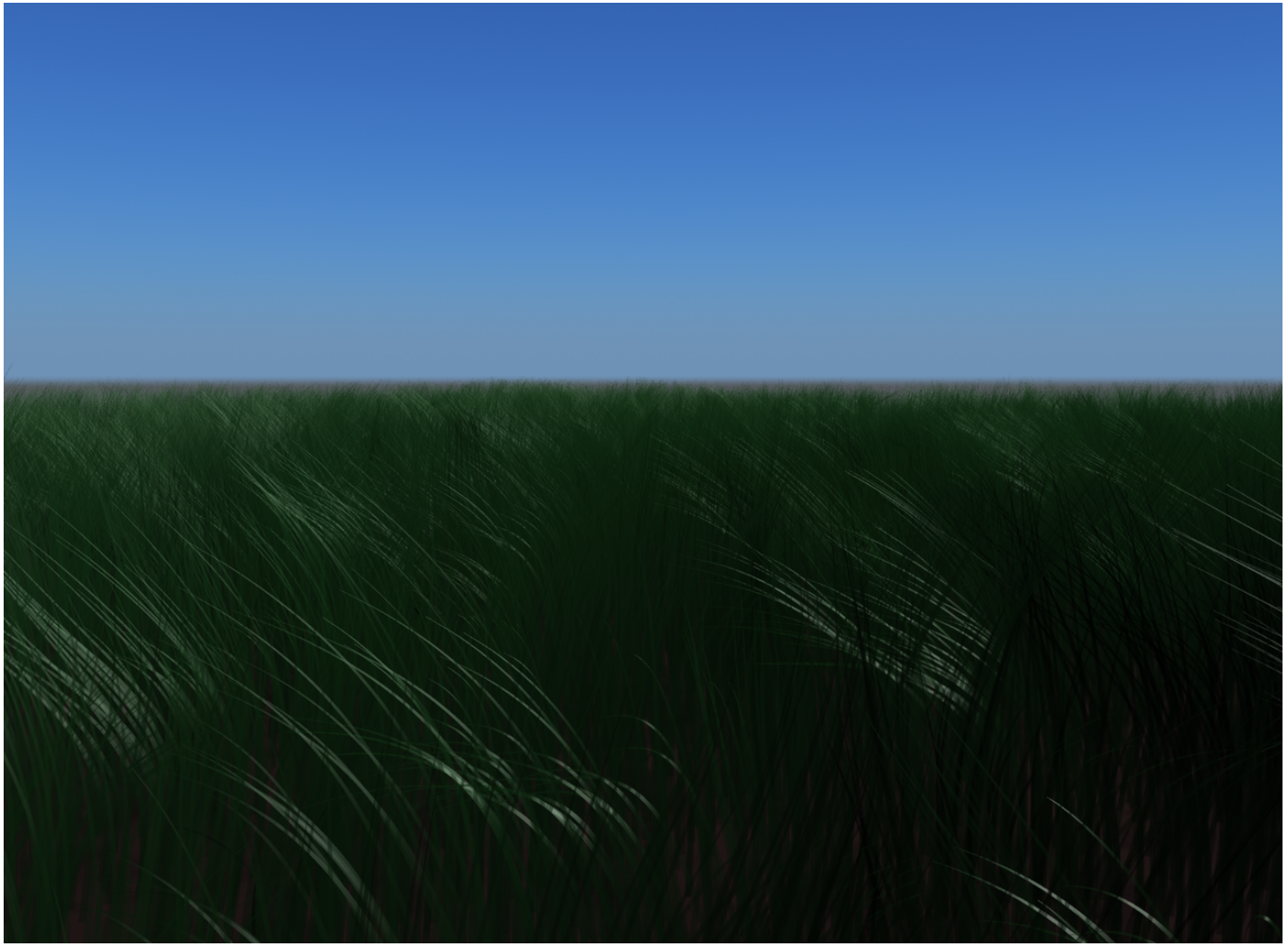


Figure 4.10: A grass field with $100,000$ leaves (each leaf consisted of 5 segments) was rendered in 27.47 seconds, the scene file was created in 0.41 seconds and the hair generation took 0.45 seconds.

Conclusion

The goal of this thesis was to create a procedural hair generator for the Stubble project that would be able to generate hair in similar manner as *Shave and a Haircut*. Furthermore, this procedural generator should be able to work both interactively during modeling in Stubble project and also it should generate hair during rendering by specialized rendering software without the need to export all hair geometry to a renderer. All of these goals were successfully achieved.

4.3 Summary

One of the first tasks I had to solve, was to determine how to distribute hair root positions on a given model surface according to a density stored as 2-dimensional texture mapped on a model surface. For that purpose, I have developed a fast sampling algorithm of triangular meshes that generates random samples according to a density texture. My sampling algorithm is up to 26 times faster than the fastest alternative — rejection sampling.

The hair generated by my generator is influenced by the hair guides and the hair properties. The hair guides are few hairs directly modeled by a 3d artist; each generated hair is interpolated from a selected number of the closest hair guides. I have discussed different hair interpolation algorithms along with an implementation of the chosen one. The hair properties were mainly taken from *Shave and a Haircut* and my hair generator had to be made in such a way that changing the hair properties would have the same effect as in *Shave and a Haircut*. This was not an easy task since *Shave and a Haircut* is a commercial product and I had no access to its source code. The hair properties fall into several categories, some cause change of hair geometry by applying noise, others influence hair color and width or tell the hair generator to create hair strands instead of single hairs. All of these properties were mentioned in the description of my hair generator.

One of the crucial differences between my hair generator and the hair generator from *Shave and a Haircut* is that it is able to generate the final hair on the fly without the need to store hair geometry in files with the rest of the rendered scene. This makes the rendering process very efficient. To implement the hair generation during rendering I export data such as the hair guides or the hair properties to files (with insignificant size compared to a scene file with complete hair geometry), that I later use to generate hair by my library which is executed by the renderer.

My hair generator is easily expandable to support numerous renderers. This is mainly achieved by its universal implementation. Thanks to this, I was also able to easily make the interactive hair generation and hair rendering by OpenGL. My hair generator currently supports two rendering softwares: 3Delight RenderMan and mentalray. Support for mentalray was written by another team member of the Stubble project using my hair generation library.

One of the original tasks of my thesis was handling of collisions of the generated hair with a model surface. The current implementation of my hair generator relies on the fact that collisions of hair guides with the surface are handled within the Stubble project and so in many cases it is unnecessary to handle collisions of

the generated hairs with the surface. Of course this is not true for many scenes, especially when very long hairs are generated. The main reason why I avoid generated hairs collisions handling is because it would be very time consuming.

4.4 Future work

There are several tasks that could be addressed in future. First of all, I would like to improve my sampling algorithm, so it is able to generate blue noise distributions of samples in a very fast way. Such a fast sampling algorithm could be used by many applications as suggested by numerous papers mentioned in Chapter 1.

As I have mentioned above, collisions of the generated hair with a model surface are not currently implemented for performance reasons; however I believe that if an approximation of the model surface was used along with some simplified hair–surface collisions reactions, it could be possible to handle the collisions for small counts of the generated hair (by small counts I mean around 100,000 which is sufficient for a human head).

The hair generated by my procedural hair generator can be rendered by 3Delight RenderMan, mentalray and OpenGL. In the future, I would like to extend this list to Pixar's RenderMan PRMan and Arnold renderer. Both of these renderers are often used by 3d artists.

Finally, I would like to speed up hair generation by implementing some features mentioned in [35], such as reducing generated hair numbers along with hair width increase when the hair is viewed from a huge distance or motion blur is applied on it.

Bibliography

[1] 3DELIGHT. *Rendering Custom Nodes* [online]. 2011. Available from: `http://www.3delight.com/en/uploads/docs/3dfm/3dfm_39.html`.

[2] AKENINE-MÖLLER, T. – HAINES, E. – HOFFMAN, N. *Real-Time Rendering 3rd Edition*. Natick, MA, USA : A. K. Peters, Ltd., 2008. ISBN 987-1-56881-424-7.

[3] AMIDROR, I. Scattered Data Interpolation Methods for Electronic Imaging Systems: A Survey, 2002.

[4] AURENHAMMER, F. Voronoi diagrams — a survey of a fundamental geometric data structure. *ACM Comput. Surv.* September 1991, 23, 3, s. 345–405. ISSN 0360-0300. doi: 10.1145/116873.116880. Available from: `http://doi.acm.org/10.1145/116873.116880`.

[5] BERTAILS, F. et al. Predicting Natural Hair Shapes by Solving the Statics of Flexible Rods. August 2005. Available from: `http://www-evasion.imag.fr/Publications/2005/BAQLLC05/`.

[6] BOWERS, J. et al. Parallel Poisson disk sampling with spectrum analysis on surfaces. *ACM Trans. Graph.* 2010, 29, 6, s. 166:1–166:10. ISSN 0730-0301.

[7] CHESLACK-POSTAVA, E. et al. Fast, realistic lighting and material design using nonlinear cut approximation. *ACM Trans. Graph.* December 2008, 27, 5, s. 128:1–128:10. ISSN 0730-0301.

[8] CHOE, B. – KO, H. A statistical wisp model and pseudophysical approaches for interactive hairstyle generation. *IEEE Transactions on Visualization and Computer Graphics*. 2005, 11, 2, s. 160–170.

[9] CHRISTENSEN, P. Point-Based Approximate Color Bleeding. Technical Report #08-01, Pixar Studios, July 2008.

[10] CORSINI, M. – CIGNONI, P. – SCOPIGNO, R. Efficient and Flexible Sampling with Blue Noise Properties of Triangular Meshes. *IEEE Transactions on Visualization and Computer Graphics*. 2012, 99, RapidPosts. ISSN 1077-2626.

[11] EBEIDA, M. S. et al. A Simple Algorithm for Maximal Poisson-Disk Sampling in High Dimensions. *Computer Graphics Forum*. May 2012, 31, 2.

[12] GOLDMAN, R. *Realistic Image Synthesis Using Photon Mapping*. Morgan Kaufmann, 2002. ISBN 978-1558603547.

[13] GROSS, M. – PFISTER, H. *Point-based graphics*. Morgan Kaufmann, 2007. ISBN 9780123706041.

[14] HADAP, S. – MAGNENAT-THALMANN, N. Interactive Hair Styler based on Fluid Flow. In *Eurographics Workshop on Computer Animation and Simulation 2000*, s. 87–99. Springer, August 2000.

[15] HALTON, J. H. Algorithm 247: Radical-inverse quasi-random point sequence. *Commun. ACM*. December 1964, 7, 12, s. 701–702. ISSN 0001-0782. doi: 10.1145/355588.365104. Available from: http://doi.acm.org/10.1145/355588.365104.

[16] HEMSLEY, R. Interpolation on a Magnetic Field. Technical report, 2009. Also available as http://interpolate3d.googlecode.com/files/Report.pdf.

[17] HERNANDEZ, B. – RUDOMIN, I. Hair Paint. In *Proceedings of the Computer Graphics International*, CGI '04, s. 578–581, Washington, DC, USA, 2004. IEEE Computer Society. doi: 10.1109/CGI.2004.56. Available from: http://dx.doi.org/10.1109/CGI.2004.56. ISBN 0-7695-2171-1.

[18] JAMES, F. A review of pseudorandom number generators. *Computer Physics Communications*. October 1990, 60, s. 329–344.

[19] JENSEN, H. W. *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Ltd., 2001. ISBN 978-1568811475.

[20] JENSEN, H. W. – BUHLER, J. A rapid hierarchical rendering technique for translucent materials. *ACM Trans. Graph.* July 2002, 21, 3, s. 576–581. ISSN 0730-0301.

[21] 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. Available from: http://doi.acm.org/10.1145/74334.74361.

[22] KIM, T.-Y. – NEUMANN, U. Interactive multiresolution hair modeling and editing. *ACM Trans. Graph.* July 2002, 21, 3, s. 620–629. ISSN 0730-0301. doi: 10.1145/566654.566627. Available from: http://doi.acm.org/10.1145/566654.566627.

[23] LAGAE, A. – DUTRÉ, P. A procedural object distribution function. *ACM Trans. Graph.* October 2005, 24, 4, s. 1442–1461. ISSN 0730-0301.

[24] LAGAE, A. – DUTRÉ, P. A Comparison of Methods for Generating Poisson Disk Distributions. *Computer Graphics Forum*. 2008, 27, 1, s. 114–129. ISSN 1467-8659.

[25] LENGYEL, J. et al. Real-time fur over arbitrary surfaces. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, I3D '01, s. 227–232, New York, NY, USA, 2001. ACM. doi: 10.1145/364338.364407. Available from: http://doi.acm.org/10.1145/364338.364407. ISBN 1-58113-292-1.

[26] MEIER, B. J. Painterly rendering for animation. In *Proceedings of SIGGRAPH '96*, s. 477–484. ACM, 1996. ISBN 0-89791-746-4.

[27] NEHAB, D. – SHILANE, P. Stratified Point Sampling of 3D Models. In *Eurographics Symposium on Point-Based Graphics*, s. 49–56, June 2004.

[28] OSADA, R. et al. Shape distributions. *ACM Trans. Graph.* October 2002, 21, 4, s. 807–832. ISSN 0730-0301.

[29] PELIKÁN, J. *JaGrLib - library for computer graphics* [online]. 2011. Available from: `http://cgg.mff.cuni.cz/JaGrLib/`.

[30] PERLIN, K. Improving noise. *ACM Transactions on Graphics*. July 2002, 21, 3, s. 681–682. ISSN 0730-0301.

[31] PHARR, M. – HUMPHREYS, G. *Physically Based Rendering, Second Edition: From Theory To Implementation*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2nd edition, 2010. ISBN 0123750792, 9780123750792.

[32] PIXAR. *The RenderMan Interface, Version 3.2.1* [online]. November 2005. Available from: `https://renderman.pixar.com/products/rispec/rispec_pdf/RISpec3_2.pdf`.

[33] QU, L. – MEYER, G. W. Perceptually driven interactive geometry remeshing. In *Proceedings of the 2006 symposium on interactive 3D graphics and games*, I3D '06, s. 199–206. ACM, 2006. ISBN 1-59593-295-X.

[34] RITSCHEL, T. et al. Micro-rendering for scalable, parallel final gathering. *ACM Trans. Graph.* December 2009, 28, 5, s. 132:1–132:8. ISSN 0730-0301.

[35] RYU, D. 500 Million and Counting: Hair Rendering on Ratatouille. Technical report, Pixar, May 2007. Also available as `http://graphics.pixar.com/library/500MillionHairs/paper.pdf`.

[36] SHEPARD, D. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM national conference*, ACM '68, s. 517–524, New York, NY, USA, 1968. ACM. doi: http://doi.acm.org/10.1145/800186.810616. Available from: `http://doi.acm.org/10.1145/800186.810616`.

[37] SHREINER, D. et al. *OpenGL Průvodce programátora*. Computer Press, 2006. ISBN 80-251-1275-6.

[38] TKALČIČ, M. Colour spaces - perceptual, historical and applicational background. *EUROCON 2003. Computer as a Tool. The IEEE Region 8*. 2003.

[39] TURK, G. Re-tiling polygonal surfaces. In *Proceedings of SIGGRAPH '92*, s. 55–64. ACM, 1992. ISBN 0-89791-479-1.

[40] WANG, T. – YANG, X. D. Geometric modeling. Norwell, MA, USA: Kluwer Academic Publishers, 2004. Hair design based on the hierarchical cluster hair model, s. 330–359. Available from: `http://dl.acm.org/citation.cfm?id=985821.985839`. ISBN 1-4020-1817-7.

[41] WANG, W. et al. Computation of rotation minimizing frames. *ACM Trans. Graph.* March 2008, 27, s. 2:1–2:18. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1330511.1330513. Available from: `http://doi.acm.org/10.1145/1330511.1330513`.

[42] WARD, K. et al. A Survey on Hair Modeling: Styling, Simulation, and Rendering. *IEEE Transactions on Visualization and Computer Graphics*. March 2007, 13, 2, s. 213–234. ISSN 1077-2626. doi: 10.1109/TVCG.2007.30. Available from: `http://dx.doi.org/10.1109/TVCG.2007.30`.

[43] WEI, Y. et al. Modeling hair from multiple views. *ACM Trans. Graph.* July 2005, 24, 3, s. 816–820. ISSN 0730-0301. doi: 10.1145/1073204.1073267. Available from: `http://doi.acm.org/10.1145/1073204.1073267`.

[44] XIANG, Y. et al. Parallel and accurate Poisson disk sampling on arbitrary surfaces. In *SIGGRAPH Asia 2011 Sketches*, s. 18:1–18:2, 2011. ISBN 978-1-4503-1138-0.

[45] YANG, G. et al. Interactive fur modeling based on hierarchical texture layers. In *Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, VRCIA '06, s. 343–346, New York, NY, USA, 2006. ACM. doi: 10.1145/1128923.1128983. Available from: `http://doi.acm.org/10.1145/1128923.1128983`. ISBN 1-59593-324-7.

[46] YANG, X. D. et al. The cluster hair model. *Graph. Models*. March 2000, 62, 2, s. 85–103. ISSN 1524-0703. doi: 10.1006/gmod.1999.0518. Available from: `http://dx.doi.org/10.1006/gmod.1999.0518`.

[47] YU, Y. Modeling Realistic Virtual Hairstyles. In *Proceedings of the 9th Pacific Conference on Computer Graphics and Applications*, PG '01, s. 295–, Washington, DC, USA, 2001. IEEE Computer Society. Available from: `http://dl.acm.org/citation.cfm?id=882473.883447`. ISBN 0-7695-1227-5.

[48] YUKSEL, C. – TARIQ, S. Advanced techniques in real-time hair rendering and simulation. In *ACM SIGGRAPH 2010 Courses*, SIGGRAPH '10, s. 1:1–1:168, New York, NY, USA, 2010. ACM. doi: 10.1145/1837101.1837102. Available from: `http://doi.acm.org/10.1145/1837101.1837102`. ISBN 978-1-4503-0395-8.

[49] YUKSEL, C. – SCHAEFER, S. – KEYSER, J. On the parameterization of Catmull-Rom curves. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, SPM '09, s. 47–53, New York, NY, USA, 2009. ACM. doi: 10.1145/1629255.1629262. Available from: `http://doi.acm.org/10.1145/1629255.1629262`. ISBN 978-1-60558-711-0.

[50] YUKSEL, C. – KEYSER, J. – HOUSE, D. H. Mesh colors. *ACM Trans. Graph.* April 2010, 29, 2, s. 15:1–15:11. ISSN 0730-0301.

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 hair generator. 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.
- **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. Some of the information inside the developer manual concerning the hair generation may be outdated.
- **thesis-source** Here you can find the source files (e.g. LaTeX documents) of my thesis.
- Furthermore, the root of the CD contains my thesis as **thesis.pdf** and the Stubble project instalator **stubble-setup.exe**. For instructions about Stubble installation and using Stubble (including the hair generation) see the Stubble user manual.