Fast Random Sampling of Triangular Meshes for Hair Modeling

Martin Šik*
Supervised by: Jaroslav Křivánek†

Faculty of Mathematics and Physics
Charles University in Prague

Abstract

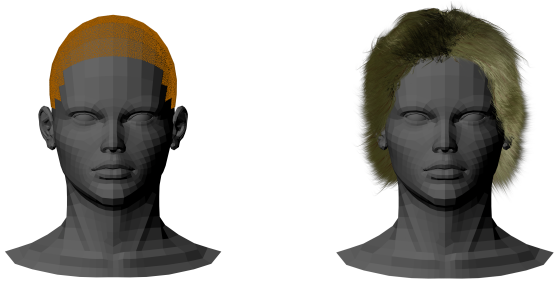
Efficient modeling of hair for realistic computer animation is a difficult problem because of the sheer number of individual hairs on a human head or an animal body. Random placement of hair roots on an arbitrary triangle mesh is an important sub-task of this problem. The main contribution of this paper is a simple, fast, and memory-efficient algorithm for randomly distributing points on a triangular mesh with a density specified by a two-dimensional texture. Our algorithm is many times faster than existing alternatives, such as rejection sampling. Furthermore, we describe a software architecture for procedural generation of hair in render-time. This module can generate millions of hairs during rendering from only a few guide hairs directly modeled by a 3d artist, which makes the rendering process very efficient.

Keywords: Mesh sampling, sample density, hair modeling, procedural generation

1 Introduction

Recent 3D animated films contain creatures with lots of fur and nowadays the trend in the film industry is to use CGI for such creatures even in non-animated films. The key question is how to effectively create hair (or fur) styling. Modifying each hair individually by a 3d artist is not practical because it would take too much time and therefore cost large amount of money. It is much easier for a 3d artist to create and model only a subset of hair fibers from which final hair will be automatically generated. The current approach is to export the generated hair fibers of the final hair into a large file that is then sent to the renderer that produces the high quality frames. However, because of the sheer number of individual hair fibers in the final hair style, such a file can take up gigabytes of disk space and therefore working with it becomes inefficient.

In this paper, we present a library that is able to generate final hair from a modeled subset of hair during render time, therefore skipping the export of hair to a scene file. Furthermore our library is able to display thousands of hair interactively. Hair generated by our library has also a number of parameters that improve realism of hair, such as creation of hair strands or influencing hair shape with a noise.



(a) Point samples (b) Generated hair

Figure 1: The left picture shows 100,000 point samples that has been used in the right picture as hair roots positions.

One of our goals is to have very fast generation of hair. The main bottleneck in hair generation is random placement of hair positions on a model. To remove this problem, we have developed a new algorithm for fast sampling of triangular meshes. Since real hair density may vary a lot over a human or an animal skin, the main feature of our sampling algorithm is that the density of generated samples can be defined by a two-dimensional texture mapped on the model from which the hair grows.

Figure 1 shows an example of hair roots placement. Image 1a demonstrates 100,000 points generated by our algorithm while the image 1b shows hair procedurally generated by our library growing from these points.

In Section 2, we discuss the present state of the art in mesh sampling algorithms and hair modeling. In Section 3, we describe our new algorithm for sampling on triangular meshes and present its results. In Section 4, we present a software architecture for procedural generation of hair. Finally, we conclude in Section 4 and present directions for future work.

2 Related Work

2.1 Random Sample Generation

Random sample generation is a problem that has been a point of interest in the field of computer graphics for a long time, since it can benefit a variety of graphics applica-

*martin_sik@centrum.cz
†jaroslav@cgg.mff.cuni.cz

tions in texturing, rendering, remeshing, and point-based graphics [6, 7, 10]. Our goal is fast generation of samples on a triangular mesh. This specific issue is addressed by [2], however their priority is not efficiency, but rather good distribution of generated samples. Apart from this article there are other works (for example [4]) that address fast generation of samples with good distribution, but they generate samples in a plane and it is unclear how to apply them for generation of samples on a triangular mesh. Standard methods like rejection sampling [6] can also be used to generate samples on a triangular mesh, however they are too slow for our purpose.

2.2 Modeling and Rendering Hair

Modeling and rendering of hair is an issue addressed by numerous works, but they do not discuss the procedural generation of hair in render-time. Among the tools used for hair modeling in standard modeling programs (such as *Autodesk Maya*), *Shave and a Haircut*[1] allows the user to model a subset of hair from which it then procedurally generates hair. However *Shave and a Haircut* is unable to execute the generation of hair in render-time without the need of saving hair geometry to a scene file. Since *Shave and a Haircut* is a commercial product, it is unknown how it generates hair or places hair roots on a model. Generation of hair in render-time is mentioned in [8], however their work is focused on speeding up rendering of hair in a specific renderer and they omit how exactly is the generation of hair done.

3 Random Points Generation

In this section we describe our algorithm for randomly distributing points on a triangular mesh with a density specified by a two-dimensional texture. Note that the problem would be trivial if the desired point density was uniform over the surface: in such a case, we would simply pick a triangle proportionally to its area (using the inversion of a discrete cumulative distribution function) and place the point uniformly in the selected triangle.

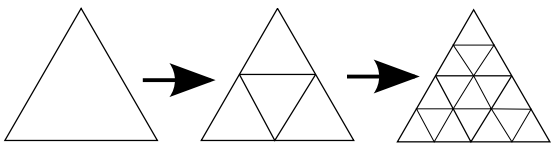


Figure 2: Recursive subdivision of a triangle.

However, since the point distribution probability density must take into account both the density texture mapped on the mesh surface and also each mesh triangle's area size, creation of the cumulative distribution function is not trivial. We overcome this problem by recursively subdividing each mesh triangle to sub-triangles (see Figure 2) until every sub-triangle's surface has uniform density. We then calculate the probability that new sample will be created on a given sub-triangle as the density texture value integrated over the sub-triangle's surface. We use these probabilities to create the discrete cumulative distribution function. Finally for every desired random point we first use this cumulative distribution function to randomly select any sub-triangle from the mesh surface based on its probability and then we uniformly sample the selected sub-triangle to determine a generated point position.

[1] http://www.joealter.com/

3.1 Defining the sub-triangle probability distribution

In order to define the sub-triangle probability distribution we iterate through all triangles of the given mesh surface. As previously mentioned, we recursively subdivide each triangle, however to make the subdivision a lot faster we do not actually check if every sub-triangle's surface has uniform density, instead we will stop the subdivision if no more than one texel of the density texture is mapped on each sub-triangle. Thanks to this we can calculate a subdivision depth for each triangle directly from number of the density texture texels mapped on it. Also we do not need to integrate the density texture over a sub-triangle to determine its probability, we only evaluate the density texture at the sub-triangle's barycenter.

As we will discuss later, both the speed and memory cost of our algorithm depends on the total number of sub-triangles. To decrease the number of sub-triangles we can check the probabilities of four sub-triangle siblings and if they have the same probability, we can use their parent instead of them (see Figure 3). Again we can check the parent's siblings and continue until the probabilities differ or we have reached a non-subdivided triangle. After these steps are applied each sub-triangle is only divided if its surface has not uniform density as it was mentioned in the brief algorithm description.

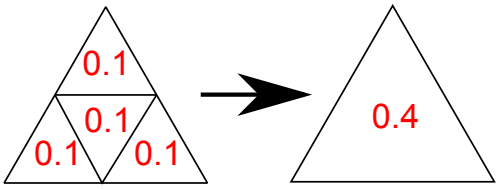


Figure 3: If the four sub-triangle siblings have the same probability, their parent may be used instead of them. The numbers represent the probabilities.

As we have said before, the final step of our algorithm is sampling the selected sub-triangle. In order to calculate the sample point position on the triangle mesh we need to store for each sub-triangle a reference to the parent mesh triangle and where it is located inside the triangle (i.e. the barycentric coordinates of the sub-triangle in the parent triangle). This has to be done during the computation of the

sub-triangle probability distribution. Since the subdivision scheme is same for every triangle, we only need to give unique index to each possible sub-triangle position inside a triangle (see Figure 4) and store barycentric coordinates of these sub-triangle positions in a separate data structure, where they can be easily accessed by the position index. This significantly reduces memory consumption, since we only need to store two indices per sub-triangle (a parent triangle index and a sub-triangle position index) and its probability and therefore each sub-triangle takes up only 12 bytes in memory (considering the fact that all 3 values takes up 4 bytes). The size of the pre-computed data structure which stores sub-triangle positions is negligible.

During a sub-triangle subdivision we will need to calculate the index i of a sub-triangle position from the index i_{parent} of its parent sub-triangle position: $i = 4(i_{parent} + 1) + j$, where $j \in [0,3]$ is the j-th sub-triangle of its subdivided parent sub-triangle.

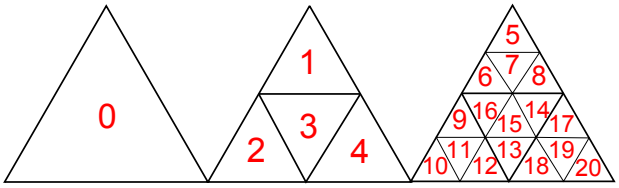


Figure 4: Indices of sub-triangle positions inside a triangle.

The complete algorithm for the computation of the sub-triangle probability distribution is described in Figure 5.

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 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 5: The computation of the sub-triangle probability distribution.

3.2 Generating a Point Sample

As mentioned before, to generate a point sample we first need to randomly select a sub-triangle based on its previously computed probability. To implement this random selection we need to convert sub-triangles probabilities to a cumulative distribution function.

The cumulative distribution function is defined as $F = P(X \leq x)$, where $P(X \leq x)$ is the probability of $X \leq x$ and X is a random variable from the distribution D. Since we use only discrete version of the cumulative distribution function, we can easily compute it:

$$F(j) = \sum_{i=0}^{j} P(X = i) = F(j-1) + P(i),$$

where $F(0)$ is defined as $P(X = 0)$ and $P(X = i)$ describes the probability of a discrete random variable X being equal to i. The discrete function F is then stored as an array.

Following [6], we find a sub-triangle as $\arg\min_i (F(i) > \xi)$, where ξ is a random number from $U(0, \max F(x))$. Because values of $F(i)$ are increasing with greater i, we can use the bisection algorithm to find $\arg\min_i (F(i) > \xi)$. Finally, we generate a random sample in the selected sub-triangle with uniform probability density, and map the sample to the barycentric coordinates of the parent mesh triangle. The complete algorithm for generating a point sample is described in Figure 6.

3.3 Further Improvements

As described in Figure 6, for each sample we select a sub-triangle using the bisection algorithm. The bisection algorithm runs in logarithmic time proportional to the size of the cumulative distribution function domain. Since the domain of F is usually very large, even logarithmic time for generating each sample may be quite a lot.

We improve speed of the sub-triangle selection by creating 1-dimensional uniform grid G over the F codomain (see Figure 7). For each cell C of the grid we store two indices C_{begin} and C_{end} (elements of the F domain):

$$\begin{aligned} C_{begin} &= \arg\min_i \{F(i) \in C\} \\ C_{end} &= \arg\max_i \{F(i) \in C\} \end{aligned}$$

When generating a sample, we first generate a random number $\xi_0 \in [0, \max F]$ as before, but then we determine a cell C of the uniform grid for which $\xi_0 \in C$ in constant time, after that we select a sub-triangle using the bisection algorithm only in limited domain of F: $[C_{begin}, C_{end}]$.

For every requested point sample:

1. Select a sub-triangle:
   (a) Generate a random number $\xi_0 \in [0, \max F]$.
   (b) Select a sub-triangle as $D = \arg\min_i F(i) > \xi_0$ using the bisection algorithm.
2. Generate a sample in the selected sub-triangle
   (a) Generate two uniform numbers $\xi_1, \xi_2 \in$ [0,1].
   (b) Calculate the barycentric coordinate $u_D = 1 - \sqrt{\xi_1}$
   (c) Calculate the barycentric coordinate $v_D = \xi_2 \cdot \sqrt{\xi_1}$
3. Map the sample to the barycentric coordinates (u,v) in the parent mesh triangle:
   (a) From u_D, v_D calculate the barycentric coordinates of a sample in the triangle T containing the selected sub-triangle D:
   $u = u_D \cdot u_1 + v_D \cdot u_2 + (1 - u_D - v_D) \cdot u_3$
   $v = u_D \cdot v_1 + v_D \cdot v_2 + (1 - u_D - v_D) \cdot v_3$
   where u_i, v_i are the barycentric coordinates of D vertices inside T. We obtain u_i, v_i by a lookup in the precomputed sub-triangle position data structure (Section 3.1) using the sub-triangle index.

Figure 6: Generating a point sample.

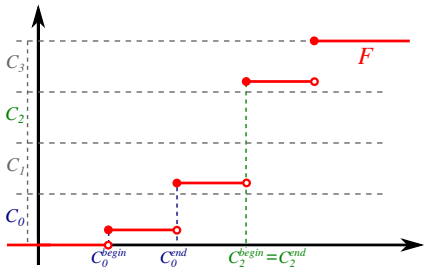


Figure 7: The uniform grid G built over the cumulative distribution function F codomain. The C_{begin} and C_{end} represent indices stored for every grid cell C.

The number of cells in the uniform grid G will influence samples generation performance. If grid has so many cells that in each cell lies only single member of the F domain then the selection of a sub-triangle will run in constant time at the cost of an increased memory consumption.

3.4 Results

We test our sampling algorithm on a single core of a 3.07 GHz PC with 6 GB RAM running Windows 7 64bit. We typically use the same number of cells of the uniform grid as is the size of the F domain (i.e. the number of generated sub-triangles) and a density texture with resolution 1024×1024 except if noted otherwise.

3.4.1 Comparison to Rejection Sampling

We compare our algorithm against rejection sampling since we were not able to find any other alternatives in the existing literature. The rejection sampling first randomly selects a triangle based on its area (for that purpose we use in our implementation a cumulative distribution function), then the triangle is uniformly sampled and the sample is accepted if a random value is not lesser than the value of the density texture at the sample point, otherwise it is rejected and rejection sampling generates a new sample.

Figure 8 plots the performance of our algorithm and the rejection sampling algorithm tested for three models with a uniform density texture. Since we have used a uniform density texture, the rejection sampling algorithm never rejects any generated sample. Faster sampling rate of our algorithm is therefore caused only by the usage of the uniform grid built over the cumulative distribution function codomain: we can see that this technique provides a speed-up between 3 and 6 for the tested cases. Usage of the uniform grid should remove the dependency on a model triangle count, however if triangles' areas differ greatly the cumulative distribution function is non-linear and therefore the uniform grid is not very efficient. All models are displayed in Figure 10 and their triangle counts can be found in table 2.

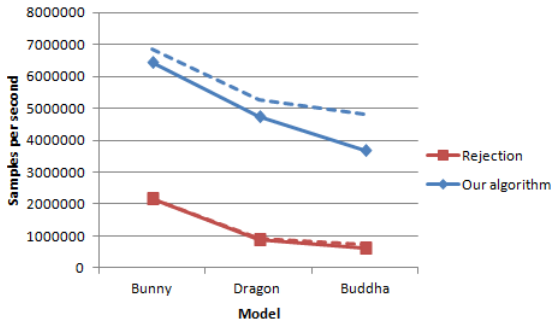


Figure 8: The sampling rate (samples per second) for our algorithm and for the rejection sampling algorithm. 3 different models with a uniform density texture were used in this test. Dashed lines represent the sampling rate without the preprocess time (i.e. triangle sub-division and the cumulative distribution function construction).

We have also tested the influence of the uniform grid resolution on the sampling performance. As shown in Table 1, total time spent on the preprocess part of our algorithm is only slightly influenced by the cell count, however memory consumption grows significantly with increasing number of the grid cells. The highest sampling rate is achieved when the number of cells is two times higher than the size of the cumulative distribution function domain.

Figure 9 also plots the performance of our algorithm and the rejection sampling algorithm, but this time we have tested them with 4 density textures with different average values. The average value of a density texture corresponds to the overall density of samples and also to the percentage

Grid size	#Samples	CDF constr.	Memory
0%	2365628	0.054s	1.33 MB
13%	5855405	0.055s	1.47 MB
25%	5951817	0.057s	1.60 MB
50%	5890408	0.058s	1.86 MB
100%	6387220	0.059s	2.39 MB
200%	6531778	0.061s	3.45 MB
400%	6447026	0.066s	5.57 MB

Table 1: The results of testing the preprocess part of our algorithm and sampling speed for the bunny model with different resolutions of the uniform grid. The grid size is reported as percentage of the cumulative distribution function size. We have used a uniform density texture for this test.

of samples accepted by the rejection sampling algorithm. The plots show that the performance of our algorithm remains roughly the same for all textures, but the performance of rejection sampling algorithm depends linearly on a decreasing texture average value.

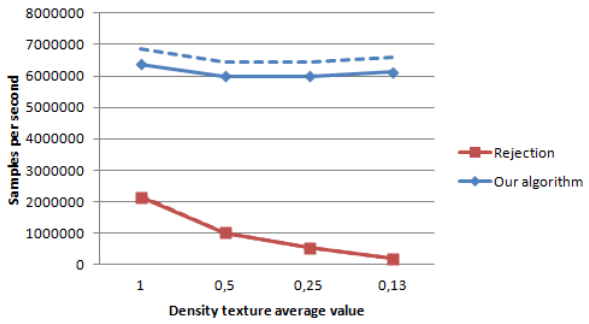


Figure 9: The sampling rate (samples per second) for our algorithm and for the rejection sampling algorithm. 4 density textures with different average values mapped on the bunny model were used in this test. Dashed lines represent the sampling rate without the preprocess time.

3.4.2 Performance

Table 2 shows the preprocess time for three different models with a uniform density texture. Both time and memory consumed by the preprocess is roughly linear in the model triangle count.

Model	Preprocess	Memory	#△
Bunny	0.070s	2.39 MB	69k
Dragon	0.127s	8.68 MB	202k
Buddha	0.25s	51.26 MB	1087k

Table 2: The result of testing the preprocess part of our algorithm for mesh surfaces with different triangle count. The results of sampling performance are shown in Figure 9. All models were used with a Perlin noise density texture.

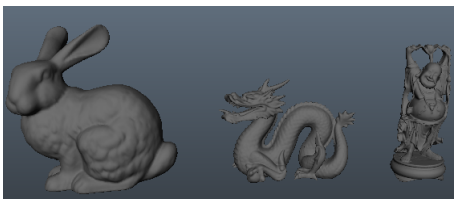


Figure 10: Models used for testing.

Finally we have tested the influence of the density texture resolution. Table 3 demonstrates that time and memory spent on the preprocess part of our algorithm depends linearly on a density texture texel count. Thanks to the uniform grid the sampling rate is only diminished by the longer preprocess time.

Resolution	#Samples	Preprocess.	Memory
512^2	6757501	0.019s	4.49 MB
1024^2	6306838	0.070s	5.33 MB
2048^2	4998126	0.253s	31.26 MB

Table 3: The results of testing our sampling algorithm for the bunny model with different resolutions of a Perlin noise density texture. The number of generated samples is per second.

3.4.3 Visual Results

Figure 11 shows a uniform distribution of points on the bunny model. The points distribution over the surface is not particularly good, but that is the price we have to pay for very fast sampling. Since the main purpose of this algorithm is the placement of hair roots positions, the quality of sampling is not so important. Figure 12 demonstrates a distribution of points controlled by a simple density texture.

Figure 11: The bunny model with 8,000 uniform points samples.

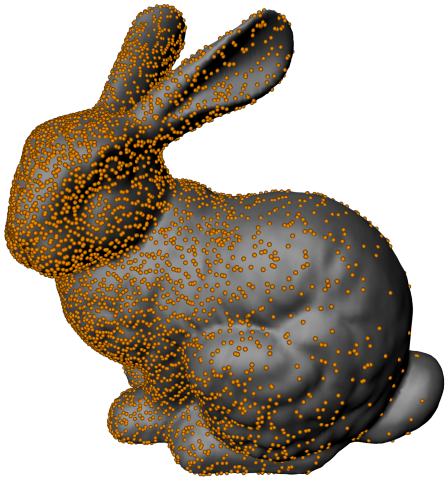


Figure 12: The bunny model with 8,000 points samples of increasing density from left to right.

4 Hair Modeling and Rendering

The procedural generator of hair we provide is a part of a complete hair modeling plug-in for Autodesk Maya called Stubble. We will first describe the pipeline of hair modeling used in Stubble in Section 4.1 and then we will closely discuss the software architecture of the hair procedural generator in Section 4.2.

4.1 Hair Modeling Pipeline

The main idea behind hair modeling with Stubble is to let a 3d artist model only small subset of hair, called hair guides, from which the rest of hair will be automatically generated by Stubble. Creating hair with Stubble can be divided into several steps:

1. **Preparing hair guides**: The first step is to create a few hair guides on a selected triangular mesh. The user may set the number of hair guides and their density over the mesh. Their roots positions are generated by our sampling algorithm described in Section 3.

2. **Modeling hair guides**: Each hair guide is represented by a polyline and user can model it using special tools which behave like comb or scissors.

3. **Applying dynamics**: After the basic modeling of guides is done, hair guides can be animated by applying hair dynamics.

4. **Hair properties**: When hair guides are properly modeled and animated, we procedurally generate the final hair fibers based on the guides. In the basic form, the generated hair simply interpolate their shape from nearby guides, but they may additionally be affected by random noise, color and other parameters. Stubble plug-in can interactively display thousands of generated hair during hair modeling in the Maya viewport, so the 3d artist may see how the final hair will look like. This is possible thanks to the efficiency of our point distribution algorithm.

5. **Rendering**: Final step is rendering hair with specialized rendering software. During rendering hundred thousands or even millions of hairs with selected properties are generated by Stubble from hair guides.

Figure 13 demonstrates hair modeling on a human head.

(a) Hair guides modeling (b) Interactive hair generation (c) Final render

Figure 13: Hair modeling pipeline

4.2 Procedural Generation of Hair

4.2.1 The Library Architecture

We have designed our procedural hair generator as a dynamic-link library that can be executed by different renderers. For each supported renderer, there is separate entry-point which implements renderer specific requirements for libraries that generate scene geometry.

To improve performance of hair generation we want to parallelize it. Since renderers are usually parallelized themselves, we take advantage of that and parallelize hair generation by calling our library separately from different threads of a renderer. Each of these calls is responsible for generating hair on one selected part of a triangular mesh and runs in a single thread. User of Stubble sets the number of parts to which the mesh is split and therefore the number of calls of our library. Is then up to the renderer to decide on how many threads these calls will be executed. It is important to mention that each frame of rendered animation is handled completely separately. For example if the mesh on which hair grows is split to 8 parts and we render 4 frames, our library will be called 32 times.

Most of the renderers require to know the bounding box of the geometry generated by any library before the library is even called. For example RenderMan interface compliant renderers use this information to optimize memory consumption, which is key feature when rendering millions of hairs. It is mentioned in [8] that it is for the best to calculate the tightest possible bounding box from complete hair geometry no matter additional time consumption. Therefore we generate hair geometry twice, first before rendering to calculate the bounding box and then during rendering we generate the hair again for actual rendering purpose. Since we have split hair generation during

rendering to several library calls, we have to calculate the bounding box for each call separately, which is done in parallel.

Generating hair twice could be avoided if we generated hair before rendering and then supplied it to a renderer directly with the rest of a rendered scene. This approach seems logical but it has one huge pitfall. We would have to save all hair geometry to a scene file. Since there can be millions of hairs, the scene file size could grow to several gigabytes for a single frame. Because the hair generation is very efficient (which is partly thanks to the speed of our point generation algorithm described in Section 3), handling such files causes much worse performance than generating hair twice, especially when these files usually have to be moved from a 3d artist workstation to computers dedicated only to rendering.

There are many parameters that influence hair generation. The modeling part of Stubble is responsible for saving them to files which are then used by our library during rendering. There are two types of these files. First of them stores properties like hair color or width that are shared among all library calls. The second type of file stores data specific for one library call, such as the selected part of the mesh and the number of hair generated by this library call. All input files are compressed to save disk space. Since Stubble enables interactive display of hair during modeling, hair parameters may be send to our library directly from the modeling part of Stubble without the need of creating any files.

4.2.2 Hair Generation

In this section we describe how every single hair is generated by our library. Before we start describing hair generation, it is important to know how we represent each hair. The most flexible way is to handle each hair as the Catmull-Rom curve, an interpolation curve defined only by the vertices it passes through. Furthermore, we specify for each of these vertices color, opacity, curve width and curve normal (an unit vector perpendicular to the curve tangent at a given vertex), which are then interpolated along the curve by a renderer.

The simplified flowchart in Figure 14 shows the generation of a single hair. It starts with creating a sample on the triangular mesh with a density defined by a texture as described in Section 3. The sample serves as the position of a hair's root. The generation of the sample must be very fast, otherwise it would be a bottleneck in the generation of hair.

When we know the position of hair, we generate its basic geometry by interpolation from few closest hair guides. To determine the closest hair guides we use euclidean distances of hair root from hair guides roots. To speed up this process, we store hair guides roots in a KD-Tree [3]. We have already mentioned that each hair is a curve defined by vertices and each hair guide is a polyline and therefore also represented by vertices. The hair curve and each hair guide

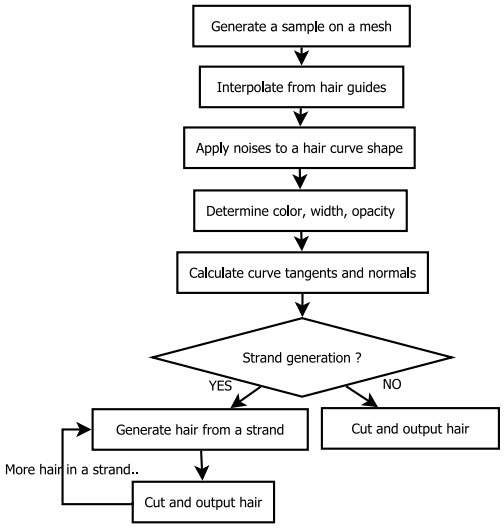


Figure 14: Flowchart of the generation of a single hair.

has the same number of vertices, so we can easily interpolate the hair curve by interpolating its vertices from corresponding hair guides vertices. To interpolate vertices we use *Scattered Data Interpolation*, specifically the Shepard method [9], which gives each guide hair a weight based on the distance from the interpolated hair.

Interpolating the hair curve from hair guides is not enough to make generated hair realistic, because the number of generated hair is far greater than the number of hair guides. To improve hair realism, we randomize each hair curve by adding random vectors generated from Perlin noise [5] to hair curve's vertices. The number of vectors used on a single hair, vectors size and noise frequency are user parameters. Figure 15 shows an example of a noise influence on hair.

Figure 15: An example of a noise influence on straight hair.

The next step is to define hair color, opacity and width. The user specifies these attributes at hair root and tip and we interpolate them to every hair vertex. Again we add some noise to hair color to increase realism.

Finally we have to calculate hair curve normals. Since curves are usually generated by a renderer as flat ribbons, we have to define the rotation of the ribbon about the curve. This is done by the curve normals. We use the method described in [11] to calculate reasonable normals. Computation of normals requires as input curve tangents, which are easily calculated for a Catmull-Rom curve from its vertices [1].

A single hair is now completely specified, we can either output it as is or we can use it to generate a whole

strand of hair. The user can specify if strands should be generated and how many hairs are in a single strand. If we generate a hair strand, we use the just generated hair as a basic hair around which all hair from this strand is generated. Each hair from strand inherits its properties from the basic hair and it is also influenced by the basic hair geometry. Furthermore, user can define several parameters of hair strands, that control the spread of hair roots and tips from the basic hair, twisting of hair around the basic hair and much more. Figure 16 demonstrates hair strands.

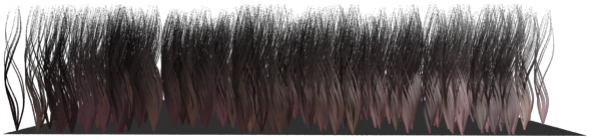


Figure 16: Generated hair strands.

Just before we output each hair to a renderer, we cut it. How much is each hair cut is defined by a texture mapped on the mesh from which the hair grows. The texture value specifies a curve parameter at which the curve is cut. Every user parameter controlling hair generation is also specified by a texture, which gives the user the ability to set different parameters for each hair.

Figure 17 shows hair generated by our library. There is 1,600,000 hair on this animal and it took only 3.2 seconds to generate it on a two core 3.07 GHz PC.

Figure 17: An animal with 1,600,000 hair generated by our library.

5 Conclusions and Future Work

In summary, we have presented a sampling algorithm suitable for fast sampling on triangular meshes. The density of samples generated by our algorithm is defined by a two-dimensional texture. In our tests, our algorithm achieves a 3 – 33 speedup compared to the fastest available alternative - the rejection sampling.

Furthermore, we have described a hair generator library that is able to generate millions of hairs in a few seconds during rendering. Our library utilizes the presented sampling algorithm for a very fast placement of hair roots. Hair generated by our library is influenced by several properties and by a few hairs modeled directly by a 3d artist. Our library is also able to display hair interactively in a modeling program.

There are several limitations of our work that should be addressed in future work. First, the uniformity of the generated samples should be improved. Second, we would like to add several modifications to our hair generator library, such as generation of low quality hair if an object with hair is motion blurred or is far away from a scene camera.

Acknowledgments

I would like to thank Jaroslav Křivánek for his useful comments and advice during the creation of this work.

References

[1] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., 2008.

[2] John Bowers, Rui Wang, Li-Yi Wei, and David Maletz. Parallel poisson disk sampling with spectrum analysis on surfaces. *ACM Trans. Graph.*, 29:166:1–166:10, 2010.

[3] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Ltd., 2001.

[4] Johannes Kopf, Daniel Cohen-Or, Oliver Deussen, and Dani Lischinski. Recursive wang tiles for real-time blue noise. *ACM Trans. Graph. (Proceedings of SIGGRAPH 2006)*, 25(3):509–518, 2006.

[5] Ken Perlin. Improving noise. *ACM Trans. Graph.*, 21(3):681–682, 2002.

[6] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., 2010.

[7] Lijun Qu and Gary W. Meyer. Perceptually driven interactive geometry remeshing. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, pages 199–206, 2006.

[8] David Ryu. 500 Million and counting: Hair rendering on Ratatouille. Technical report, Pixar, May 2007.

[9] Donald Shepard. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM National Conference*, pages 517–524, 1968.

[10] Greg Turk. Texture synthesis on surfaces. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pages 347–354, 2001.

[11] Wenping Wang, Bert Jüttler, Dayue Zheng, and Yang Liu. Computation of rotation minimizing frames. *ACM Trans. Graph.*, 27:2:1–2:18, 2008.