High-Performance Real-Time Implicit Strand-Based Hair Rendering via Software Rasterization

LUKAS LIPP, Meta Reality Labs Research, Switzerland and TU Wien, Austria
ADRIAN JARABO, Meta, Spain
MICHAEL WIMMER, TU Wien, Austria
LUKAS BODE, Meta Reality Labs Research, Switzerland

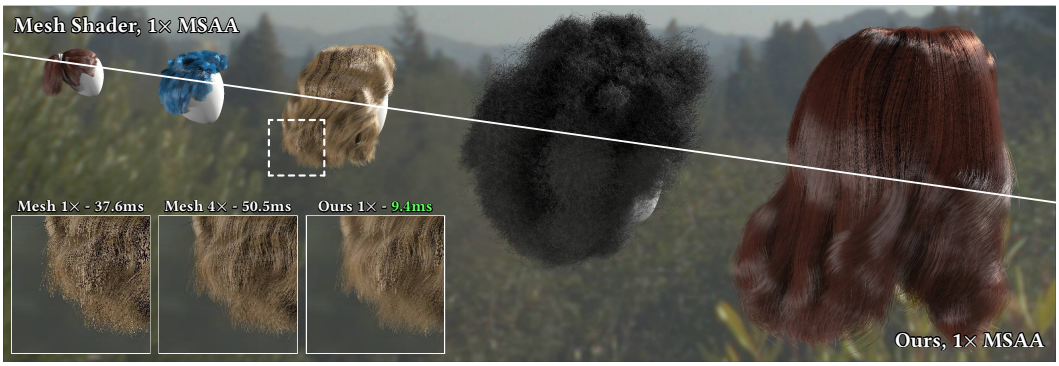
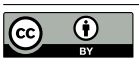


Fig. 1. Multiple grooms (total of 483k strands, 4K resolution) rendered at different distances to the camera using our deferred software rasterization pipeline with level-of-detail and baked ambient occlusion, compared against the reference mesh shader-based mplementation [Bhokare et al. 2025]. Our method is 4x faster than the mesh shader, with quality comparable to 4x MSAA. Per-style LOD strength (left to right) λ={4.0, 5.5, 2.0, 2.8, 3.0}.

In this work we propose an efficient deferred software rasterization pipeline for real-time rendering of strand-based hair using hair meshes. Hair plays a crucial role in creating expressive 3D characters, yet strand-based approaches are often restricted to high-end hardware and typically applied to only a small number of hero characters. Hair meshes have proven to be an efficient representation capable of handling a wide variety of groom styles, but existing mesh shader-based implementations still suffer from significant bottlenecks. In this work, we address these limitations with a software rasterization approach that improves performance and compatibility. Our method enables efficient far-field strand-based hair rendering—even at a single sample per pixel—by combining deferred shading with a strand filtering and reconstruction step, while requiring only minimal hardware support. To further enhance scalability, we introduce a level-of-detail (LOD) scheme that adapts hair representation and shading complexity based on viewing distance and screen-space coverage, reducing computational cost further while preserving visual fidelity. To the best of our knowledge, this is the first approach to achieve this combination of efficiency, flexibility, scalability, and broad hardware compatibility.

CCS Concepts: • **Computing methodologies → Computer Graphics**.

Authors' Contact Information: Lukas Lipp, Meta Reality Labs Research, Zurich, Switzerland and TU Wien, Vienna, Austria, lukas.lipp@cg.tuwien.ac.at; Adrian Jarabo, Meta, Zaragoza, Spain, ajarabo@meta.com; Michael Wimmer, TU Wien, Vienna, Austria, wimmer@cg.tuwien.ac.at; Lukas Bode, Meta Reality Labs Research, Zurich, Switzerland, lbode@meta.com.



ACM 2577-6193/2026/7-ART58
https://doi.org/10.1145/3820015

Additional Key Words and Phrases: Real-time rendering, Strand-based hair rendering

ACM Reference Format:
Lukas Lipp, Adrian Jarabo, Michael Wimmer, and Lukas Bode. 2026. High-Performance Real-Time Implicit Strand-Based Hair Rendering via Software Rasterization. *Proc. ACM Comput. Graph. Interact. Tech.* 9, 4, Article 58 (July 2026), 19 pages. https://doi.org/10.1145/3820015

1 Introduction

Rendering hair and fur is an important long-standing problem in computer graphics. Its intricate volumetric look, formed by hundreds of thousands of individual thin strands, makes it one of the most challenging appearances to render: It requires a large amount of small sub-pixel-scale semi-transparent primitives with a high-frequency scattering function. This results into a significant rendering computational cost due to memory bandwidth and overdraws, but also is prone to aliasing due to rendering thin primitives with glinty appearance. As a result, a common approach in real-time rendering is to severely simplify hair rendering, either by means of textured surfaces [Scheuermann 2004] or cards [Jiang 2016], and only recently explicit strand rendering has become feasible in real-time [Tafuri 2019; Taillandier and Valdes 2020]. However, these current strand-based approaches are usually restricted to very high-end hardware and for a limited number of hero characters, while on the other hand there is a growing demand for rendering large numbers of characters, each with unique and detailed hairstyles, efficiently across a wide range of hardware.

To partially alleviate some of these problems, Bhokare et al. [2025] proposed to represent hair strands implicitly using a coarse cage-like structure of the groom (the *hair mesh*) and styling operators, and only generate them on-the-fly at rendering time. This significantly reduced bandwidth requirements, becoming significantly more efficient than the commonly used linear hair skinning (LHS) [Somasundaram 2015]. However, this only addressed part of the problems: A significant number of overdraws were still required, and aliasing was still an issue, which degraded the rendering quality. Moreover, Bhokare's work [Bhokare et al. 2025] uses a mesh shader-based implementation, which is not supported by all existing hardware.

In this work, we introduce a new solution for efficient real-time hair rendering, which builds upon the idea of Bhokare et al. [2025], but implements it as a highly efficient deferred software rasterizer. This has several benefits, including better support across platforms, improved efficiency on the strand generation by carefully designing the GPU warp synchronization, and a significant reduction of the shading computational cost by shading only the visible hair strands via a deferred scheme, while at the same time reducing aliasing by post-filtering from strand-pixel intersection data. We accelerate this deferred pipeline by using a level-of-detail scheme that reduces both the number of strands and their complexity without visual penalty. Finally, we leverage the hair mesh structure as a proxy for occlusion, allowing us to use complex image-based lighting with occlusions, significantly increasing the type of lighting that can be used to shade explicit strands in real time. All of these contributions combined are a step forward for efficient hair rendering in real-time, allowing efficient rendering and shading of hair grooms with complex lighting and reduced visual artifacts (Fig. 1). In particular, our contributions are:

- A fast indirect dispatch based LOD-enabled hair software rasterization pipeline, with parallel construction of styled strands via SIMD-wide register communication and deferred shading using an atomic-enabled compressed G-buffer;
- a statistically driven depth-map visibility correction method for on-the-fly level-of-detail adaptation of groom complexity;

- a single-pass strand connectivity reconstruction and filtering method, enabled by conservative rasterization, that produces smooth, anti-aliased transparent strands even in our single-sample pipeline;
- and an ambient occlusion term directly encoded on the hair mesh that allows volumetric complex probe-based lighting in grooms.

2 Related Work & Background

Hair rendering. Early methods for rendering of hair in real-time leveraged textured hair surfaces [Scheuermann 2004] or cards [Koh and Huang 2001]. With the improvement of GPUs, rasterizing explicit strands have become feasible, with two main problems: On one hand, the complexity of grooms — with over 100K splines per groom — makes it unpractical to store and transfer the explicit strands. On the other hand, the amount of overdraw of semi-transparent strands results in a very computationally expensive, and aliasing-prone, rendering and shading phase. To avoid the former problem, the common approach is to only store, edit, simulate, and transfer a set of representative *guide hairs*, which are interpolated by each individual strand via *linear hair skinning* (LHS) [Somasundaram 2015], optionally applying some procedural styling operator [Chang et al. 2025]. This, however, requires storing the interpolation weights per strand, making it less practical in terms of memory footprint and bandwidth. Other more recent approaches have leveraged neural networks for the strand generation [Rosu et al. 2022], but these methods are still far from practical for real-time rendering. A more compact approach was proposed by Bhokare et al. [2025], which used the *hair mesh* structure [Yuksel et al. 2009] as a proxy for representing the whole groom. The hair mesh uses a proxy geometry to model the overall shape of a groom, using connected meshes to define hair bundles. Then, in render time, explicit strands are generated in a mesh shader, by interpolating within each individual bundle: In essence, this is very similar to LHS, where the edges of the bundle act as guide hairs, with the key difference that the interpolation weights are implicitly defined by the position of the strand in the bundle, and thus no weight needs to be stored, which significantly reduces the storage and bandwidth requirements. We build on Bhokare's approach, but proposing a highly-efficient method based on a deferred software rasterization method, while at the same time increasing its representative power and adding extended support for level of detail.

Software rasterization. With the advent of fully-programmable GPU pipelines, software rasterization has been demonstrated to be a viable pipeline [Karis et al. 2021; Laine and Karras 2011], even superior to hardware rasterization in certain scenarios with semitransparent objects or subpixel features, such as point clouds [Schütz et al. 2022] or semi-transparent Gaussians [Kerbl et al. 2023]. In the context of hair, Taillandier and Valdes [2020] proposed using software rasterization for efficient hair rendering in AAA games, by leveraging tiled per-pixel depth-bucket-based order-independent transparency (OIT) [Everitt 2001; Thibieroz 2018] and analytic line anti-aliasing. Later work by Ishihara and Mishima [2023] reduced memory consumption by implementing OIT using only three layers for hair rendering. Hair software rasterization is nowadays a standard in high-end AAA game engines [Cifariello Ciardi et al. 2022; Epic Games 2021; Kulikov 2025]. We describe an efficient single-bucket deferred software rasterization pipeline for hair, followed by a filtering pipeline to account for semi-transparency and aliasing.

Hair level-of-detail. An industry-standard for hair level-of-detail is to switch strand representations to cards, though it is usually a computationallyan expensive, manual process, with the notable exception of the work of Zheng et al. [2025], which automatized the process using differentiable rendering. Stochastic simplification [Cook et al. 2007] has been successfully applied to hair simplification by reducing the number of strands and thickening the remaining ones to account for the lost projected area, as well as reducing the complexity of the strands themselves by removing

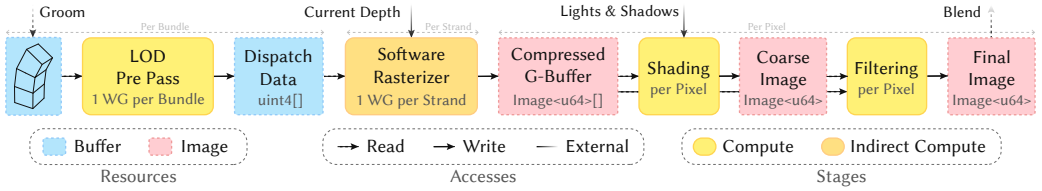


Fig. 2. Our software rasterization pipeline consumes a groom represented as a simple mesh composed of multiple bundles. A level-of-detail (LOD) pre-pass first prepares an indirect dispatch buffer, determining how many strands and control points will be rendered. During the rasterization stage, the pipeline reads the existing depth buffer and updates strand data in a `uint64` composite buffer using `atomicMin` operations. This data is subsequently used for per-pixel shading, with an optional filtering stage applied to further refine the result. The final output is then blended into the main render target. WG = workgroup.

control points [Bhokare et al. 2025]. Zhu et al. [2023, 2022] clustered groups of strands accounting for the aggregated scattering and internal scattering in the cluster. A similar approach was used by Montazeri et al. [2021, 2020] in the context of cloth rendering. We built upon the stochastic simplification approach for level of detail, but introducing simple yet efficient approaches to prevent for aliasing and other artifacts due to these simplifications.

Anti-Aliasing. Due to their extremely small screen-space footprint, individual hair strands suffer from severe under-sampling and noisy rendering. Furthermore, the thin, intricate, and highly detailed sub-pixel structure of hair presents a unique spatial challenge that general anti-aliasing and modern temporal upscaling methods, such as AMD FidelityFX Super Resolution [2021] or NVIDIA DLSS 4 [2025], struggle to resolve accurately without introducing blurring or ghosting artifacts. While some methods rely on neural-networks [Currius et al. 2022], recent work by Huang et al. [2025] demonstrates that a non-neural, explicit filtering framework can offer a superior alternative. By implementing explicit strand linking and directional filtering, their algorithmic approach preserves sub-pixel continuity and fine fiber details, like flyaway strands, without the blurring typical of black-box networks.

Background: Hair mesh rendering. The key idea of Bhokare et al. [2025] is to utilize the implicit definition of strands using hair meshes [Yuksel et al. 2009]. Hair meshes define the groom using a set of geometric bundles (see Fig. 3, left) that define the coarse structure of the groom. Each bundle is formed by a set of L layers, with the first and last layers placed at the hair roots and tips respectively. Each layer i is defined as a polygonal shape (in our case, we use quads) with vertices $\mathbf{v}^i$ and associated per-vertex tangents $\mathbf{t}^i$ and uvw coordinates.

During rendering, each strand is assembled as a spline with N control points uniformly spaced along the length of the strand $w \in [0, 1]$, with points at $w = 0$ and $w = 1$ placed the root and tip layers, respectively. Each strand origin is sampled using its uv coordinates which remain fixed along the strand, and its base position is computed by using spline interpolation from the bilinear interpolation of vertices and tangents in the bundle layers. Then, the final strand is obtained by applying the styling function as a function of each control point's uvw coordinates.

3 Our Method

Our deferred software rasterization, designed for grooms encoded as hair meshes, is described in Fig. 2. In its core, the software rasterizer takes the input hair mesh and a strand ID (defined, e.g., by its UV coordinate) and generates the strand on the fly, which is rasterized into a compressed G-Buffer (Section 3.1). The geometric information in the G-Buffer is later used for shading (Section 3.4).

To maximize efficiency and image quality, during the strand generation phase before rasterizing, we apply a per-bundle level-of-detail that reduces both the number of strands and their complexity (Section 3.2). Finally, to avoid visual artifacts due to undersampling and lack of transparency information in the G-Buffer, we apply a filtering phase to reconstruct the final image (Section 3.3), which is depth-composited with the rest of the scene.

3.1 Deferred Software Rasterization

To achieve high frame rates for distant hair rendering, we extend the hair mesh approach proposed by Bhokare et al. [2025] with a custom software rasterizer. Software rasterization has proven particularly effective in scenarios with extensive subpixel geometry as well as a huge amount of overdraw as shown by Schütz et al. [2022]. Hair strands are geometrically thin and often cover only a single pixel, which incurs significant computational overhead in traditional rasterization pipelines due to the mandatory 2×2 fragment shader quad dispatches — wasting up to 75% of shader invocations on helper lanes. Additionally, assembling a strand requires each segment to access data from its neighboring control points; done naively, every thread would redundantly re-read and re-compute all prior vertices. Alternatively, using shared memory to store intermediate data is a better approach but still uses up unnecessary storage and bandwidth.

To address this, our implementation leverages a workgroup-wide, register-only cooperative strand assembly to maximize efficiency and minimize memory traffic. To dynamically adapt hair complexity, we perform a prepass that computes fine-grained resolution data (details in Section 3.2) consumed by the subsequent indirect dispatch stage for spawning the correct amount of workgroups (WG). The screen-space strand data gets compressed and stored in a per-pixel 64-bits wide G-Buffer which is then used for deferred shading.

3.1.1 Cooperative Strand Assembly. Every WG, consisting of subgroup (SG) size number of threads, cooperatively assembles one strand and writes the end result to the target image via atomic min operations. While Bhokare et al. [2025] describe the strand assembly at a high level, they do not provide concrete implementation details on how to parallelize it efficiently. Since assembling each strand's segment requires information from previous and next segments, parallelization is not trivial. Done naively, every thread would redundantly recompute the data dependencies surrounding the current control point multiple times to get the final styled strand segment. Instead, we write the bundle's layer data into shared memory while using fast subgroup communication to share computed control points between neighboring threads. In this way, each control point needs to be computed only once (see Fig. 3).

The first pass assembles a base spline where each control point has a tangent calculated from its previous and next vertices via finite differences. This tangent is then used together with the position to calculate the Hermite spline between two control points. Once the base spline is built, we further subdivide it based on the desired resolution, which gives a list of final vertex positions. Each position is then used as the input for the styling function $f(x)$ resulting in the final styled strand. Note that for the styled strand we no longer use a spline, and instead render a polyline. For the styling we need to calculate new tangents for shading; otherwise we run into shading artifacts as mentioned in the original work [Bhokare et al. 2025]. Each tangent is again calculated from the previous and next control points. This two-pass dependency for calculating the final tangents makes it non-trivial to parallelize; once again, our approach uses sub-pass communication to share vertex and tangent data between neighboring threads (see code 1).

3.1.2 Compressed G-Buffer for Deferred Atomic Shading. Once each styled segment is computed, we rasterize them using the digital differential analyzer (DDA) line-drawing algorithm. Because hair strands are extremely thin, we rasterize them as single-pixel-wide lines. We experimented

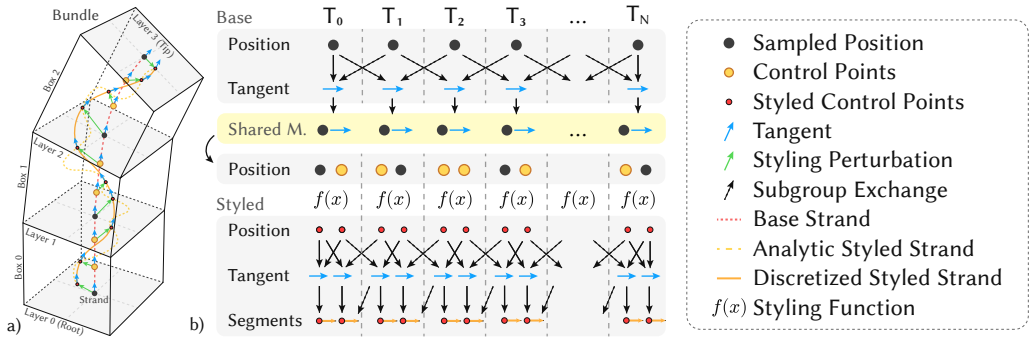


Fig. 3. In order to efficiently parallelize the strand assembly each workgroup (WG) first cooperatively loads all layer data into shared memory. Each thread shares the first and last calculated positions of its strip with the previous and next thread to circumvent recomputation. We do this process twice: First when assembling the layer data for the base spline (position), and second when assembling the final styled strand (position and tangent).

with rendering thick strands; however, we found that this provided no visual benefit in the far field case while requiring much more compute.

To support our reconstruction filter (see Section 3.3), the rasterization stage writes to two separate G-buffer targets (or array slices): a *center-sample* G-buffer that is only updated if the strand passes within a configurable diameter through the pixel center, and a *conservative* G-buffer that is updated for any pixel the strand intersects.

Shading each strand during this line-rasterization is prohibitively computationally expensive due to overdraws: We evaluated a per-fragment (similar to a fragment shader) and a per-vertex shading approach, and both proved to be too slow in practice. Consequently, we adopted a deferred shading strategy.

For deferred shading, we need the following per-pixel data for shading (see details later in Section 3.4):

- **Position** (`float3`)
- **Tangent** (`float3`)
- **Shading parameters**: albedo (`float3`), roughness (`float`), tilt (`float`)
- **Ambient occlusion** (`float`)

Additionally, depth values are required in the high bits to allow for an atomic min operation.

However, for deferred shading to be viable, all necessary shading data must fit within the framebuffer, and be written simultaneously to avoid race conditions, into a 64-bit per-pixel payload. This constraint is crucial, as recomputing the necessary shading data is computationally expensive: It would require either reconstructing the full strand geometry or storing indices for each bundle, strand, and segment, which quickly exceeds the available bits. To address this, we compress the data needed for shading as follows:

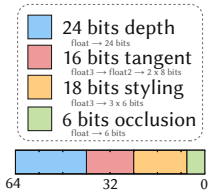
(1) **Reduced depth precision:** 24 bits are used for depth.
(2) **Position reconstruction:** Instead of storing the position (float3), we use the common approach in deferred engines and reconstruct it from the depth value. This approach restricts us to a specific sample location within the pixel, as the position can only be accurately recovered at that point [Mittring 2007].
(3) **Shading Parameters:** Instead of storing the shading parameters in the G-Buffer, we store the uvw styling coordinates, which we use for querying these parameters at shading time.

This allows us to query these shading parameters only once per pixel, and store the whole appearance model in three floats.

(4) **Tangent compression:** We encode the tangent vector using an octahedral unit vector encoding, reducing it from float3 to float2.

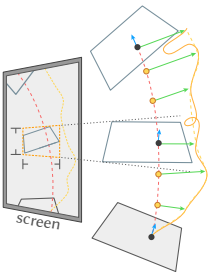
(5) **Data quantization:** All float values except for depth are quantized as either uint8 or uint6 to further reduce memory usage (see inset).

By applying this compression, we can fit all required data into a 64-bit framebuffer, enabling efficient deferred shading. Note that as with common deferred rendering, we do not handle transparency: This unfortunately is very important for hair, since strands are usually significantly smaller than the pixel footprint, and thus multiple strands might contribute to the pixel footprint, and transparency is very important for compositing with the rest of the scene. We solve this issue as a post-process filter, as we discuss later in Section 3.3.



3.2 Level-of-detail

For far-field rendering we want to reduce the workload by limiting the number of line primitives being rendered, which can be achieved by both reducing the number of strands and the strand resolution. We leverage the on-the-fly nature of our strand generation to adaptively generate strands based on its importance on screen. First, we run a pre-pass per bundle, where we compute the number of strands per bundle and the accuracy of each strand, storying it to a device-only buffer. Each thread of this pass processes a single bundle and writes its corresponding data.



This data is then used as the source for the strand assembly pass in the software rasterizer as the source for an indirect dispatch call, dispatching a variable amount of work-groups per bundle. The level-of-detail (LOD) is calculated based on the screen-space footprint [Unterguggenberger et al. 2024, 2025] of a layer (see inlay) and the sensitivity can be controlled via a parameter λ, which can be different for different hairstyles or strand counts.

3.2.1 Dynamic Strand Counts. In order to reduce the complexity of the groom we use an approach inspired by stochastic simplification methods [Cook et al. 2007]. Essentially, we randomly remove strands in far away bundles based on their projected screen-space footprint, so that the number of strands per bundle N_{LOD} is

$$N_{\text{LOD}} = \text{clamp}\big(\lceil L \cdot (N + \delta) \rceil,\ 1,\ N\big), \tag{1}$$

where N is the full strand count per bundle, $\delta \sim \mathcal{U}[0, 1)$ is a per-bundle random offset that smooths LOD transitions, and $L \in [0, 1]$ is the LOD selection value computed as

$$L = \text{clamp}\left(\frac{\| \text{AABB}_{\text{max}} - \text{AABB}_{\text{min}} \|}{R_y} \cdot \lambda,\ 0,\ 1\right), \tag{2}$$

where AABB is the screen-space bounding box of the bundle, which is normalized by R_y the vertical resolution of the screen, and λ is the LOD sensitivity parameter. The projected bounding box is computed by projecting each layer's vertices to screen space, and accumulating a single axis-aligned bounding box (AABB) as the component-wise maximum over all per-layer AABBs:

$$\text{AABB}_{\text{min}} = \min_i \min\big(\mathbf{s}^i_x,\ \mathbf{s}^i_y,\ \mathbf{s}^i_z,\ \mathbf{s}^i_w\big), \quad \text{AABB}_{\text{max}} = \max_i \max\big(\mathbf{s}^i_x,\ \mathbf{s}^i_y,\ \mathbf{s}^i_z,\ \mathbf{s}^i_w\big), \tag{3}$$

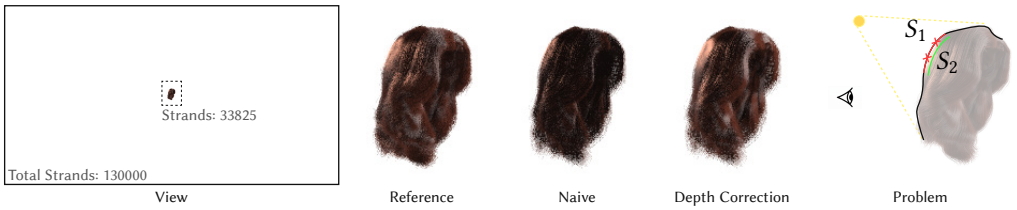


Fig. 4. **Shadow artifacts due to LOD-based strand prunning.** Applying stochastic strand removal results in inconsistent shadow or deep opacity maps (DOM), which might produce shadow artifacts when applying LOD. When the depth map is calculated using the full geometry set, but the front-most strand (S_1) is randomly culled, the shadow volume becomes "stale". Consequently, the remaining strand (S_2) incorrectly samples the high-opacity region deeper inside the groom (*Problem*), leading to unnatural darkening and visual popping (*Naive*). Pushing all strands closer based on the ratio of removed strands fixes this (*Depth Correction*).

where $\mathbf{s}^i_{x,y,z,w}$ are the screen-space positions of the four quad vertices of layer i. Bundles with AABBs laying entirely outside the viewport are frustum-culled ($N_{\text{LOD}} = 0$). The resulting value is written to the instance count of an indirect dispatch buffer with B entries, and computed via a dispatch of B threads, with one thread assigned to each bundle.

3.2.2 Dynamic Strand Resolution. A second form of LOD we apply is when selecting the number of control points for each strand geometry, which inherently reduces the number of rasterized segments. We use the same LOD selection L value as before (Eq. (2)), and — following Bhokare's approach [Bhokare et al. 2025, Eq.9] — compute the number of control points per bundle as

$$C_{\text{raw}} = \max\left(\lfloor \sqrt{L} \cdot C_{\max} \rfloor,\ C_{\text{layers}}\right), \tag{4}$$

where $C_{\max} = 127$ is the maximum number of control points and C_{layers} is the number of bundle layers (a hard lower bound). We use $\sqrt{L}$ instead of L to better preserve curvature at intermediate distances. To ensure smooth vertex LOD transitions, we snap C_{raw} to a power-of-two-plus-one value, giving a final control points count value of

$$C = \min\left(2^{\lfloor \log_2(C_{\text{raw}}-1) \rfloor} + 1,\ C_{\max}\right), \qquad C_{\text{raw}} \geq 3. \tag{5}$$

This computation is also done with one thread per bundle, in the same pass as the strand count computation. The final control point count is stored in a GPU buffer next to the (indirect) dispatch count (fitting into a single `uint4` per bundle), which is consumed by the software rasterizer. Combined with the per-bundle random offset δ in Eq. (1), which staggers strand-count transitions across bundles, this snapping rule keeps geometric LOD switches sub-pixel under typical viewing speeds; we did not observe perceptible popping in our test sequences.

3.2.3 Depth Comparison Artifact Fix. An important problem of removing strands is that previously computed depth maps become invalid (recomputing them is computationally expensive, which we want to avoid). This is especially noticeable with shadow or opacity maps (Fig. 4, *Problem*), where stochastically removing a strand at the boundary of the groom might lead to inner strands becoming visible, but still fall behind the stored depth value. In the case of shadow maps, this results in a dark region (Fig. 4, *Naive*). We fix this issue by moving the inner strand closer to the camera during depth testing by a small offset Δ, which depends on the distance to the camera and the proportion of remaining strands.

To derive this offset Δ we make use of the Beer-Lambert law and statistically account for aggregated visibility as

$$V(d) = \exp(-\sigma \cdot d), \tag{6}$$

with σ proportional to strand density. When stochastically removing strands, only a fraction $\beta = \frac{N_{\text{remaining}}}{N_{\text{total}}}$ of the strands remain, so that density becomes $\sigma \cdot \beta$ and visibility becomes

$$V'(d) = \exp(-(\sigma \cdot \beta) \cdot d) = \exp(-\sigma \cdot (\beta \cdot d)). \tag{7}$$

This shows that reducing the number of strands is equivalent to traversing a reduced effective depth $d_{\text{eff}} = rd$. To express this compensation as an additive shift in depth space, we invert the exponential, yielding an offset

$$\Delta = -\log(\beta) = -\log\left(\frac{N_{\text{remaining}}}{N_{\text{total}}}\right). \tag{8}$$

This shift is $\Delta = 0$ when no culling occurs ($\beta = 1$) and increases smoothly as more strands are removed ($\beta \downarrow 0$).

3.3 Anti-Aliasing and Opacity Reconstruction

In order to achieve low frame render times, the number of pixels available to write via atomic operations is very limited (only 64 bits per pixel, with 24 of those being used for depth), which limits us to a single sample per pixel in high-resolution renders. This results in significant aliasing and disconnected strands, since only strands passing through the pixel's center sample point are taken into account (see Fig. 5, *MSAA 1*). Using conservative rasterization on the other hand results in thick lines, specially for isolated strands (see Fig. 5 *Conservative Rasterization*).

We use an approach inspired by the work of Huang et al. [2025]. However, they tackle these problems using a multi-pass approach that includes an explicit strand-connection step relying on per-strand indices which we cannot provide given our tight framebuffer. Instead, our solution stores two layers per pixel: A *center-sample* layer that records only the strands passing through the pixel center, and a *conservative* layer which records the atomic-min G-buffer value over all strands intersecting the pixel.. Combined, these layers provide the filter with sufficient context about strand connectivity to reconnect disconnected strands, eliminating the need for the explicit connection pass proposed by Huang et al. [2025]. We also extend this approach by supporting multisampling for the software rasterizer, recording samples into multiple layers of a `uint64` image and resolving them later (see Fig. 5, *MSAA 8*). However, we only use this approach as a reference, since it is not practical in terms of memory and computational cost.

3.3.1 Elliptical Bilateral Filter. We denoise the center-sample frame buffer with an orientation-aware bilateral filter similar to Huang et al. [2025]. We apply the filter over a square window of radius r=5 (i.e., a $(2r+1)\times(2r+1)$ kernel). For every neighbor pixel $\mathbf{Q}$ of center pixel $\mathbf{P}$ at offset $\mathbf{d}$ the displacement is decomposed into tangential and perpendicular components using the screen-space tangent $\mathbf{T}_P$:

$$\mathbf{d} = \mathbf{Q} - \mathbf{P}, \qquad d_{\parallel} = \mathbf{d} \cdot \mathbf{T}_P, \qquad d_{\perp} = \left\| \mathbf{d} - d_{\parallel}\, \mathbf{T}_P \right\|. \tag{9}$$

The bilateral weight [Huang et al. 2025, Eq. 11] is the product of an elliptical spatial term and a color-similarity term, following

$$w_{PQ} = \underbrace{\exp\left(-\frac{d_{\parallel}^2}{\sigma_{\parallel}^2} - \frac{d_{\perp}^2}{\sigma_{\perp}^2}\right)}_{w_{\text{spatial}}} \cdot \underbrace{\exp\left(-\frac{\|\mathbf{C}_P - \mathbf{C}_Q\|^2}{\sigma_c^2}\right)}_{w_{\text{color}}}, \tag{10}$$

where $\sigma_{\parallel} = r \cdot s_{\parallel}$ and $\sigma_{\perp} = r \cdot s_{\perp}$ are configurable major- and minor-axis standard deviations, and σ_c the color-similarity standard deviation (which we set to $\sigma_c = 0.9$). Neighbors whose depth

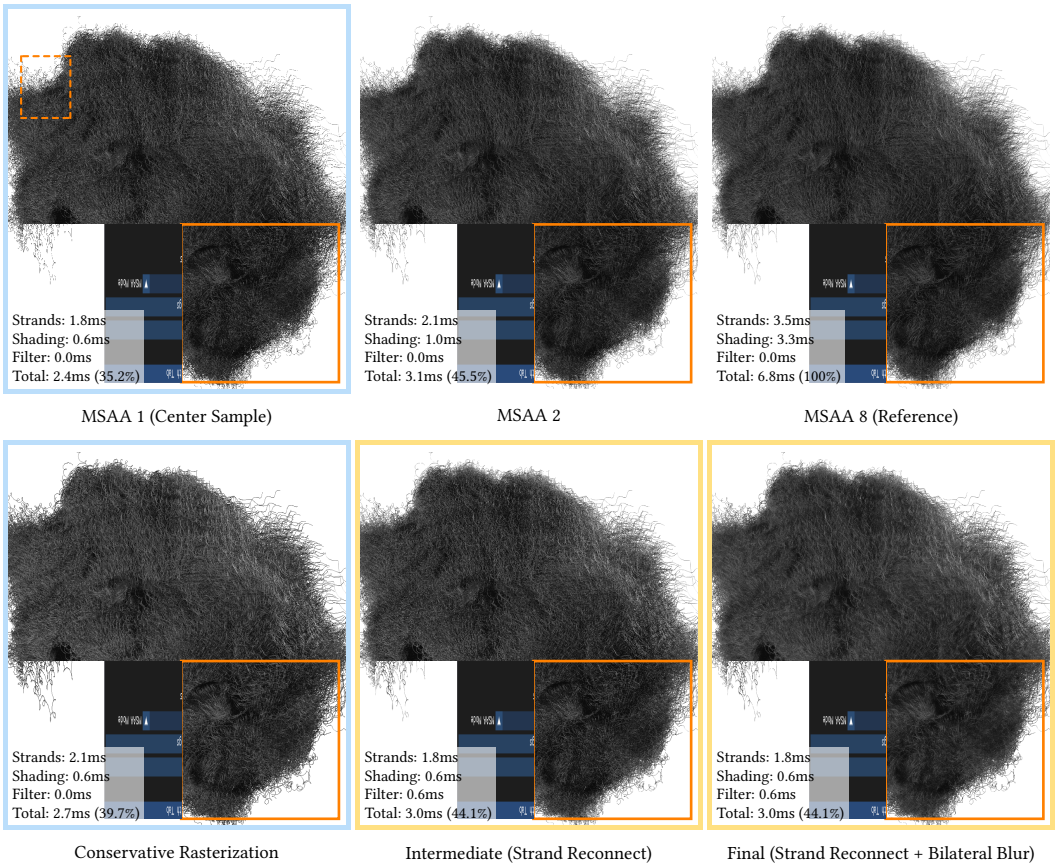
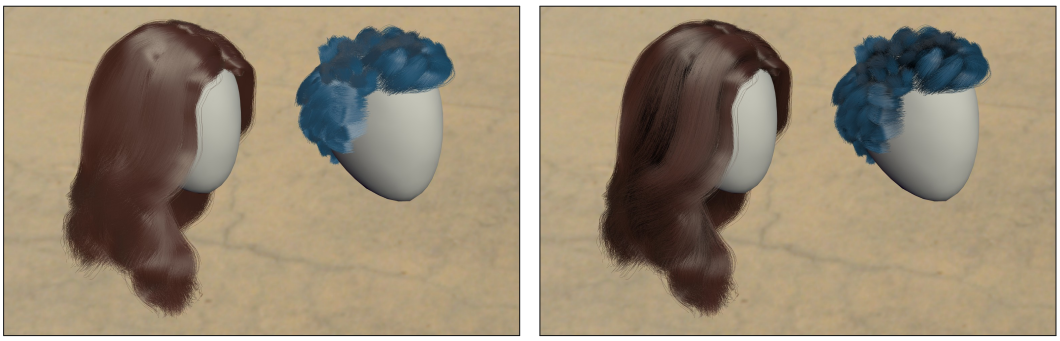


Fig. 5. The filter takes two inputs: a center-hit render at MSAA 1 and a conservative render, which captures all strand-touched pixels regardless of center hits and serves purely as spatial guidance. These feed into a two-stage pipeline: a reconnection pass that promotes isolated strand pixels by borrowing shading from nearby center-hit neighbors, followed by a directional blur that smooths along the strand tangent. Both stages are executed within a single 7×7 kernel pass. The final result closely matches the quality of high-sample references at a fraction of the cost.

differs from P by more than a threshold of 1.45×10^{-3} are rejected before weighting. The filtered color $\hat{C}_P$ is thus the result of the convolution of the kernel following $\hat{C}_P = \frac{\sum_Q C_Q \cdot w_{PQ}}{\sum_Q w_{PQ}}$.

3.3.2 Strand Connection. Pixels that appear in the conservative frame buffer but have a missing sample in the center-sample frame buffer are *promoted*: They borrow shading from nearby center-hit neighbors. Because the conservative rasterization layer already captures every strand that touches a pixel, this promotion mechanism implicitly reconnects visually disconnected strand segments without the explicit connection pass of Huang et al. [2025]. When a closer strand exists in the conservative rasterization layer in front of the center-hit strand (depth mismatch $\geq 5 \times 10^{-4}$), the reconstruction is blended on top of the existing center-hit frame-buffer. For promoted pixels the color-similarity term is disabled ($w_{\text{color}} = 1$) because the pixel has no shaded color to compare against.



(a) Without Ambient Occlusion (b) With Ambient Occlusion

Fig. 6. Comparison of the effect of pre-baked ambient occlusion. Both images are rendered using our software rasterizer with LOD and reconstruction filter (SWR 1+F+L) without (a) and with (b) baked ambient occlusion. The ambient occlusion term adds depth and volume to the groom by darkening occluded regions near the scalp and within dense strand clusters.

3.3.3 Alpha Estimation. Per-pixel opacity is derived from the ratio of hits in the center-sample frame buffer versus hits in the conservative frame buffer along a tangent-aligned strip of adjustable width w_{strip} (we choose 0.5 px), whose length is bounded by the bilateral filter's kernel diameter. Within this strip we count center hits H_{center} and conservative hits pixels H_{cons} and estimate the transparency of the pixel $\mathbf{P}$ as $\alpha_P = \frac{H_{\text{center}}}{H_{\text{cons}}}$. For center-sample hit pixels with a conservative layer depth mismatch the output opacity is 1 as blending already happened (important for depth separated flyaway strands in front of groom); otherwise it equals α_P (see code 2 for a complexity-reduced version).

3.4 Shading Model

As described in Section 3.1, we defer shading to a second step that operates in screen space over the G-Buffer with the information stored in that buffer. We use a physics-based hair shading model similar to the one proposed by Pekelis et al. [2015], but fitted to Chiang's [Chiang et al. 2016] model instead of Marschner's. We obtain the parameters of the model on a per-pixel level, by querying the shading parameters using the uvw-coordinates stored in the G-Buffer.

Point Light Sources . We render point light sources by evaluating the shading model directly, plus a Lambertian term similar to Kajiya-Kay's [1989] to simulate the contribution of multiple scattering. Volumetric shadows are computed using deep-opacity maps [Yuksel and Keyser 2008], which we query using the depth-correction described in Section 3.2.

Probe Lighting and Baked Ambient Occlusion. For probe-based lighting, we use the standard approach of pre-filtered environment maps for specular components (the single scattering hair shading model), and spherical harmonics expansion for more diffuse contributions (the diffuse multiple scattering). For the specular components, we approximate the R and TRT lobes with a single GGX lobe, which we use for querying the pre-filtered probe. For the diffuse part, we simply use Ramamoorthi's [Ramamoorthi and Hanrahan 2002] irradiance expansion centered at a normal of the fiber oriented towards the view direction. This introduces a subtle view-dependent effect, but enables accounting for more directional incoming light than tangent-based projections [Mehta et al. 2012].

A key element for improving shading quality from probes is our baked occlusion approach. In the context of hair, screen-space approaches [Jiménez et al. 2016] might fail, since these assume surfaces and not volumetric-like assets. Instead, we leverage the hair mesh structure and pre-bake an ambient occlusion value in each vertex of the hair mesh proxy. This value is interpolated in the same way as the position or tangents during the strand assembly, and rasterized into the G-Buffer, which is very efficient. Figure 6 shows the effect of accounting for this pre-baked ambient occlusion term.

4 Results

We evaluate our method on an NVIDIA RTX 5080 (SG size = 32) at 1920×1080 resolution using a hair scene with 127k strands. Figure 7 and Table 1 report rendering quality and GPU frame times across three camera distances (close, mid, far) for all pipeline configurations for two different grooms. We compare our deferred software rasterizer against the mesh-shader-based approach from Bhokare et al. [2025], with and without our proposed LOD schemes, and with the baked ambient occlusion. In Fig. 1 we render the scene with a high-frequency environment map and three fill point light sources, while for Fig. 7 we only use three point light sources.

Quality. Hair rendering at low sample counts suffers from strong aliasing: Individual strands are thinner than a pixel and their sub-pixel coverage changes rapidly with camera distance, producing flickering and loss of fine detail. This is especially pronounced at MSAA 1, where Figure 7 reveals severe strand dropout and fragmentation across all distances. Our reconstruction filter (SWR 1+F) directly addresses this by aggregating coverage information across neighboring pixels, recovering strand detail comparable to MSAA 4–8 while maintaining performance close to MSAA 2. Adding LOD (SWR 1+F+L, highlighted in yellow) further reduces computational cost at distance without sacrificing this quality level, making it our recommended configuration. Depending on the hairstyle it can help to adjust LOD λ or the filters $\sigma_\perp$, $\sigma_\parallel$ or radius. Overall we found no cases where the filter did not work at all.

Performance. Table 1 shows that our software rasterizer outperforms the mesh shader baseline across all distances and MSAA levels. At MSAA 1 without LOD, the software rasterizer takes 3.1 ms vs. 13.6 ms for mesh shaders — a 4.4× speedup. The reconstruction filter adds moderate computational overhead, except at far distance where heavy atomic contention in the conservative layer causes a spike to 4.5 ms (shown in orange). Enabling LOD as we will discuss next or disabling the strand connection pass at far distance are possible solutions to address this issue. Computational cost is dominated by the strand generation and software rasterization phase, which takes up about 96% of the rendering computational cost even in the presence of complex shading, which has a very small computational cost thanks to the deferred approach. Similarly, the filtering computational cost is negligible.

LOD. LOD brings the largest absolute gains at far distance, where strand counts drop from 127k to 21k. For the software rasterizer with filter (our recommended configuration), the far frame time falls from 4.5 ms to 0.4 ms — an 11× reduction — while close and mid views performance improves by 12% and 28% respectively. In contrast, the mesh shader LOD variant is *slower* than its non-LOD counterpart at close and mid distances (shown in red). This counter-intuitive result stems from a fundamental limitation of the task shader model: a task shader can only launch mesh-shader workloads at the granularity of an entire workgroup. Our compute pre-pass, in contrast, evaluates this per thread — one thread per bundle — and feeds the result into an indirect dispatch, incurring nearly zero computational cost due to better parallelization, and LOD is strictly beneficial at every distance. The pre-pass for determining the LOD value adds a very small computational cost (3%

of the total frame), but significantly accelerates the strans generation and software rasterization phases.

Multiple Grooms. Our pipeline scales naturally to scenes containing multiple grooms. Each groom is processed sequentially through the same rasterization and filtering pipeline, with its result blended into the render target before the next groom is processed. This allows an arbitrary number of grooms to be composed without any modifications to the core pipeline. By preserving the depth buffer between groom passes, strands occluded by previously rendered grooms are rejected early, reducing atomic write contention and improving performance. Figure 1 demonstrates a scene with multiple grooms rendered this way.

Table 1. GPU frame time benchmark (ms) on an RTX 5080 at 1920×1080. The number in parentheses is the raw strand rasterization time. The MSAA column denotes samples per pixel. Red text reveals increased mesh shader frame times due to culling computational overhead. Orange indicates performance degradation from atomic contention in the filter layer. The yellow-highlighted row represents our best quality-to-performance ratio.

			Groom 1			Groom 2		
Rasterizer	**Filter**	**MSAA**	**Close**	**Mid**	**Far**	**Close**	**Mid**	**Far**
Software Rasterizer – Compute								
		1	3.08 (2.68)	2.35 (2.17)	1.81 (1.76)	3.40 (2.87)	2.74 (2.50)	2.40 (2.33)
	✓	1	3.72 (2.89)	2.70 (2.37)	4.50 (4.38)	4.23 (3.03)	3.00 (2.60)	4.23 (4.09)
		2	4.10 (3.37)	2.84 (2.56)	2.03 (1.94)	4.37 (3.41)	3.12 (2.76)	2.58 (2.47)
		4	5.97 (4.61)	3.68 (3.21)	2.37 (2.21)	6.24 (4.45)	3.96 (3.33)	2.87 (2.68)
		8	8.27 (5.67)	4.71 (3.77)	2.77 (2.44)	9.10 (5.61)	4.95 (3.85)	3.18 (2.84)
Software Rasterizer – Compute + LOD								
		1	2.73 (2.30)	1.69 (1.49)	0.25 (0.18)	3.07 (2.52)	2.24 (1.95)	0.35 (0.27)
	✓	1	3.35 (2.48)	1.94 (1.56)	0.36 (0.22)	3.79 (2.59)	2.40 (1.97)	0.45 (0.30)
		2	3.73 (2.99)	2.15 (1.81)	0.29 (0.19)	4.08 (3.08)	2.55 (2.17)	0.49 (0.34)
		4	5.54 (4.16)	2.87 (2.36)	0.42 (0.22)	5.95 (4.14)	3.27 (2.65)	0.52 (0.32)
		8	7.71 (5.08)	3.69 (2.78)	0.62 (0.25)	8.56 (5.06)	4.17 (3.03)	0.76 (0.36)
Hardware Rasterizer – Mesh Shader								
		1	13.58	10.58	9.66	15.07	11.24	10.55
		2	16.42	12.14	9.75	18.47	12.83	10.67
		4	19.80	14.15	10.14	22.36	14.92	11.12
		8	23.14	16.47	10.76	26.38	18.44	12.39
Hardware Rasterizer – Mesh Shader + LOD								
		1	19.45	13.31	2.28	20.47	15.10	3.02
		2	23.12	16.54	2.40	25.49	19.51	3.15
		4	26.18	18.51	2.68	28.60	21.19	3.57
		8	28.40	20.34	3.16	32.00	23.70	4.60

5 Discussion

Compatibility. Our solution relies only on widely supported features, making it applicable across a broad range of devices. In particular, it requires compute shader support with indirect dispatch functionality (available since Vulkan 1.0), as well as 64-bit integer atomic minimum operations. These atomic operations are supported natively from Vulkan 1.2 onward or can be enabled via extensions on earlier versions. For devices that do not support atomic operations on images, a buffer-based fallback can be used instead. Note that throughout this paper we solely use the term image for better distinction from non-framebuffer related resources.

Interactive Updates. Inherited from hair meshes, we can express a large variety of hairstyles with minimal storage requirements. Furthermore, both the hair mesh topology and the styling function

Fig. 7. Rendering quality comparison at close, mid, and far distance. SWR = software rasterizer, F = reconstruction filter, L = LOD, 1, 2, 4, 8 = number of MSAA samples. Zoom insets show center crop from the current view. λ=3.0 for Groom 1 and λ=7.0 for Groom 2.

can be efficiently modified on the fly, enabling interactive workflows, including animations and efficient physics simulations.

Multiview Rendering. Our approach is well suited for multiview rendering scenarios, including virtual reality. Strand assembly is performed only once, while projection is executed per view. Additionally, the deferred shading and filtering computational cost remains small relative to the rasterization computational cost when rendering to multiple targets, enabling stereo and multiview output with minimal computational overhead.

5.1 Limitations and Future Work

While the proposed software rasterizer delivers high performance for far-field rendering, it becomes less efficient for near-field rendering, particularly when strands project wider than a single pixel. A hybrid approach combining software rasterization for distant hair with hardware rasterization for close-up strands could address this limitation. Additionally, although the strand assembly step distributes work well across threads, the DDA rasterization stage exhibits workload imbalance

when threads draw lines of varying lengths. Increasing the number of control points alleviates this, but a more balanced work distribution scheme would further improve utilization. Similar, when the number of control points is significantly lower than the GPU subgroup size, processing multiple strands per workgroup could result in better SIMD lane utilization. Finally, there is room to enhance filtering quality and performance, notably through the integration of temporal filtering to reduce flickering during camera motion.

Acknowledgments

We would like to thank Olivier Maury for his continued support throughout this project. Lukas Lipp and Michael Wimmer have been supported by the Austrian Science Fund (FWF, Grant F77).

References

Advanced Micro Devices, Inc. (AMD). 2021. AMD FidelityFX Super Resolution. https://gpuopen.com/fidelityfx-superresolution/. Accessed: 2026-04-03.

Gaurav Bhokare, Eisen Montalvo, Elie Diaz, and Cem Yuksel. 2025. Real-Time Hair Rendering with Hair Meshes. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Papers '24*. ACM, Denver CO USA, 1–10. doi:10.1145/3641519.3657521

Wesley Chang, Andrew L Russell, Stephane Grabli, Matt Jen-Yuan Chiang, Christophe Hery, Doug Roble, Ravi Ramamoorthi, Tzu-Mao Li, and Olivier Maury. 2025. Transforming Unstructured Hair Strands into Procedural Hair Grooms. *ACM Transactions on Graphics (TOG)* 44, 4 (2025), 1–20.

Matt Jen-Yuan Chiang, Benedikt Bitterli, Chuck Tappan, and Brent Burley. 2016. A Practical and Controllable Hair and Fur Model for Production Path Tracing. In *Computer Graphics Forum*, Vol. 35. Wiley, 275–283.

Francesco Cifariello Ciardi, Lasse Jon Fuglsang Pedersen, and John Parsaie. 2022. Probe-based Lighting, Strand-based Hair System, and Physical Hair Shading in Unity's "Enemies". In *ACM SIGGRAPH Courses: Advances in Real-Time Rendering in Games Course*.

Robert L. Cook, John Halstead, Maxwell Planck, and David Ryu. 2007. Stochastic simplification of aggregate detail. *ACM Trans. Graph.* 26, 3 (July 2007), 79–es. doi:10.1145/1276377.1276476

Roc R. Currius, Ulf Assarsson, and Erik Sintorn. 2022. Real-Time Hair Filtering with Convolutional Neural Networks. *Proc. ACM Comput. Graph. Interact. Tech.* 5, 1, Article 15 (May 2022), 15 pages. doi:10.1145/3522606

Epic Games. 2021. Unreal Engine. https://www.unrealengine.com/. Accessed 2026-04-07.

Cass Everitt. 2001. Interactive order-independent transparency. *White paper, nVIDIA* 2, 6 (2001), 7.

T. Huang, J. Yuan, R. Hu, L. Wang, Y. Guo, B. Chen, J. Guo, and J. Zhu. 2025. Detail-Preserving Real-Time Hair Strand Linking and Filtering. *Computer Graphics Forum* 44, 4 (2025), e70176. doi:10.1111/cgf.70176

Toshiaki Ishihara and Hitoshi Mishima. 2023. *Resident Evil 4 Hair Discussion.* Technical Report. Capcom R&D.

Yibing Jiang. 2016. The Process of Creating Volumetric-Based Materials in Uncharted 4. In *ACM SIGGRAPH Courses: Advances in Real-Time Rendering in Games*. Anaheim, CA, USA. Course presentation.

Jorge Jiménez, Xianchun Wu, Angelo Pesce, and Adrian Jarabo. 2016. Practical real-time strategies for accurate indirect occlusion. *SIGGRAPH 2016 Courses: Physically Based Shading in Theory and Practice* (2016).

James T Kajiya and Timothy L Kay. 1989. Rendering fur with three dimensional textures. *ACM Siggraph Computer Graphics* 23, 3 (1989), 271–280.

Brian Karis, Rune Stubbe, and Graham Wihlidal. 2021. A Deep Dive into Nanite Virtualized Geometry. SIGGRAPH 2021 Course: Advances in Real-Time Rendering in Games. https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf Industry talk.

Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, George Drettakis, et al. 2023. 3d gaussian splatting for real-time radiance field rendering. *ACM Trans. Graph.* 42, 4 (2023), 139–1.

Chuan Koon Koh and Zhiyong Huang. 2001. A simple physics model to animate human hair modeled in 2D strips in real time. In *Proceedings of Workshop on Computer Animation and Simulation*. Springer, 127–138.

Sergei Kulikov. 2025. Strand-based Hair and Fur in "Indiana Jones and the Great Circle". In *ACM SIGGRAPH Courses: Advances in Real-Time Rendering in Games Course*.

Samuli Laine and Tero Karras. 2011. High-Performance Software Rasterization on GPUs. In *Proceedings of HPG*. Conference presentation.

Soham Uday Mehta, Ravi Ramamoorthi, Mark Meyer, and Christophe Hery. 2012. Analytic tangent irradiance environment maps for anisotropic surfaces. In *Computer Graphics Forum*, Vol. 31. Wiley Online Library, 1501–1508.

Martin Mittring. 2007. Finding next gen: CryEngine 2. In *ACM SIGGRAPH 2007 Courses* (San Diego, California) *(SIGGRAPH '07)*. Association for Computing Machinery, New York, NY, USA, 97–121. doi:10.1145/1281500.1281671

Zahra Montazeri, Soren Gammelmark, Henrik W Jensen, and Shuang Zhao. 2021. A practical ply-based appearance modeling for knitted fabrics. *arXiv preprint arXiv:2105.02475* (2021).

Zahra Montazeri, Søren B Gammelmark, Shuang Zhao, and Henrik Wann Jensen. 2020. A practical ply-based appearance model of woven fabrics. *ACM Transactions on Graphics (TOG)* 39, 6 (2020), 1–13.

NVIDIA Corporation. 2025. DLSS 4: Transforming Real-Time Graphics with AI. https://research.nvidia.com/labs/adlr/DLSS4. Accessed: 2026-04-03.

Leonid Pekelis, Christophe Hery, Ryusuke Villemin, and Junyi Ling. 2015. A data-driven light scattering model for hair. *Pixar Technical Memo* 2 (2015).

Ravi Ramamoorthi and Pat Hanrahan. 2002. Frequency space environment map rendering. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. 517–526.

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 *European Conference on Computer Vision*. Springer, 73–89.

Thorsten Scheuermann. 2004. Practical real-time hair rendering and shading. In *ACM SIGGRAPH 2004 Sketches*. 147.

Markus Schütz, Bernhard Kerbl, and Michael Wimmer. 2022. Software Rasterization of 2 Billion Points in Real Time. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 5, 3 (2022), 1–17. doi:10.1145/3543863

Arunachalam Somasundaram. 2015. Dynamically controlling hair interpolation. In *ACM SIGGRAPH 2015 Talks*. 1–1.

Sebastian Tafuri. 2019. Strand-based Hair Rendering in Frostbite. *ACM SIGGRAPH Courses: Advances in Real-Time Rendering in Games Course* (2019).

Robin Taillandier and Jon Valdes. 2020. Every Strand Counts: Physics and Rendering Behind Frostbite's Hair. In *Digital Dragons*. Conference presentation.

Nicolas Thibieroz. 2018. Order-independent transparency using per-pixel linked lists. In *GPU PRO 360 Guide to GPGPU*. AK Peters/CRC Press, 23–46.

Johannes Unterguggenberger, Lukas Lipp, Michael Wimmer, Bernhard Kerbl, and Markus Schütz. 2024. Fast Rendering of Parametric Objects on Modern GPUs. *Eurographics Symposium on Parallel Graphics and Visualization* (2024). doi:10.2312/PGV.20241129

Johannes Unterguggenberger, Lukas Lipp, Michael Wimmer, Markus Steinberger, Bernhard Kerbl, and Markus Schütz. 2025. Real-Time Rendering Methods with Adaptive Levels of Detail for Fast Rendering of Parametric Objects on Modern GPUs. *IEEE Transactions on Visualization and Computer Graphics* (2025), 1–12. doi:10.1109/TVCG.2025.3638697

Cem Yuksel and John Keyser. 2008. Deep opacity maps. In *Computer Graphics Forum*, Vol. 27. Wiley Online Library, 675–680.

Cem Yuksel, Scott Schaefer, and John Keyser. 2009. Hair Meshes. *ACM Transactions on Graphics* 28, 5 (Dec. 2009), 1–7. doi:10.1145/1618452.1618512

Zhongtian Zheng, Tao Huang, Haozhe Su, Xueqi Ma, Yuefan Shen, Tongtong Wang, Yin Yang, Xifeng Gao, Zherong Pan, and Kui Wu. 2025. Auto Hair Card Extraction for Smooth Hair with Differentiable Rendering. *ACM Transactions on Graphics (TOG)* 44, 6 (2025), 1–13.

Junqiu Zhu, Zahra Montazeri, J Aubry, Lingqi Yan, and Andrea Weidlich. 2023. A practical and hierarchical yarn-based shading model for cloth. In *Computer Graphics Forum*, Vol. 42. Wiley Online Library, e14894.

Junqiu Zhu, Sizhe Zhao, Lu Wang, Yanning Xu, and Ling-Qi Yan. 2022. Practical level-of-detail aggregation of fur appearance. *ACM Transactions on Graphics (TOG)* 41, 4 (2022), 1–17.

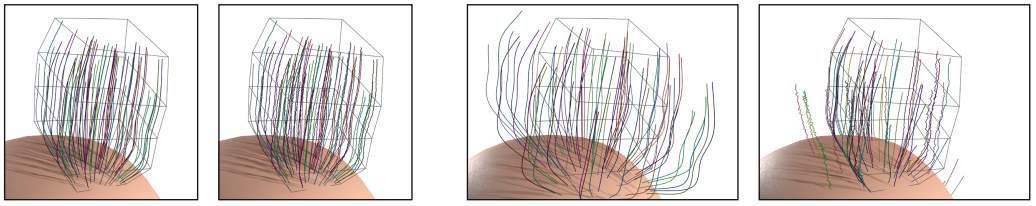


Fig. 8. The hardware-supported strand evaluation approach proposed by Bhokare et al. [2025] suffers from precision errors, which manifest as "jagged" or stepped artifacts (a). This issue arises because hardware interpolation is limited to 256 discrete steps between pixels (i.e., 8-bit subpixel precision). The resulting lack of resolution prevents hair strands from forming smooth, continuous curves, leading to a staggered appearance. Furthermore, hardware samplers can only interpolate within a predefined range and cannot extrapolate beyond it (b). Consequently, hair strands abruptly degenerate near bundle boundaries rather than continuing linearly. Our software-based solution does not exhibit these limitations.

A Appendix

A.1 Comparison: Hardware vs. Software Interpolation

In our implementation, we evaluated the benefits of using hardware-supported strand evaluation as proposed by Bhokare et al. [2025]. Their approach approximates a cubic spline using two quadratic splines, which are then evaluated via the de Casteljau algorithm. The expected performance gain arises from offloading a significant portion of the interpolation work to the GPU's texture hardware.

We compared our buffer-based software interpolation approach against the 3D texture-based method of Bhokare et al. [2025]. From a performance perspective, the hardware interpolation path is consistently slightly slower than the software-based approach (see Table 2).

From a visual standpoint, the hardware method exhibits precision issues and does not support extrapolation, leading to noticeable artifacts (see Figure 8).

Table 2. Performance comparison of hardware (texture-based) interpolation and software (buffer-based) interpolation across compute and mesh shader pipeline.

Method	MSAA	Interpolation	Time (ms)
Software Rasterizer	1	Hardware (Texture)	2.8
Software Rasterizer	1	Software	2.5
Mesh Pipeline	1	Hardware (Texture)	13.5
Mesh Pipeline	1	Software	13.0

B Pseudocode

```
float3 calcTangent(int idx, int count, float3 prev, float3 curr, float3 next) {
    if (idx == 0) return normalize(next - curr); // Root
    else if (idx == count - 1) return normalize(curr - prev); // Tip
    else return normalize(next - prev);
}
// We use a WG size of 32 since NVIDIA has a subgroup size of 32
[shader("compute"), numthreads(32, 1, 1)]
void main(uint3 dispatchThreadID : SV_DispatchThreadID) {
    int tid = int(dispatchThreadID.x); int WG = 32; int lane = WaveGetLaneIndex();
    BundleStrandInfo b = getBundleStrandInfo();
    PCG rand(b.strandIndex);
    float2 uv = rand.next2();

    // Pass 1: Load base strand layers into shared memory
    int S = (b.numLayers + WG - 1) / WG; // Layers per lane
    int first = tid * S;
    L_first = L_curr = getBundleLayer(b, first, uv); // First of current lane
    L_last = (S > 1) ? getBundleLayer(b, first + S - 1, uv) : L_curr; // Last of current lane
    L_prev = subgroupShuffleUp(L_last); // Tail of previous lane
    for (int i = 0; i < S && (first + i) < b.numLayers; ++i) {
        int idx = first + i;
        L_next = (i == S - 1) ? subgroupShuffleDown(L_first) : ((i == S - 2) ? L_last :
        ↪ getBundleLayer(b, idx + 1, uv)); // Next, last or shuffle
        smLayer[idx].tangent = calcTangent(idx, b.numLayers, L_prev.pos, L_curr.pos, L_next.pos);
        smLayer[idx].pos = L_curr.pos; // Write all other attributes...
        L_prev = L_curr; L_curr = L_next;
    }
    GroupMemoryBarrierWithGroupSync();

    // Pass 2: Evaluate styled strand and draw segments
    int N_cp = (b.numLayers - 1) * b.segmentsPerInterval + 1;
    S = (N_cp - 1 + WG - 1) / WG; // CPs per lane
    first = tid * S;

    CP_first = CP_curr = evalStyledCP(first); // First of current lane
    CP_last = (S > 1) ? evalStyledCP(first + S - 1) : CP_curr; // Last of current lane
    CP_prev = subgroupShuffleUp(CP_last); // Tail of previous lane
    CP_next = (S == 1) ? subgroupShuffleDown(CP_first) : ((S == 2) ? CP_last : evalStyledCP(first +
    ↪ 1)); // Next, last or shuffle
    T_first = T_curr = calcTangent(first, N_cp, CP_prev, CP_curr, CP_next); T_prev = {};
    for (int i = 1; i <= S; ++i) {
        int idx = first + i;
        CP_prev = CP_curr; CP_curr = CP_next; T_prev = T_curr;
        CP_next = (i >= S - 1) ? subgroupShuffleDown(CP_first) : ((i == S - 2) ? CP_last :
        ↪ evalStyledCP(idx + 1)); // Next, last or shuffle
        T_curr = (i >= S) ? subgroupShuffleDown(T_first) : calcTangent(idx, N_cp, CP_prev, CP_curr,
        ↪ CP_next);
        // Handle both center and conservative layer writes
        if(idx < N_cp) drawSegment(CP_prev, CP_curr, T_prev, T_curr);
    }
}
```

Code 1. Simplified Software Hair Rasterizer. Dispatched per strand.

```
RWTexture2D<uint64_t> C; // Center-Color
Texture2D<uint64_t> G; // Center-G-Buffer
Texture2D<uint64_t> A; // Conservative-G-Buffer
bool isEmpty(uint64_t data);

[shader("compute"), numthreads(8, 4, 1)] // in case of subgroup size of 32
void main(uint3 dispatchThreadID : SV_DispatchThreadID)
{
    int2 P = int2(dispatchThreadID.xy);
    if (P >= C) return;

    uint64_t G_P = G[P], A_P = A[P];
    if (isEmpty(G_P) && isEmpty(A_P)) return; // Early out
    float4 C_P = C[P]; // Center Color

    bool isCenter = !isEmpty(G_P), isCons = !isEmpty(A_P);
    float3 t_P; float z_P;
    if (isCenter) t_P, z_P = unpackTangentAndDepth(G_P) // Center Sample
    else if (isCons) t_P, z_P = unpackTangentAndDepth(A_P); // Conservative Sample
    float2 T_P = getScreenTangent(t_P);

    float4 S = {}; float W = 0.0, H_center = 0.0, H_cons = 0.0, w_strip = 0.5; int r_filter = 7;
    for (int dy = -r_filter; dy <= r_filter; ++dy) {
        for (int dx = -r_filter; dx <= r_filter; ++dx) {
            int2 Q = P + int2(dx, dy);
            if (Q < 0 || Q >= C) continue; // Bounds check

            float2 d = float2(dx, dy);
            float d_∥ = d·T_P;
            float d_⊥ = ‖d − d_∥ T_P‖;

            if (d_⊥ <= w_strip) { // Within Strip
                if (isCenter) H_center++;
                else if (isCons) H_cons++;
            }

            float4 C_Q = isEmpty(C[Q]) ? float4(0.0) : unpackColor(C[Q]); // Neighbor color
            float w_s = exp(−d_∥²/σ_∥² − d_⊥²/σ_⊥²); // Spatial
            float w_c = exp(−‖C_P−C_Q‖²/σ_c²); // Color
            float w = w_s * w_c; S += C_Q * w; W += w;
        }
    }

    float4 Ĉ_P = (W > 0.0) ? (S / W) : C_P;
    float α = ((H_center + H_cons) > 0.0) ? H_center / (H_center + H_cons) : 1;
    C[P] = packColor({Ĉ_P, α}); // Write back to the Center-Color texture
}
```

Code 2. Simplified Orientation-Aware Anisotropic Hair Reconnection Filter. Dispatched per pixel.