Real-Time Hair Rendering with Hair Meshes

Gaurav Bhokare
bhokare.gaurav@utah.edu
University of Utah
USA

Eisen Montalvo
eisen.montalvo@utah.edu
University of Utah
USA

Elie Diaz
elie.diaz@utah.edu
University of Utah
USA

Cem Yuksel
cem@cemyuksel.com
Cyber Radiance
USA

Figure 1: *A scene with 100 characters, each with a unique hair model comprising of 100 thousand strands, rasterized in only 2 ms on an NVIDIA GTX 4090 GPU (with 8× MSAA) using our real-time hair rendering method with hair meshes and our level-of-detail techniques. All 100 hair mesh models in this scene fit in 1.7 MB (between 13 KB and 21 KB per model).*

ABSTRACT

Hair meshes are known to be effective for modeling and animating hair in computer graphics. We present how the hair mesh structure can be used for efficiently rendering strand-based hair models on the GPU with on-the-fly geometry generation that provides orders of magnitude reduction in storage and memory bandwidth. We use mesh shaders to carefully distribute the computation and a custom texture layout for offloading a part of the computation to the hardware texture units. We also present a set of procedural styling operations to achieve hair strand variations for a wide range of hairstyles and a consistent coordinate-frame generation approach to attach these variations to an animating/deforming hair mesh. Finally, we describe level-of-detail techniques for improving the performance of rendering distant hair models. Our results show an unprecedented level of performance with strand-based hair rendering, achieving hundreds of full hair models animated and rendered at real-time frame rates on a consumer GPU.

This work is licensed under a Creative Commons Attribution International 4.0 License.

SIGGRAPH Conference Papers '24, July 27–August 01, 2024, Denver, CO, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0525-0/24/07
https://doi.org/10.1145/3641519.3657521

CCS CONCEPTS

• **Computing methodologies** → **Rasterization**.

KEYWORDS

Real-time rendering, hair rendering, hair modeling, hair meshes.

ACM Reference Format:
Gaurav Bhokare, Eisen Montalvo, Elie Diaz, and Cem Yuksel. 2024. Real-Time Hair Rendering with Hair Meshes. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Papers '24 (SIGGRAPH Conference Papers '24), July 27–August 01, 2024, Denver, CO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3641519.3657521

1 INTRODUCTION

Hair is a crucial visual component of virtual characters. That is why it has received a substantial amount of attention in graphics in the context of modeling [Chai et al. 2016; Daldegan et al. 1993; Wang et al. 2009; Yuksel et al. 2009a], animation [Chai et al. 2014; Hsu et al. 2022, 2023; Wu and Yuksel 2016], and rendering [Marschner et al. 2003; Yan et al. 2015; Yuksel and Tariq 2010]. In all of these, the geometric complexity of hair is one of the leading challenges. This is because most humans have on the order of 100 thousand hair strands and, depending on the length and style, each may require a curve with many control points. Therefore, a typical full hair model can easily exceed a million vertices, making it particularly expensive for real-time rendering.

In this paper, we present how the *hair mesh* structure [Yuksel et al. 2009a] can be used for accelerating real-time strand-based hair rendering on the GPU and managing its geometric complexity (Figure 1). Hair meshes were initially introduced as a method for modeling hair. A hair mesh is a volumetric structure formed by extruding polygonal faces of a scalp model. The individual hair strands are generated within these extrusions, which are then modified using a variety of *styling* operations that define the geometric variations of hair strands. This allows the user to precisely specify the overall shape of a hair model, circumventing the geometric complexity of individual strands. In addition, hair meshes allow automatically placing the internal vertices of the volumetric structure, allowing the user to concentrate on the external surface of a hair model, thereby bringing hair modeling close to polygonal modeling of typical surfaces, commonplace in computer graphics.

Later on, hair meshes were shown to be effective for fast and stable hair simulation [Wu and Yuksel 2016], though we concentrate on the real-time rendering problem in this paper.

We generate hair strands during rendering within GPU shaders using a given hair mesh and a set of styling parameters that control the procedural functions specifying strand variations. This avoids the need for storing strand-based hair data, which can easily take hundreds of megabytes, and updating it as the hair moves/deforms. Instead, hair motion is captured by simulating the hair mesh and animating the styling parameters, reducing the amount of data to be stored and managed by several orders of magnitude.

Our hair geometry generation process follows the steps of the original hair mesh modeling framework [Yuksel et al. 2009a] with some key differences for facilitating real-time rendering:

- We carefully organize the computation workload to match the SIMD parallelization of mesh/compute shaders.
- We introduce a novel texture-space layout of the hair mesh data for utilizing the hardware texture units to efficiently perform the interpolation operations and spline generation within the volumetric embedding of the hair mesh.
- We propose a consistent method for local coordinate-frame generation needed for applying styling variations that adapt to arbitrary deformations of the hair mesh.
- We describe procedural styling operations that can generate a wide variety of hairstyles from a small set of parameters (the details are included in the supplemental document).
- We present level-of-detail techniques that can automatically reduce the geometry to be generated, allowing a large collection of hair models to be efficiently rendered.

Our results show that our framework can render hundreds of unique strand-based hair models (without instancing) at real-time frame rates on current GPUs, offering an unprecedented geometric complexity for real-time hair rendering on high-end GPUs of today and making strand-based hair rendering a more affordable option for lower-end devices. An example from our tests is presented in Figure 1, showing a scene with 100 characters with full-resolution strand-based unique hair models with individual animations, (simulated and) rendered at real-time frame rates.

2 BACKGROUND

Before we describe the details of our method, in this section, we present the related prior work on real-time hair rendering (Section 2.1) and the details of the hair mesh structure (Section 2.2).

2.1 Related Work

The immense geometric complexity of hair has always been a challenge for real-time rendering. Earlier methods avoided this complexity by representing hair as a surface with a texture [Scheuermann 2004], severely limiting the quality and the realism of the rendered models. As GPUs became more powerful, strand-based hair rendering emerged as an attractive, though expensive, alternative [Chai et al. 2014; Ren et al. 2010; Tariq and Bavoil 2008; Xu et al. 2011; Yu et al. 2012; Yuksel and Keyser 2008; Zinke et al. 2008].

The geometric complexity of strand-based hair rendering not only requires a substantial amount of computation but also incurs the cost of data movement when each strand is explicitly stored. Besides the additional storage cost, this data movement can easily become the bottleneck of GPU rendering performance.

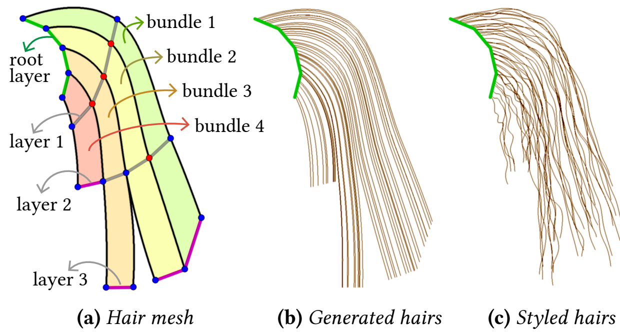
One solution for significantly reducing the cost of data movement is generating hair strands on-the-fly during rendering [Yuksel and Tariq 2010]. Obviously, this approach also dictates how the full hair model and its animation are defined.

A common approach is using a small number of *guide hairs* that are used for determining the shapes of other hair strands. However, without any additional information, this can only generate hairs that simply interpolate the guides [Yuksel and Tariq 2010]. Therefore, the state-of-the-art methods first generate all hair strands, then for each hair vertex compute interpolation weights of neighboring guide hairs, and finally use these weights for *linear hair skinning* (LHS) [Games 2021; Somasundaram 2015] during rendering to generate the full-resolution hair model by interpolating these guide hairs. This approach is very effective for efficiently animating strand-based hairstyles by only computing the deformations of the guide hairs. However, it must still store the interpolation weights, which can easily take as much space as storing the individual hair vertex positions. That is why rendering full-resolution hair models remains expensive in practice for real-time graphics applications.

Our solution utilizes on-the-fly hair generation using hair meshes [Yuksel et al. 2009a] that avoids the cost of storing the full-resolution hair model. A similar approach is also used for fiber-level rendering of yarn-based cloth models on the GPU [Wu and Yuksel 2017a,b].

Level-of-detail methods are ubiquitous in real-time rendering [Liu et al. 2017; Mercier et al. 2022; Zhu et al. 2022]. They have also been used for real-time hair rendering [Yuksel and Tariq 2010] and they are easy to incorporate with on-the-fly hair generation. Our approach for level-of-detail is similar to existing methods, but includes some critical details to avoid artifacts when transitioning between detail levels and properly utilizing GPU parallelism.

Some recent work presents hair rendering methods using neural networks as a promising new direction [Chai et al. 2020; Rosu et al. 2022; Wei et al. 2018]. While these methods are slower than LHS for rasterizing 3D hair models and consume even more memory, they offer advantages in generating realistic hair appearance. Our method can be utilized in future neural hair rendering pipelines that rely on strand-based hair rasterization.



(a) *Hair mesh* **(b)** *Generated hairs* **(c)** *Styled hairs*

Figure 2: *A 2D representation of (a) a hair mesh with 4 bundles extruded from a scalp mesh at the root level, (b) the hair strands generated from the hair mesh, and (c) the strands after applying procedural styling operations (source: [Yuksel et al. 2009a]).*

2.2 The Hair Mesh Structure

The hair mesh structure [Yuksel et al. 2009a] can be considered as layers of extrusions on a polygonal scalp model, as shown in Figure 2. The scalp mesh is called the *root layer*. Multiple layers of extrusions starting from one scalp face form a *bundle*. Thus, each root-layer face corresponds to a different bundle.

The individual hair strand curves are generated within these extruded volumes. The root position of each hair strand corresponds to a barycentric coordinate on a face of the scalp model. This forms the first control point of the strand curve. The other control points are formed using the same barycentric coordinate on successive extrusion layers from the same scalp face. Yuksel et al. [2009a] uses cubic Catmull-Rom splines [Catmull and Rom 1974] with centripetal parameterization [Yuksel et al. 2009b] to form an interpolating curve from these control points, though other curve formulations can be used as well. The vertices of the hair mesh on successive extrusions over the same root vertex are connected using the same curve formulation, forming curved edges along the hair growth direction and corresponding curved external surfaces.

Since hair meshes form volumetric structures, many of their vertices are hidden behind their external surfaces. For simplifying the volumetric modeling process, these internal vertices are placed automatically, based on the user-specified positions of the external hair mesh vertices that appear on their external surfaces.

The individual hair strands generated from a hair mesh, as described above, form a uniform hair flow within the hair mesh volume, lacking any variation between neighboring strands. Such variations are introduced via a set of *styling* operations. These are typically random perturbations or procedural functions applied to the vertices of a hair strand, which are initially placed along the strand curve generated from the hair mesh. Yuksel et al. [2009a] uses the barycentric embedding of a hair strand within the hair mesh for defining the coordinate frame in which these styling operations are applied, so that these variations follow the deformations of an animating hair mesh.

The on-the-fly hair generation method we describe in this paper follows the same high-level process, but differs in details, as it is customized for the computation process of the GPU cores and for utilizing texture filtering units to offload some computations.

3 RENDERING WITH HAIR MESHES

Our strand-based hair rendering approach converts the hair mesh into a special texture that allows using the texture filtering hardware for accelerating the generation of a hair strand curve from a hair mesh (Section 3.1). We use mesh/compute shaders to generate hair strands from the given hair mesh during rendering (Section 3.2). This involves placing the hair strand roots (Section 3.3), perturbing their vertices via a series of styling operations (Section 3.4), and computing their tangent directions (Section 3.5), which is needed for shading. For further acceleration, we apply level-of-detail by dynamically reducing the number of hair strands we generate and the vertex count per hair based on the camera distance (Section 3.6).

3.1 The Hair Mesh Texture

We use a ***hair mesh texture*** for storing the vertex positions of the given hair mesh (Figure 3). It is common to send various types of custom data to the GPU shaders using textures. Our **hair mesh texture**, however, is structured to take advantage of the texture filtering hardware available on the GPU to perform some of the interpolation computations we need during hair strand generation.

Without loss of generality, we design our approach for quad-dominant hair meshes that are formed by extruding a scalp mesh of quads and triangles. Note that any scalp face (and its bundle) that is a higher-degree polygon can be trivially split into quads/triangles.

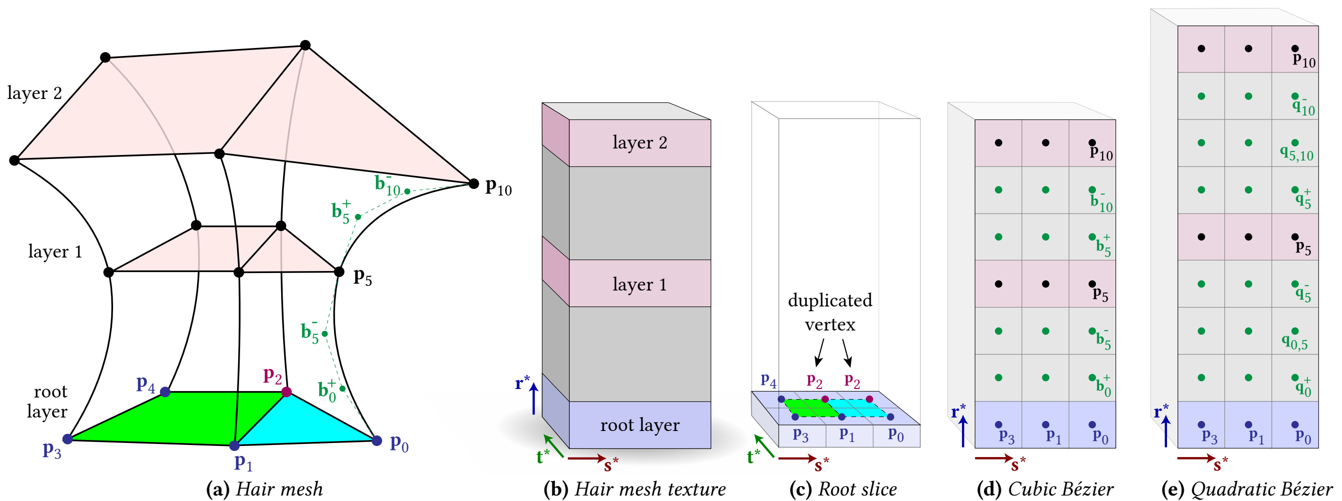
The **hair mesh texture** is stored as a standard 3D texture on the GPU, with each texel storing a 3D position in object space. We use three vectors $\mathbf{s}^\star$, $\mathbf{t}^\star$, and $\mathbf{r}^\star$ to denote the orthogonal coordinate frame of this texture (Figure 3b). The texels on its first 2D slice perpendicular to the $\mathbf{r}^\star$ direction correspond to the root layer of the hair mesh and we call it the *root slice* (Figure 3c). The texels of the root slice store the root-layer vertex positions (i.e. vertex positions of the scalp mesh). For each quad face of the root layer, the vertex positions are copied to the texels of a 2×2 block on the root slice. Triangle faces are handled by duplicating one of their vertices and similarly copying onto 2×2 blocks.

This allows using the texture filtering hardware to compute a bilinear interpolation of the four vertices of a quad face during hair generation. Given a point $\mathbf{p}^\star$ on the root slice of the **hair mesh texture** between the centers of a 2×2 block of texels (the highlighted green area in Figure 3c), sampling this texture at $\mathbf{p}^\star$ returns the corresponding object-space coordinate $\mathbf{p}$ by bilinearly interpolating these four root-layer vertex positions.

This process also works for triangle faces of the scalp by adjusting the sampling position, as we explain in Section 3.3.

The vertex positions of the hair mesh on successive layers are copied onto other 2D slices of the **hair mesh texture**, which we call *layer slices*, with each layer corresponding to a specific slice perpendicular to the $\mathbf{r}^\star$ direction of the 3D texture (e.g. $\mathbf{p}_5$ and $\mathbf{p}_{10}$ positions in Figure 3 are copied to texels on slices marked as *layer 1* and *layer 2*, respectively). This allows using hardware trilinear interpolation to compute the 3D position within the hair mesh volume.

Yet, hair strands are cubic Catmull-Rom splines, so we cannot rely on a single trilinear interpolation between layers of a hair mesh to compute a position along the curve. For supporting cubic spline interpolation, we convert each segment of the Catmull-Rom



(a) *Hair mesh* **(b)** *Hair mesh texture* **(c)** *Root slice* **(d)** *Cubic Bézier* **(e)** *Quadratic Bézier*

Figure 3: *An example hair mesh model and its hair mesh texture: (a) the hair mesh, (b) its 3D hair mesh texture, (c) the 2D root slice of this hair mesh texture, (d) the first 2D slice along the* $\mathbf{t}^\star$ *direction with cubic Bézier control points, and (e) the same slice with quadratic Bézier control points that are generated by splitting each cubic Bézier segment into two quadratic Bézier curves. In this example, the cubic Bézier curve with control points* $\mathbf{p}_0$, $\mathbf{b}_0^+$, $\mathbf{b}_5^-$, *and* $\mathbf{p}_5$ *is converted two quadratic Bézier curves using* $\mathbf{p}_0$, $\mathbf{q}_0^+$, *and* $\mathbf{q}_{0,5}$ *for the first one and* $\mathbf{q}_{0,5}$, $\mathbf{q}_5^-$, *and* $\mathbf{p}_5$ *for the second.*

spline into a cubic Bézier curve with 4 control points (Figure 3a). Then, control points can be placed on four successive *intermediate slices* of the 3D texture, placed between layer slices (the gray layers shown in Figure 3d). Then, any point along the curves within a bundle of the hair mesh can be computed by three trilinear interpolations (between three successive slices) followed by three linear interpolations, using de Casteljau's algorithm. This way, the texture filtering unit performs most of the computation needed for both bilinear interpolations within the bundle and the evaluation of the interpolated hair spline at any point along the hair.

For further optimization, we approximate each cubic Bézier segment as two quadratic Bézier segments using the method of Truong et al. [2020]. For each quadratic segment, it is sufficient to perform two trilinear interpolations using the texture-filtering hardware followed by a single linear interpolation in software. Thus, our 3D **hair mesh texture** is formed as shown in Figure 3e with three intermediate slices between each pair of root/layer slices.

In general, the topology of an arbitrary scalp mesh may not be suitable for simply copying all its vertex positions on the 2D lattice of the root slice. Such cases can be easily handled by splitting the scalp mesh into multiple pieces to be copied onto the **hair mesh texture** separately.

We assume that the mapping of the hair mesh root layer onto the root slice of the **hair mesh texture** is prepared manually (analogous to texture mapping of common surfaces), though it can be automated, while the successive layers along the $\mathbf{r}^\star$ direction are handled automatically based on the given 2D mapping.

3.2 Hair Generation on the GPU

We generate hair strands on the GPU during rendering. These strands are re-generated for each render pass and never stored. This means repeating some computations that could otherwise be shared between passes and even frames. On the other hand, eliminating the storage of hair strands, which can easily take hundreds of megabytes for a single hair model, is a substantial saving that more than justifies repeated computation during rendering. Nonetheless, to achieve the best trade-off, we must carefully organize the hair generation workload in a way that matches the computation flow of the GPU hardware. Below we describe our solution for parallel hair generation process using GPU mesh/compute shaders.

Standard methods for on-the-fly hair generation on the GPU use tessellation shaders. We can use tessellation shaders with our method as well, though this approach has strict limits on the number of hair vertices and the number of hair strands that can be generated per patch. Therefore, we favor mesh/compute shaders instead.

Compute shaders are preferred if hair generation will be followed by software rasterization. This might be favorable, since hair strands often turn into many small triangles that can be rendered more efficiently in software than hardware rasterizers on current GPUs. Our implementation, however, relies on the hardware rasterizer, so we describe our hair generation method using mesh shaders.

Nonetheless, the process we describe below can be applied to compute shaders as well, as we explain at the end.

We utilize 3 levels of parallelism offered by hair meshes:

(1) **Bundle-level:** A hair mesh contains multiple bundles.
(2) **Strand-level:** Each bundle contains multiple hair strands.
(3) **Vertex-level:** Each hair strand contains multiple vertices.

Our mesh shaders execute at the strand-level parallelism. We match the computation load to the operation of GPU cores that work in SIMD blocks (i.e. *wavefronts* in AMD and *warps* in NVIDIA hardware) of S threads, where S varies by hardware (typically 32 or 64). To maximize occupancy we split our mesh shader workload into groups of s threads, where S is a multiple of s.

Each mesh shader thread in a SIMD block computes n vertices, where a hair strand is formed by $ns+1$ vertices. Computing a vertex involves evaluating its initial curve position within the hair mesh by sampling the **hair mesh texture** multiple times followed by

linear interpolation (as explained above in Section 3.1) and then applying procedural styling operations. One exception is the root vertex of the strand, which only requires a single lookup of the **hair mesh texture** on the root slice without applying any other interpolation or styling perturbation, since styling is not applied to the root vertices. The resulting hair strand curves are rendered as camera-facing triangle strips each with $2ns + 2$ vertices and $2ns$ triangles.

We spawn a task shader for each hair mesh bundle. Our task shader determines the number of hair strands to be generated within its bundle, which may involve culling and level of detail, as explain in Section 3.6. Then, it spawns as many mesh shaders as the number of hair strands to be rendered for that bundle. Thus, task shaders work with bundle-level parallelization, strand-level parallelization is exploited with different mesh shader instances, and the SIMD cores running a mesh shader utilize the vertex-level parallelization.

The main challenge with a compute shader implementation is handling our task shader operations that determine the number of hair strands per bundle and the total number of compute shader executions. This can be performed on the CPU prior to dispatching the compute shaders or it can be handled as an initial pass with compute shaders on the GPU.

Then, compute shaders can operate similarly to our mesh shaders, followed by software rasterization. An alternative implementation of compute shaders can switch between our task shader and mesh shader computations, which would require implementing a software schedular. We have not explored these compute shader alternatives in our tests. Future work can determine which option would be preferable for a given workload and GPU hardware.

3.3 Hair Root Placement

A hair strand is uniquely defined by its root position within the hair mesh. Therefore, for placing a hair strand within a bundle, we simply need to pick a random root location for it. We do so using a precomputed set of 2D sample locations $\boldsymbol{\xi} = [\xi_s \ \xi_t]^T$ within $[0, 1]^2$. A sample is chosen using the local index of a hair strand and mapped onto the root slice of the hair bundle.

To achieve a natural placement of hair roots, we use a blue noise distribution for precomputing the 2D sample locations, generated by *sample elimination* [Yuksel 2015]. This method allows us to generate a progressive sample set, such that any number of samples taken from the beginning of the set forms blue noise. In our implementation, we use the same set of random samples for all bundles. To achieve more variety multiple sets of random samples can be precomputed, each bundle using a different set. For picking root positions on triangle scalp faces, we use a different set of precomputed 2D sample locations. This is because using the sample set generated for a quad face on a triangle would bias the hair root locations towards the duplicated vertex in the **hair mesh texture** layout. Instead, we generate the sample set for triangles using sample elimination within $\boldsymbol{\xi} \in [0, 1]^2$ but by first selecting samples within a triangular half, such that $\xi_s + \xi_t < 1$. Then, we map them onto $\boldsymbol{\xi}^\triangle = [\xi_s^\triangle \ \xi_t^\triangle]^T \in [0, 1]^2$ within the full quad space, such that $\xi_s^\triangle = \xi_s(1 - \xi_t)$ and $\xi_t^\triangle = \xi_t$. Using bilinear interpolation with $\boldsymbol{\xi}^\triangle$ corresponds to barycentric interpolation with coordinates ξ_s, ξ_t, and $1 - \xi_s - \xi_t$, where the barycentric coordinate ξ_t corresponds to the duplicated vertex on the **hair mesh texture** layout.

Since our precomputed sample sets are generated for a square shape, the blue noise characteristics of the distribution weaken when mapped onto a skewed or non-uniformly-scaled quad/triangle of the scalp surface. Nonetheless, this approach still produces a more natural hair root distribution with close-by hair roots mostly avoided, as compared to simply using random numbers with a uniform distribution (i.e. white noise)

3.4 Styling Coordinates

Hair styling is a crucial process, altering the shapes of individual hair strands and specifying the geometric variations between them. It is typically applied as a set of procedural functions or random offsets altering hair vertex positions. We include the details of the styling operations we use in our implementation in our supplemental document. Other styling operations can be used as well.

The most important component of the styling process is defining the local *styling coordinates* that would remain consistent as the hair mesh deforms. The styling perturbations of hair vertices are defined within these local styling coordinates. The amplitudes of the styling perturbations, however, are defined in the object space, so that they are not scaled by expansions/contractions of the hair mesh bundles, which can happen when the hair mesh is animated.

We achieve this by defining a texture-space embedding of the hair mesh. Similar to standard texture mapping, hair mesh vertices at the root layer are placed on a 2D texture space by specifying their uv coordinates. Just like texture mapping, seams may be introduced and some root layer vertices along seams can be placed at multiple locations within this 2D texture space. This process is handled manually or simply copied from the uv layout of the scalp mesh.

For handling the extruded layers of the hair mesh, we extend this texture space to 3D with uvw coordinates. Extrusions of the hair mesh bundles take their uv coordinates from their corresponding root layers, but are assigned a different w coordinate, such that $w = 0$ at the root layer with increasing w towards the tip layers. We assume that these w coordinates are provided as a part of the hair mesh, but they can also be automatically generated based on the length of the path from each vertex to its root.

We record the texture-space coordinates of a hair mesh in two textures. The first one is our ***uv*-texture**, a 2D version of the **hair mesh texture**, storing only the uv coordinates. The layout of this 2D *uv*-**texture** matches the root slice layout of the **hair mesh texture** (Figure 3c), but it only records the uv values of the hair mesh. The second one is our ***w*-texture**, which is a 3D texture similar to our **hair mesh texture** but only stores the scalar w values. Unlike the **hair mesh texture**, however, *w*-**texture** needs no intermediate slices. For a given position $\mathbf{p}^\star$ in the **hair mesh texture**, we can easily sample these two textures to find the corresponding uvw coordinate.

Most styling operations can be computed based on this uvw coordinate. However, the resulting styling perturbation directions $\mathbf{d}^\tau$ are defined in this 3D texture space. Therefore, we must

transform them to the corresponding object-space directions $\mathbf{d} = \mathbf{M}_{\mathbf{p}^\star}\mathbf{d}^\tau$ using a transformation matrix

$$\mathbf{M}_{\mathbf{p}^\star} = \begin{bmatrix} \hat{\mathbf{u}} & \hat{\mathbf{v}} & \hat{\mathbf{w}} \end{bmatrix}$$

(a) *Finite differences* **(b)** *Ours*

Figure 4: *Computing styling coordinates (a) using finite differences can lead to styling discontinuities, visible as a vertical seam down the middle in this example, (b) our solution avoids such seams.*

where $\hat{\mathbf{u}}$, $\hat{\mathbf{v}}$, and $\hat{\mathbf{w}}$ are the object-space vectors that correspond to the texture-space $\hat{\mathbf{u}}^\tau$, $\hat{\mathbf{v}}^\tau$, and $\hat{\mathbf{w}}^\tau$ directions at position $\mathbf{p}^\star$ of the **hair mesh texture**.

For computing these object-space vectors $\hat{\mathbf{u}}$, $\hat{\mathbf{v}}$, and $\hat{\mathbf{w}}$, a simple solution would be using finite difference by sampling the **hair mesh texture** around $\mathbf{p}^\star$. Yet, besides the numerical issues of finite difference, this approach would not produce desirable results, because it would not form a continuously rotating coordinate frame within the hair mesh. This is because the *uvw* coordinates vary linearly within a bundle and object-space $\hat{\mathbf{u}}$, $\hat{\mathbf{v}}$, and $\hat{\mathbf{w}}$ directions computed for a bundle can be arbitrarily different from the ones within a neighboring bundle. Therefore, using finite differences would lead to styling perturbations that change discontinuously, forming visible seams between hair mesh bundles, as shown in Figure 4.

To ensure a continuous transformation of styling perturbations, we define consistent object-space $\hat{\mathbf{u}}$, $\hat{\mathbf{v}}$, and $\hat{\mathbf{w}}$ directions for each vertex of the hair mesh, which are shared by all neighboring bundles using the same vertex. The $\hat{\mathbf{u}}$ direction of a hair mesh vertex is defined as the weighted average of all linearly-transformed directions $\hat{\mathbf{u}}_i$ determined by each triangle i that surrounds the vertex (i.e. triangles containing the vertex and its two edges) at the same layer of the hair mesh, as shown in Figure 5. Thus,

$$\hat{\mathbf{u}} = \frac{\mathbf{u}}{\|\mathbf{u}\|} \qquad \text{with} \qquad \mathbf{u} = \frac{\sum_i A_i \mathbf{u}_i}{\sum_i A_i}\,, \tag{1}$$

where A_i is the texture-space area of the triangle i.

This linear transformation for a triangle i can be represented by a 2×2 matrix $\mathbf{T}_i$, such that

$$\mathbf{T}_i = \begin{bmatrix} u_1 - u_0 & u_2 - u_0 \\ v_1 - v_0 & v_2 - v_0 \end{bmatrix} \tag{2}$$

where u_j and v_j with $j \in \{0, 1, 2\}$ are the *uv* coordinates of the three vertices of the triangle. Let $\mathbf{p}_j$ be their object-space positions. We transform the texture-space $\hat{\mathbf{u}}^\tau$ and $\hat{\mathbf{v}}^\tau$ for this triangle using

$$\begin{bmatrix} \mathbf{u}_i & \mathbf{v}_i \end{bmatrix} = \begin{bmatrix} \mathbf{p}_1 - \mathbf{p}_0 & \mathbf{p}_2 - \mathbf{p}_0 \end{bmatrix} \mathbf{T}_i^{-1}\,. \tag{3}$$

Note that this process does not guarantee that $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$ directions at a hair mesh vertex are perpendicular, thus the transformation of

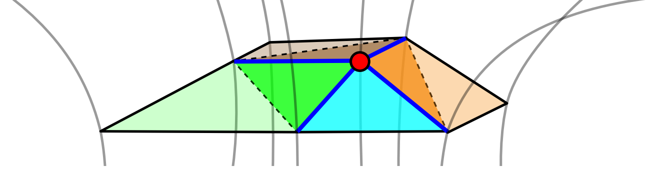


Figure 5: *Four triangles surrounding a vertex (highlighted in red) at the same hair mesh layer. Notice that each triangle contains the vertex and two edges coming out of the vertex, highlighted in blue.*

the resulting coordinate frame may include skew. We define $\hat{\mathbf{w}}$ as the orthogonal direction to both $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$, such that

$$\hat{\mathbf{w}} = \frac{\hat{\mathbf{u}} \times \hat{\mathbf{v}}}{\|\hat{\mathbf{u}} \times \hat{\mathbf{v}}\|}\,. \tag{4}$$

We precompute the $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$ directions for the hair mesh vertices and store them in two separate 3D textures, **u-*texture*** and **v-*texture***, which are stored similarly to the **hair mesh texture** by recording the computed $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$ vectors instead of positions. Just like our *w*-**texture**, we do not need intermediate slides for these two textures either. Thus, $\mathbf{u}$ and $\mathbf{v}$ are obtained by single lookups.

Table 1 lists all 5 textures used in our system. As the hair mesh deforms, we must update the textures that contain object-space positions and directions:

hair mesh texture, **u-texture**, and **v-texture**. The other textures (*uv*-**texture** and *w*-**texture**), do not need any dynamic updates.

Table 1: *All 5 textures we use for representing hair meshes.*

Texture	Dim.	Texel Data
hair mesh texture	3D	Object-space vertex position
uv-**texture**	2D	Root vertex *uv* styling coordinate
w-**texture**	3D	Vertex *w* styling coordinate
u-texture	3D	Object-space styling direction $\mathbf{u}$
v-texture	3D	Object-space styling direction $\mathbf{v}$

3.5 Hair Tangents

We need the hair tangent directions for shading. It may be possible to compute them using finite differences or the analytical derivatives of the procedural styling operations to determine the infinitesimal tangent direction at each hair vertex we compute. However, in addition to the extra computation cost of evaluating such a tangent direction, it may not be representative of the final hair strand shape we generate. This is because the styling functions we use can have a higher frequency than what we can reliably represent using the number of vertices we compute. Therefore, we favor using a cheaper and simpler scheme that we describe below.

Let $k \in \{0, 1, \cdots, ns\}$ represent the index of a vertex along a hair strand, where $k = 0$ corresponds to the root vertex.

We assign the tangent direction $\mathbf{t}_k$ as the vector from the previous vertex to the next vertex, such that $\mathbf{t}_k = \mathbf{p}_{k+1} - \mathbf{p}_{k-1}$, except for the first and the last vertices. The tangent of the last vertex is set such that its angular separation with the last hair segment $\overline{\mathbf{p}_{ns-1}\mathbf{p}_{ns}}$ matches the angle of the previous tangent $\mathbf{t}_{ns-1}$, using

$$\hat{\mathbf{t}}_{ns} = 2\hat{\mathbf{q}}_{ns}\left(\hat{\mathbf{t}}_{ns-1} \cdot \hat{\mathbf{q}}_{ns}\right) - \hat{\mathbf{t}}_{ns-1} \tag{5}$$

where $\hat{\mathbf{q}}_k = (\mathbf{p}_k - \mathbf{p}_{k-1})/\|\mathbf{p}_k - \mathbf{p}_{k-1}\|$ is the last segment direction and $\hat{\mathbf{t}}_k = \mathbf{t}_k/\|\mathbf{t}_k\|$. Similarly, the tangent of the first vertex is set as

$$\hat{\mathbf{t}}_0 = 2\hat{\mathbf{q}}_1 \left(\hat{\mathbf{t}}_1 \cdot \hat{\mathbf{q}}_1\right) - \hat{\mathbf{t}}_1 \ . \tag{6}$$

This particular way of defining the tangents of the endpoints is important to achieve consistent shading when we change the number of hair vertices with level-of-detail, as we describe below.

3.6 Level-of-Detail

On-the-fly hair generation allows two level-of-detail (LOD) types by dynamically reducing the number of hair strands and the number of vertices per hair strand.

We use a simple scheme for determining the detail level L for either of the LOD types. It is based on the minimum distance d to the camera, the vertical image resolution R, and a user-defined scaling parameter α that controls the performance/quality trade-off, such that

$$L = \min\left(1, \frac{hR}{d \tan\theta}\alpha\right) \ , \tag{7}$$

where θ is the camera's field of view and h is the world-space hair model size that can be interpreted differently for the two LOD types. When using an orthographic camera, such as when rendering shadows for a directional light, the $d \tan\theta$ term can be replaced by the view height in world space.

The number of hair strands in each bundle and the number of vertices per hair strand are scaled by L, subject to some restrictions described below. The main challenge with level-of-detail is ensuring seamless transitions between detail levels.

The α value that would provide a good trade-off between quality and performance can be different for different hairstyles. Also, it would be favorable to use two different α parameters for the two LOD types, so that they can be tuned independently.

3.6.1 Strand Level-of-Detail. When reducing the number of hair strands with LOD, we would like to avoid suddenly removing a large number of hair strands at once with a slight change in L, since that would make the detail transitions more noticeable.

Consider a hair mesh with F bundles, each with exactly N hair strands. If we simply calculate the number of hair strands to be generated using $\lceil LN \rceil$, all F bundles would reduce their hair strand count simultaneously, so the effective detail levels would differ by exactly F hair strands.

We reduce the likelihood of such abrupt transitions by adding a random increment $\delta_i \in [0, 1)$ to the number of hair strands N_i we compute for each bundle i. The resulting number of hair strands for bundle i is calculated as

$$N_i^{\text{LOD}} = \lceil L(N_i + \delta_i) \rceil \ . \tag{8}$$

Though this simple solution does not guarantee that all detail levels for a hair model would differ by exactly one hair strand, it greatly reduces the likelihood of detail levels that differ by a large number of hair strands. Equation 8 also ensures that each bundle is assigned at least one hair strand for $L > 0$.

When reducing hair strands with LOD, we must also compensate for the reduction in hair material. We accomplish this by increasing the thickness of hair strands by a factor of $\min(1/L, N_i)$. Notice that this increase in hair thickness is not synchronized with discrete reductions in hair count to minimize the amount of geometry difference between detail levels.

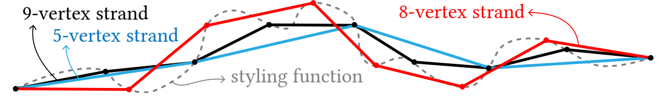


Figure 6: *Strands with different numbers of vertices generated from the same styling function. Notice that the difference between 9 and 8 vertices can be significant, even more so than 5 vertices.*

For computing L, it would be reasonable to use a measure of the hair bundle's thickness as h, such as the bundle's scalp face size (i.e. the radius of its bounding sphere or length of its longest edge) at rest shape, since bundles that appear thinner on screen can be approximated by rendering fewer hair strands.

3.6.2 Vertex Level-of-Detail. Though it is possible to reduce the hair vertex count by increments of 1, that would lead to two important problems. First, it becomes a challenge to maintain full SIMD occupancy with an arbitrary number of hair vertices. Second, levels may have distinctly different geometry, as illustrated in Figure 6, making it difficult to seamlessly transition between levels. This is because the styling functions can have strong high-frequency components along a hair strand, beyond what can be represented with the given number of hair vertices.

Our solution to avoid these issues ensures that the number of strand vertices is always a power of 2 plus 1. We begin with picking powers of 2 values for both the number of threads s and the number of vertices computed per thread n for the highest-resolution LOD. A lower-resolution level is formed by replacing either n or s with a smaller power of 2, $n^{\text{LOD}} = 2^{e_n}$ and $s^{\text{LOD}} = 2^{e_s}$, respectively. The desired level is formulated using

$$e = e_n + e_s = \max\left(0, \left\lceil \log_2(Lns) \right\rceil\right) \tag{9}$$

that results in $2^e + 1$ hair vertices. Any combination of $e_n \geq 0$ and $e_s \geq 0$ that satisfies $e = e_n + e_s$ can be used.

In practice, replacing n with n^{LOD} can be implemented easily. Replacing s with s^{LOD}, however, is not as trivial, since s is a compile-time constant, determining the number of threads to be used by the mesh shader. Therefore, it is advisable to use $s = s^{\text{LOD}}$, keeping e_s constant and only modifying e_n. If e_s must be modified, the mesh shader can be configured to generate multiple hair strands, instead of a single hair strand per execution, or different pre-compiled shader programs can be used for different values of e_s.

In addition, we must ensure that the LOD transition would be seamless when reducing the number of vertices. We achieve this by geometrically transitioning between detail levels.

Let $\lambda \in [0, 1]$ be a user-defined parameter defining the transition window between two detail levels. We define the next detail level using

$$\tilde{e} = \max\left(0, \left\lceil \log_2(Lns) - \lambda \right\rceil\right) \ . \tag{10}$$

When $e > \tilde{e}$, we transition between levels by morphing the higher-resolution strand shape towards the lower-resolution one. This is accomplished by moving each computed higher-resolution vertex position $\mathbf{p}_k$ with an odd index k toward the centers of its neighbors,

such that the updated vertex position $\mathbf{p}'_k$ is

$$\mathbf{p}'_k = (1-\gamma)\,\mathbf{p}_k + \gamma\,\frac{\mathbf{p}_{k-1}+\mathbf{p}_{k+1}}{2}\,, \tag{11}$$

where γ is the transition factor, calculated using

$$\gamma = 1 - \frac{\log_2(Lns) - \tilde{e}}{\lambda}\,. \tag{12}$$

This successfully morphs the strand shape between two levels, but we must also adjust the tangent directions to ensure that the change in the appearance of the shaded strand is also continuous. We achieve this by computing two sets of tangents: $\mathbf{t}_k$ using the original positions $\mathbf{p}_k$ (as in Section 3.5) and $\tilde{\mathbf{t}}_k$ that only uses the vertex positions with even k (the vertices that will remain in the lower-resolution level), such that $\tilde{\mathbf{t}}_k$ with even k is computed similarly and $\tilde{\mathbf{t}}_k$ with odd k is set as the average $(\tilde{\mathbf{t}}_{k-1}+\tilde{\mathbf{t}}_{k+1})/2$.

Then, we simply morph the tangents for *all* hair vertices to calculate the updated vertex tangents $\mathbf{t}'_k$ using $\mathbf{t}'_k = (1-\gamma)\,\mathbf{t}_k + \gamma\,\tilde{\mathbf{t}}_k$.

4 RESULTS

We tested our method on an NVIDIA GTX 4090 GPU. Figure 1 shows the performance of our approach with a scene containing 100 characters, each with unique hairstyles of 100 thousand strands. The hair meshes are simulated with extended position-based dynamics [Macklin et al. 2016], using the force models of Wu and Yuksel [2016] and sag-free initialization [Hsu et al. 2022]. The hair strands are generated and rasterized in only 2 ms with 8× MSAA (multisample antialiasing), excluding pixel shading time, using our method with LOD. Obviously, for such a large scene LOD plays an important role; without it, our method takes 47 ms to rasterize hair in this scene.

To provide a direct comparison to LHS, the state-of-the-art method for rendering strand-based hair models in practical applications, we generated the hair models in Figure 7 using our method and extracted the strand data, to ensure that both methods render identical hair models. Then, 1% of these hair strands are selected as guide hairs, and LHS weights are precomputed for all hair vertices. Finally, we rasterized the resulting models using LHS, which takes 18 ms (with or without MSAA). We experimented with different numbers of guide hairs (.1% to 10% of hairs) and did not observe a measurable performance difference. In comparison, our method can generate the identical models on the fly and rasterize them in 1.8 ms without LOD to produce an identical image (without MSAA), a performance improvement of 10×. Using 8× MSAA, our method's time goes up to 3.3 ms for these models. This cost increase with MSAA hints that our method could benefit from an optimized software rasterizer, instead of relying on the hardware rasterizer of the GPU, as we use in our tests. Note that we do not animate the guide hairs of LHS in this test, so the resulting rendered image is *identical* to ours.

The performance difference between LHS and our method can be explained by the amount of data used by the two methods. While LHS requires 340 MB for these models, our textures only take 39 KB. Note that we do not use LOD with either method in this comparison.

Figure 8 shows hairstyles rendered from different distances to the camera with and without our LOD. Notice that our LOD methods can produce a similar image to rendering without LOD while gradually reducing the rendered model complexity.

Figure 9 shows 200 Utah teapots, each with a separate hair mesh and hairstyle of 50 thousand strands rendered using our method without instancing. All hair in this scene rasterizes in 1.8 ms with 8× MSAA and 1.1 ms without MSAA using our method with LOD. Without LOD, hair models in this scene form 66 billion triangles, which rasterize in 150 ms using our method. All hair strands are generated using our 5 textures per teapot that fit in 3.2 MB for this entire scene, varying between 11 KB and 27 KB per teapot (based on hair mesh resolutions of 198 to 378 vertices).

Figure 10 shows different hairstyles generated using our method with different sets of styling parameters. Notice that a relatively low-resolution hair model is sufficient for generating a complex strand-based hair model with intricate details. These hairstyles are merely some examples of what can be produced with the styling functions we implemented. Note that various other procedural styling functions can be easily integrated into our system.

Almost all hair models in Figure 10 take 1 ms to rasterize. In our tests, disabling all styling computations provided only about 10% to 20% improvement in the total render time.

5 CONCLUSION

We have presented how hair meshes can be used for real-time strand-based hair rendering to achieve an unprecedented level of performance, accomplished via careful distribution of the workload in mesh shaders, offloading a part of the computation to texture filtering units, and LOD. We also describe how to attach continuous styling coordinates to the hair mesh, such that the styling perturbations remain consistent as the hair mesh deforms. Thus, our method can render strand-based hair for hundreds of characters at real-time frame rates on high-end GPU and makes strand-based hair rendering much more affordable for lower-end devices.

One major constraint of our approach is that the hair models must be defined using a hair mesh and a set of styling parameters. We cannot take arbitrary strand-based hair models and automatically convert them into our representation.

ACKNOWLEDGMENTS

We would like to thank Mitchell John Allen for the initial explorations and his help, Lee Perry-Smith for the character models, Dani Garcia for the hair mesh models in Figure 10d and f, and Alexander Tomchuk for the hair mesh and the character model in Figure 10e.

REFERENCES

Edwin Catmull and Raphael Rom. 1974. A Class of Local Interpolating Splines. In *Computer Aided Geometric Design*. Academic Press, 317–326. https://doi.org/10.1016/B978-0-12-079050-0.50020-5

Menglei Chai, Jian Ren, and Sergey Tulyakov. 2020. Neural Hair Rendering. In *Computer Vision – ECCV 2020*, Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm (Eds.). Springer International Publishing, Cham, 371–388.

Menglei Chai, Tianjia Shao, Hongzhi Wu, Yanlin Weng, and Kun Zhou. 2016. AutoHair: Fully Automatic Hair Modeling from a Single Image. *ACM Trans. Graph.* 35, 4, Article 116 (jul 2016), 12 pages. https://doi.org/10.1145/2897824.2925961

Menglei Chai, Changxi Zheng, and Kun Zhou. 2014. A Reduced Model for Interactive Hairs. *ACM Trans. Graph.* 33, 4, Article 124 (jul 2014), 11 pages. https://doi.org/10.1145/2601097.2601211

A. Daldegan, N. M. Thalmann, T. Kurihara, and D. Thalmann. 1993. An integrated system for modeling, animating and rendering hair. *Computer Graphics Forum* 12, 3, 211–221. https://doi.org/10.1111/1467-8659.1230211 MIRALab, Geneva Univ., Switzerland.

Epic Games. 2021. Unreal Engine. https://www.unrealengine.com

Figure 7: *Three hair models totaling 300 thousand hair strands, generated and rasterized without LOD in 1.8 ms without MSAA and 3.3 ms with 8× MSAA, using our method. The identical image takes 18 ms (with or without MSAA) using LHS with 3 thousand guide hairs in total. Our 5 textures for these three hair meshes fit in 39 KB in total, while the strand-based representation for LHS uses 340 MB for its hair strand data of almost 10 billion vertices.*

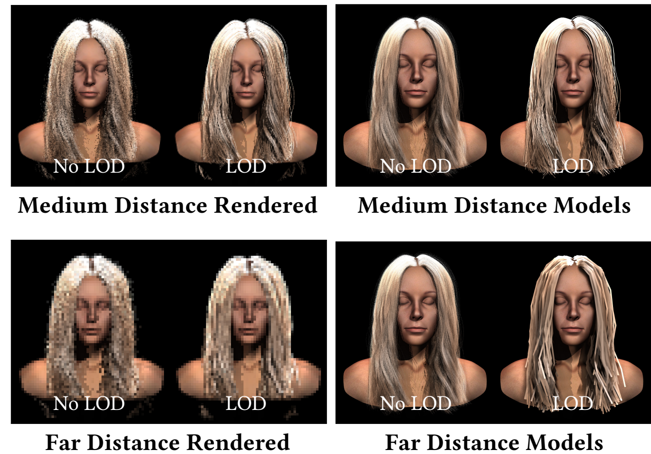


Figure 8: *Rendered images and the corresponding models from medium and far distances, showing that our LOD can produce a similar rendered image as rendering the model without LOD, while substantially simplifying the generated model geometry.*

Jerry Hsu, Nghia Truong, Cem Yuksel, and Kui Wu. 2022. A General Two-Stage Initialization for Sag-Free Deformable Simulations. *ACM Trans. Graph.* 41, 4, Article 64 (jul 2022), 13 pages. https://doi.org/10.1145/3528223.3530165

Jerry Hsu, Tongtong Wang, Zherong Pan, Xifeng Gao, Cem Yuksel, and Kui Wu. 2023. Sag-Free Initialization for Strand-Based Hybrid Hair Simulation. *ACM Trans. Graph.* 42, 4, Article 74 (jul 2023), 14 pages.

Songrun Liu, Zachary Ferguson, Alec Jacobson, and Yotam Gingold. 2017. Seamless: Seam Erasure and Seam-Aware Decoupling of Shape from Mesh Resolution. *ACM Trans. Graph.* 36, 6, Article 216 (nov 2017), 15 pages. https://doi.org/10.1145/3130800.3130897

Miles Macklin, Matthias Müller, and Nuttapong Chentanez. 2016. XPBD: position-based simulation of compliant constrained dynamics. In *Proceedings of the 9th International Conference on Motion in Games* (Burlingame, California) *(MIG '16)*. Association for Computing Machinery, New York, NY, USA, 49–54. https://doi.org/10.1145/2994258.2994272

Stephen R. Marschner, Henrik Wann Jensen, Mike Cammarano, Steve Worley, and Pat Hanrahan. 2003. Light Scattering from Human Hair Fibers. *ACM Trans. Graph.* 22, 3 (jul 2003), 780–791. https://doi.org/10.1145/882262.882345

Corentin Mercier, Thibault Lescoat, Pierre Roussillon, Tamy Boubekeur, and Jean-Marc Thiery. 2022. Moving Level-of-Detail Surfaces. *ACM Trans. Graph.* 41, 4, Article 130 (jul 2022), 10 pages. https://doi.org/10.1145/3528223.3530151

Zhong Ren, Kun Zhou, Tengfei Li, Wei Hua, and Baining Guo. 2010. Interactive Hair Rendering under Environment Lighting. In *ACM SIGGRAPH 2010 Papers* (Los Angeles, California) *(SIGGRAPH '10)*. Association for Computing Machinery, New York, NY, USA, Article 55, 8 pages. https://doi.org/10.1145/1833349.1778792

Radu Alexandru Rosu, Shunsuke Saito, Ziyan Wang, Chenglei Wu, Sven Behnke, and Giljoo Nam. 2022. Neural Strands: Learning Hair Geometry and Appearance from Multi-view Images. In *Computer Vision – ECCV 2022*, Shai Avidan, Gabriel Brostow, Moustapha Cissé, Giovanni Maria Farinella, and Tal Hassner (Eds.). Springer Nature Switzerland, Cham, 73–89.

Thorsten Scheuermann. 2004. Practical Real-Time Hair Rendering and Shading. In *ACM SIGGRAPH 2004 Sketches* (Los Angeles, California) *(SIGGRAPH '04)*. Association for Computing Machinery, New York, NY, USA, 147. https://doi.org/10.1145/1186223.1186408

Arunachalam Somasundaram. 2015. Dynamically Controlling Hair Interpolation. In *ACM SIGGRAPH 2015 Talks* (Los Angeles, California) *(SIGGRAPH '15)*. Association for Computing Machinery, New York, NY, USA, Article 36, 1 pages.

Sarah Tariq and Louis Bavoil. 2008. Real Time Hair Simulation and Rendering on the GPU. In *ACM SIGGRAPH 2008 Talks* (Los Angeles, California) *(SIGGRAPH '08)*. Association for Computing Machinery, New York, NY, USA, Article 37, 1 pages. https://doi.org/10.1145/1401032.1401080

Nghia Truong, Cem Yuksel, and Larry Seiler. 2020. Quadratic Approximation of Cubic Curves. *Proc. ACM Comput. Graph. Interact. Tech. (Proceedings of HPG 2020)* 3, 2, Article 16 (2020), 17 pages. https://doi.org/10.1145/3406178

Lvdi Wang, Yizhou Yu, Kun Zhou, and Baining Guo. 2009. Example-based hair geometry synthesis. In *ACM SIGGRAPH 2009 Papers* (New Orleans, Louisiana) *(SIGGRAPH '09)*. Association for Computing Machinery, New York, NY, USA, Article 56, 9 pages. https://doi.org/10.1145/1576246.1531362

Lingyu Wei, Liwen Hu, Vladimir Kim, Ersin Yumer, and Hao Li. 2018. Real-Time Hair Rendering Using Sequential Adversarial Networks. In *Computer Vision – ECCV 2018*, Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss (Eds.). Springer International Publishing, Cham, 105–122.

Kui Wu and Cem Yuksel. 2016. Real-Time Hair Mesh Simulation. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D 2016)* (Redmond, WA). ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/2856400.2856412

Kui Wu and Cem Yuksel. 2017a. Real-Time Cloth Rendering with Fiber-Level Detail. *IEEE Transactions on Visualization and Computer Graphics* PP, 99 (2017), 12 pages. https://doi.org/10.1109/TVCG.2017.2731949

Kui Wu and Cem Yuksel. 2017b. Real-Time Fiber-Level Cloth Rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D 2017)* (San Francisco, CA). ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3023368.3023372

Kun Xu, Li-Qian Ma, Bo Ren, Rui Wang, and Shi-Min Hu. 2011. Interactive Hair Rendering and Appearance Editing under Environment Lighting. *ACM Trans. Graph.* 30, 6 (dec 2011), 1–10. https://doi.org/10.1145/2070781.2024207

Ling-Qi Yan, Chi-Wei Tseng, Henrik Wann Jensen, and Ravi Ramamoorthi. 2015. Physically-Accurate Fur Reflectance: Modeling, Measurement and Rendering. *ACM Trans. Graph.* 34, 6, Article 185 (nov 2015), 13 pages. https://doi.org/10.1145/2816795.2818080

Xuan Yu, Jason C. Yang, Justin Hensley, Takahiro Harada, and Jingyi Yu. 2012. A Framework for Rendering Complex Scattering Effects on Hair *(I3D '12)*. Association for Computing Machinery, New York, NY, USA, 111–118. https://doi.org/10.1145/2159616.2159635

Cem Yuksel. 2015. Sample Elimination for Generating Poisson Disk Sample Sets. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2015)* 34, 2 (2015), 25–32. https://doi.org/10.1111/cgf.12538

Cem Yuksel and John Keyser. 2008. Deep Opacity Maps. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2008)* 27, 2 (2008), 675–680. https://doi.org/10.1111/j.1467-8659.2008.01165.x

Cem Yuksel, Scott Schaefer, and John Keyser. 2009a. Hair Meshes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2009)* 28, 5, Article 166 (2009), 7 pages. https://doi.org/10.1145/1661412.1618512

Cem Yuksel, Scott Schaefer, and John Keyser. 2009b. On the Parameterization of Catmull-Rom Curves. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling* (San Francisco, California). ACM, New York, NY, USA, 47–53.

Cem Yuksel and Sarah Tariq. 2010. Advanced Techniques in Real-Time Hair Rendering and Simulation. In *ACM SIGGRAPH 2010 Courses* (Los Angeles, California) *(SIGGRAPH 2010)*. ACM, New York, NY, USA, Article 1, 168 pages. https://doi.org/10.1145/1837101.1837102

Junqiu Zhu, Sizhe Zhao, Lu Wang, Yanning Xu, and Ling-Qi Yan. 2022. Practical Level-of-Detail Aggregation of Fur Appearance. *ACM Trans. Graph.* 41, 4, Article 47 (jul 2022), 17 pages. https://doi.org/10.1145/3528223.3530105

Arno Zinke, Cem Yuksel, Andreas Weber, and John Keyser. 2008. Dual Scattering Approximation for Fast Multiple Scattering in Hair. *ACM Trans. Graph.* 27, 3 (aug 2008), 1–10. https://doi.org/10.1145/1360612.1360631

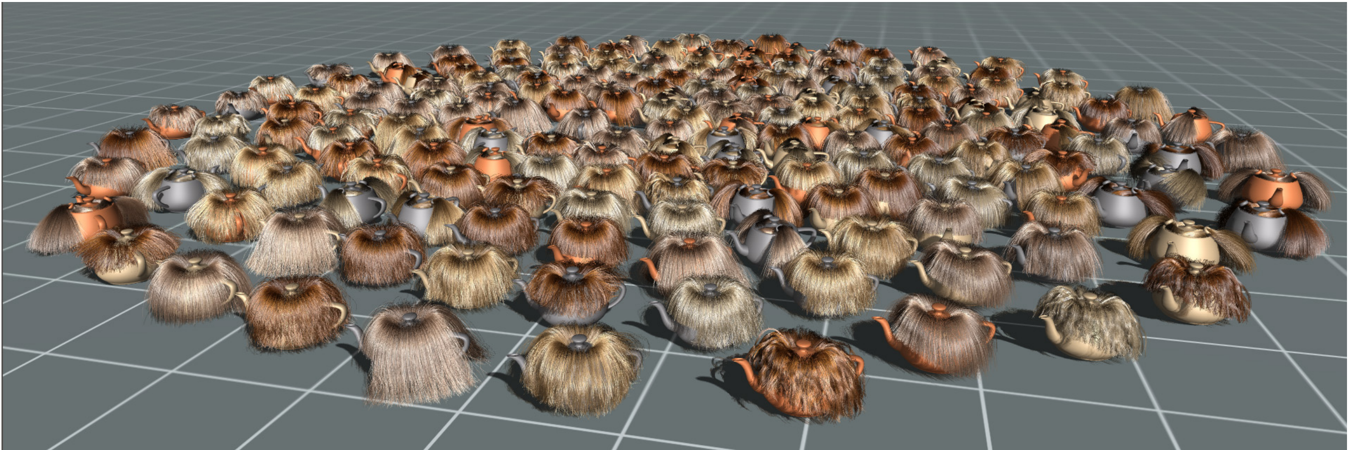


Figure 9: *200 Utah teapot models with unique hair models. Each hair model includes 50 thousand hair strands with a different set of styling parameters. The hair meshes are simulated to generate the hair animations. All hair in this scene is rasterized in 1.8 ms in total with 8x MSAA. Our 5 textures for 200 separate hair mesh models in this scene fit in 3.2 MB (17 KB per teapot on average).*

Figure 10: *Different hair meshes and hairstyles generated from them, rendered using our method without LOD and with* 8× *MSAA. All hair models have 100 thousand hair strands and take 1 ms to rasterize, except for (e), which takes 1.8 ms. The hair mesh resolutions and the storage costs of the 5 textures we use for representing each of them are (a-b) 185 vertices/13 KB, (c) 477 vertices/34 KB, (d) 7892 vertices/563 KB, (e) 1316 vertices/94 KB, and (f) 3236 vertices/231 KB.*