Eurographics Symposium on Rendering (DL-only Track) (2019)
T. Boubekeur and P. Sen (Editors)

Real-Time Hybrid Hair Rendering

Erik S. V. Jansson[1]   Matthäus G. Chajdas[2]   Jason Lacroix[2]   Ingemar Ragnemalm[1]

[1]Linköping University, Sweden   [2]Advanced Micro Devices, Inc.

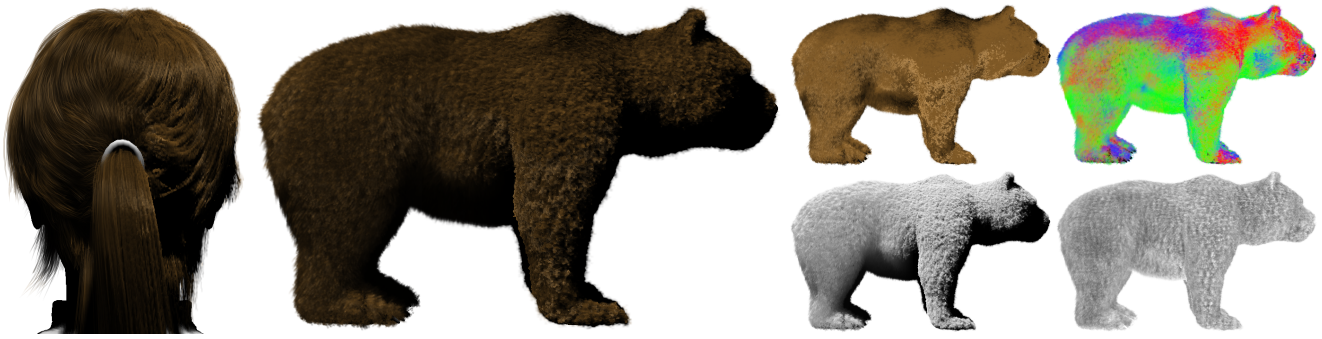

Figure 1: *We present a volume-based approximation of strand-based hair that is suitable for level-of-detail minification. It scales better than existing raster-based solutions with increasing distances, and can be combined with them, to create a hybrid technique. This figure shows the split alpha blended level-of-detail transition between our strand-based rasterizer (left side) and its volume-based approximation (right side). The four subfigures on the right showcase the Bear's individual components: Kajiya-Kay shading, tangents, shadows and ambient occlusion.*

Abstract

Rendering hair is a challenging problem for real-time applications. Besides complex shading, the sheer amount of it poses a lot of problems, as a human scalp can have over 100,000 strands of hair, with animal fur often surpassing a million. For rendering, both strand-based and volume-based techniques have been used, but usually in isolation. In this work, we present a complete hair rendering solution based on a hybrid approach. The solution requires no pre-processing, making it a drop-in replacement, that combines the best of strand-based and volume-based rendering. Our approach uses this volume not only as a level-of-detail representation that is raymarched directly, but also to simulate global effects, like shadows and ambient occlusion in real-time.

CCS Concepts

• ***Computing Methodologies*** $\rightarrow$ ***Rasterization; Visibility; Volumetric Models;*** *Antialiasing; Reflectance Modeling; Texturing;*

1. Introduction

Hair rendering and simulation are challenging problems in the field of computer graphics due to the complex nature of hair [WBK*07]. The human scalp typically has over 100,000 sub-pixel sized strands of hair attached to it [WBK*07], making it non-trivial to accurately represent in practice. There are two main models to represent hair: as a *volume*, or by using *explicit hair strands* [WBK*07]. Each of these have their own strengths and weaknesses; with volume-based representations for instance, finding effects like ambient occlusion is trivial [HLY10], but not in strand-based representations, that use rasterization, leading to cheap screen-space techniques. In real-time applications, such as games, the most widely used representation in practice is to use explicit hair strands [ND05; YT10; MET*14], as they have proven to work well in real-time hair simulation [Han14].

Hair rendering still remains a challenging problem, however, as existing raster-based solutions [ND05; MET*14] only have enough budget to render a couple of characters on the screen at once in real-time frames. This happens because rasterization does not scale well for this type of sub-pixel sized geometry [Ric14], for instance, with far away hair. Games typically solve this scaling problem by using a level-of-detail scheme, like reducing the number of hair strands dynamically [YT10; SD15], and compensating for any volume loss by making strands thicker. This last approach suffers from the same dilemma found in mesh simplification [CMS98], as visual features important to the hair style could be evicted by accident. Yet another issue with strand-based rasterization, is that global effects, such as ambient occlusion, are not easy to compute accurately in real-time. In this paper we present a new approach that uses both strand-based and volume-based representations, a hybrid, to resolve these issues.

Our contribution is a real-time hybrid hair rendering pipeline that uses strand- and volume-based representations together. It is faster, and scales better, for far away distances when compared to existing purely strand-based rasterizers like TressFX. We do this by finding a volumetric approximation of the strand-based representation with a fast voxelization scheme. This volume is used to do raymarching, and then shading on an isosurface, providing a novel level-of-detail minification technique for strand-based hair. It can be used together with a strand-based rasterizer to create a full hair rendering solution that smoothly scales in the performance & quality domain. We also use the volume to estimate global effects, like ambient occlusion, a problem that's not solvable with strand-based rasterization by itself. Pre-processing is not required in our pipeline, and can even be used for fully animated or simulated hair, as it voxelizes once per-frame.

We start off by talking about existing work within hair rendering, and then describe our hybrid hair rendering pipeline. We later show results that demonstrate the benefits of our method, and talk about its limitations. All the source code for the reference implementation is available from: `https://github.com/CaffeineViking/vkhr`.

2. Related Work

Several hair rendering frameworks have been presented in real-time computer graphics. Most of these are strand-based [YYH*12], and support light scattering, self-shadowing and transparency, as these are important aspects in hair rendering [WBK*07]. The framework by Yu et al. [YYH*12], that is also used for TressFX 3.1 [MET*14], is used in real-time applications, such as games [Lac13; SD15], but doesn't allow for more than a few characters to be on the screen at the same time. Our hair rendering pipeline is able to render multiple hair styles, by using a scalable level-of-detail minification based on a volumetric approximation of hair that is raymarched in real-time.

Rendering hair as a volume has been done before for both offline [PHA05; MWM08; XDY*15] and interactive [RZL*10; XDY*12] settings. In our solution we use this volume for many purposes, not only for rendering. This includes using it as a scalable complement to strand-based rasterization that can be used for real-time contexts. This has not been explored in previous work within hair rendering.

However, strand-based methods result in higher-quality hair, and are generally used for simulation [Han14] in real-time applications.

The only other work which combines strand-based and volume-based representations into a hybrid approach for hair rendering is Andersen et al. [AFFC16]. They use it for fur rendering, as explicit hair strand geometry is not suitable to model undercoat hairs, while volume textures excel at it. While they also use the raymarcher and rasterizer together, like us, they do not use it for hair level-of-detail.

Another paper that proposes a hybrid level-of-detail approach is Loubet and Neyret [LN17]; they use a mesh for macroscopic details and switch to a pre-filtered volume for microscopic surfaces. It's a appearance preserving technique for complex geometry, that scales across different LoDs. This technique can be used in hair rendering, but because of its expensive pre-filtering [HDCD15], it can not be used in real-time, since hair could be animated. The most common level-of-detail approach for games is to reduce the hair strands like in Steward et al. [SD15]. Ward et al's. [WLJ*03] method is better at preserving visual details, but it comes with a greater run-time cost.

3. Hybrid Hair Rendering Pipeline

The input to our pipeline consists of *hair strands*, which we model as an array of *straight-line segments*, as shown in Figure 2. This is a common way to model hair [WBK*07], and it is used by several existing systems, such as TressFX, for both simulation [Han14] and rendering [MET*14]. Some of these renderers, including TressFX, expand these segments into *billboards* to support varying the strand thickness [YT10]. While there is nothing limiting our pipeline from also doing this, we have chosen to use *line rasterization* directly, as it leads to a fast coverage algorithm, and also because the expansion to camera-facing triangles is expensive [YT10]. This means that all strands have *constant radius* in our pipeline, and are assumed to be of *sub-pixel* size [LTT91], to simplify our volume-based technique.

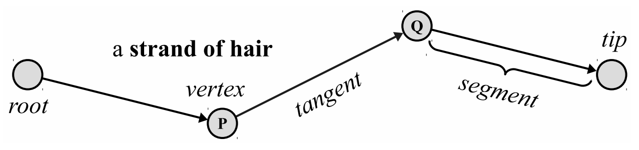

Figure 2: *The geometry given to our pipeline, and its terminology.*

Overview. The pipeline itself consists of a *strand-based rasterizer*, similar to the one in TressFX, and a *volume-based approximation* of the rasterizer's output. Our approach is to use this approximation for level-of-detail, as the raymarcher has better performance scaling with distance than the rasterizer. The idea is to use the high-quality rasterized results for "close-up shots", and the fast but also scalable raymarcher for the far-away cases. The output from both renderers are closely matched, making the transition between them seamless. A simple dithered or alpha blended transition is enough to mix them. Since both of the solutions use the same hair geometry as input, our pipeline easily integrates into existing hair renderers, such as TressFX, that already have strand-based hair for simulation.

Strand-Based Rasterizer. Our rasterized solution renders strands of hair as line list primitives with the GPU's built-in line rasterizer, and shades each fragment using a hair strand *lighting model* which uses tangents for shading. Since hair is sub-pixel sized [LTT91], it leads to high aliasing, which we solve by using the pixel *coverage* of a line for *anti-aliasing*. Hair is not completely opaque [SA08], and since our coverage calculation modifies the alpha component, we need to *sort*, and then *blend*, the fragments in the correct order to handle strands with *transparency*. Another important property of hair are *self-shadows* [WBK*07], that we solve with a approximate technique, that we later generalize using a volume. This results in a high-quality, but expensive, real-time solution which scales poorly.

Volume-Based Approximation. To make up for this weakness in our rasterizer, we developed a volumetric hair approximation that is faster to compute, and that scales better with increasing distances, when comparing it against our rasterized solution. These properties make it a good *level-of-detail* technique for hair. The idea is to find a density volume with *strand voxelization*, and then use that for raymarching. In order to approximate the *shading* of the rasterizer we also need to voxelize the strand's tangents. To find an isosurface to shade we accumulate the density (which is the same as the number of strands in the way) until a threshold is reached. Since we don't shade inside the volume (as that's too expensive) the transparency also needs to be approximated by treating dense regions differently.

We approximate the strand self-shadowing component by using the volume as well, as a raymarch towards the light source gives us the number of occluding hair strands. This is the same information needed for our rasterizer's self-shadowing model, which we simply plug in. To also account for the *ambient occlusion* going on around a strand, we accumulate the number of strands inside a sphere. This leads to a visibility component that our rasterizer also needs to use.

Since both the rasterizer and raymarcher use similar techniques for each component, the level-of-detail transition becomes smooth.

Because hair is not usually static, the underlying geometry may change in-between frames as part of a simulation or animation pass. This means pre-computations are generally not possible, unless the geometry can be assumed to be static (e.g. short hair). Our pipeline is quite flexible, and re-computes its components every frame. If the hair is static, our *strand voxelization* and *ambient occlusion* passes can be pre-computed once, or at simulation frequency, to save time.

In the following sections we go into more detail on how we have solved each of these problems, and how these tie into our pipeline.

3.1. Lighting Model

After rasterizing the strand segments with the GPU's line rasterizer, we shade each fragment with Kajiya-Kay's [KK89] lighting model, that estimates the light scattering inside a strand with a *diffuse* and *specular* term. It is not physically-based [MJC*03; dFH*11], but it is still widely used [WBK*07; YT10] for real-time applications for its simplicity and performance (or variants of it [Sch04; MET*14]).

For a single light source, it produces the characteristic highlights of hair as shown in Figure 3, which are based on a strand's tangent. Because hair is very thin, rasterization leads to aliasing artifacts as shown on the left. We solve this issue in Section 3.2, and also blend transparent strands. This still leads to unnaturally flat-looking hair, because self-shadows are not fully accounted for until Section 3.4.

This lighting model uses the *tangent* for shading, which is trivial to find in rasterization: $(\mathbf{Q}-\mathbf{P}) \div |\mathbf{Q}-\mathbf{P}|$, but not within a volume. We show how to voxelize these and use it for shading in Section 3.3.

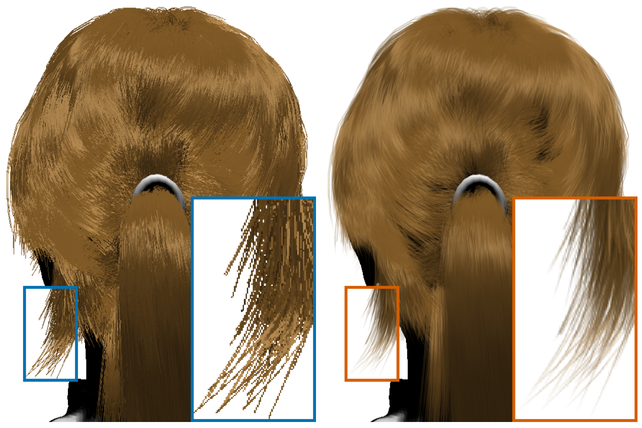
Figure 3: *Kajiya-Kay shaded results from our strand rasterizer, on the left without any anti-aliasing, and with our coverage method on the right. A correct blending order is needed, since our calculation modifies the alpha channel (we use a PPLL to sort these fragments).*

3.2. Transparency and Anti-Aliasing

Because strand rasterization by itself gives severely aliased results, we need an anti-aliasing solution to remove these artifacts. A naïve solution would be to use MSAA, and while this reduces aliasing, it comes at a significant cost, that also doesn't solve for transparency, that we need for natural hair. Instead we choose to find the strand's pixel coverage in screen-space, which we can use to vary the strand opacity based on pixel coverage [Per12]. This method does not require any extra fragments to be evaluated, but because our strands are now transparent, these fragments need to be sorted and blended in the right order. But since hair strands are already non-opaque, we would have to handle transparency anyway. We start by explaining our coverage algorithm, and then show our transparency technique.

Coverage. Our coverage calculation is similar to GPAA [Per12], but without the post-processing pass, as we only need to draw lines. The idea is to render the lines with constant width, and then when shading the line, find the distance between its center and the shaded fragment (that will be non-zero as long as the line width is not 1). In practice, this means converting the interpolated vertex positions in the fragment shader to screen-space, and then finding the distance from that to `gl_FragCoord`, and dividing by the line width to get a normalized number. Below is how it looks like as a GLSL shader.

```
float strand_coverage(vec2 screen_fragment, vec4 world_line,
                      mat4 view_projection, vec2 resolution,
                      float line_thickness) {
    vec4 clip_line = view_projection * world_line;
    vec3 ndc_line = (clip_line.xyz / clip_line.w);
    vec2 screen_line = ndc_line.xy;
    screen_line = (screen_line + 1.0f) * (resolution / 2.0f);
    float d = length(screen_line-screen_fragment);
    return 1.00f - (d / (line_thickness / 2.00f));
}
```

Sorting and Blending. In order to draw transparent strands of hair we need to sort and blend the non-opaque hair fragments in back-to-front order. There are several of ways to solve this [MCTB11], with their own set of strengths and weaknesses. We have chosen to use the same transparency solution as TressFX [MET*14], as that has been proven to work for real-time hair rendering. It is based on the method by Yu et al. [YYH*12], which uses a Per-Pixel Linked List (PPLL) [YHGT10] and *k*-buffer approach [BCL*07] to handle transparency. For it to work, writes to the depth buffer are disabled.

Instead of composing our fragments directly to the render target, we insert them into a PPLL with their color and depth information. It is built concurrently on the GPU [YHGT10], and is then sorted in parallel based on depth in a separate resolve pass. We use the same approach as in TressFX, and only blend the first *k* fragments in the correct order, while the remaining are merged order-independently.

By using a *k*-buffer with 16 elements, we get results like those on the right of Figure 3. Most of the aliasing artifacts are now gone, and because we have now also taken into account the opacity of the strands, the shading is smoother and more natural. The problem is that this is expensive, as the hair in Figure 3 takes between 5-6ms to render. This would only allow us to render a couple of characters on the screen at once. Even worse, the performance doesn't scale nicely with distance, as the same hair style, but only covering 0.5% of the display (instead of 30%), takes 3ms. In the next section we present our level-of-detail technique that improves on these aspects.

3.3. Level of Detail

While the strand-based rasterizer we have described so far produces fine-grained results for each individual hair strand, making it suitable for close-up shots, performance does not scale proportionally with increasing distance (and fragments shaded). For far-away hair these details will not be noticeable anyway. Thus, in order to render more than a couple of characters on the screen at once, in real-time, we need a scalable level-of-detail scheme. A commonly used one is to reduce the number of hair strands based on distance [WBK*07]. This removes visual features from the hair that could be important to the hair style. A critical part of a good level-of-detail scheme is that the transitions should be seamless. The way to circumvent this is to have an artist markup important regions of the hair style, and only evict these strands as a last resort. Another common solution in games [Lac13] is to have artists hand-tailor different assets for each level-of-detail. Both of these solutions require varying amounts of manual work by the artist. It would be economical if there was an automatic technique that does not suffer from these same problems.

Instead, we use a volume-based approximation of this hair style as level-of-detail. This representation has many benefits over these raster-based level-of-detail schemes, as it uses raymarching instead of rasterization. Rasterization performs badly on this type of small geometry, as it is numerous, and, whenever projected from far-away distances, falls only onto a few thousand fragments. This becomes especially problematic when used with our transparency technique, as the PPLL will have a lot of fragments to sort and blend per-pixel. Raymarching on the other hand, thrives in these situations, as it gets linear performance scaling based on the number of fragments. Our volume used for raymarching is derived from the original geometry, and not a reduced version of it, which means it will retain all of the visual features of the original hair style, but in a discretized format. Since this voxelization can be done on-the-fly, it doesn't need any artist intervention, and can be used with animated or simulated hair.

Our volume-based approximation for strand-based hair uses the same underlying geometry as before: line segments. It takes these, and voxelizes them into two different volumes: the *density volume*, and the *tangent volume* with GPU-based compute. The densities are used to represent the number of strands that have passed through a voxel, which we use to do our raymarch later. The tangents are also needed as a volume because they can't be derived from the density, unlike the normals which are just $-\nabla d$, the gradient of the density.

We use these volumes to approximate the rasterizer's shading on an isosurface. In our pipeline this means finding approximations of the lighting model, self-shadowing algorithm, and transparency in a volume. More lighting components can likely also be approximated in real-time by using these volumes, but we have chosen to limit us to the subset of effects that are commonly used when rendering hair in real-time. We show how to approximate self-shadowing and the ambient occlusion in Section 3.4. Before we can do this though, we need to find an isosurface to shade on. In our approach, we do not use Direct Volume Rendering (DVR) [HKSB06], as shading inside the volume would be too expensive for our real-time needs. Instead, we only do shading on an isosurface. We start with explaining how to find this isosurface, and then go on to show how to do shading on it by using the volumes we have derived. We also show how to fake transparency without having to use DVR, by using the hair density.

Surface. We start by building an AABB of the hair by expanding a cuboid with each incoming vertex. This is our proxy geometry, that houses our volume data, and which we use for raymarching. As we can't render volumes with a rasterizer, we render our cuboid, which gives us the fragment $\mathbf{f}$ on its surface. We use $\mathbf{f}$ and the eye $\mathbf{e}$ to find the direction $\mathbf{f}-\mathbf{e}$ in which we should be raymarching into, with an origin set for $\mathbf{f}$, we raymarch into this direction with constant steps.

Next, we want to find an isosurface $\mathbf{s}$ inside this volume to do all our shading on. As we mentioned before, the hair densities contain the number of strands passing through each voxel. A raymarch from a voxel at $\mathbf{a}$ to another voxel at $\mathbf{b}$ gives us the number of hair strands in-between them. We use this to find the isosurface $\mathbf{s}$, as we classify a surface as "solid" only if it passes a minimum strand count. With a raymarch from $\mathbf{f}$ launched towards $\mathbf{f}-\mathbf{e}$, we accumulate strands until a "solid" isosurface $\mathbf{s}$ is found. We use this surface $\mathbf{s}$ to do our shading. In order to "fake" transparency we also shade "non-solid" isosurfaces, by finding the first intersection: a non-empty voxel, and changing its opacity based on the density. This leads to a smoother level-of-detail transition, especially for low-density (stray) regions.

Shading. Shading is straightforward once this isosurface is found, and as long as a tangent volume can be voxelized. We simply shade the fragment using standard Kajiya-Kay shading, by using the same shader as our rasterizer, but replacing the analytical tangent with an approximation, found by querying the tangent volume at $\mathbf{s}$. We use a direct translation of Kajiya-Kay's [KK89] algorithm for shading.

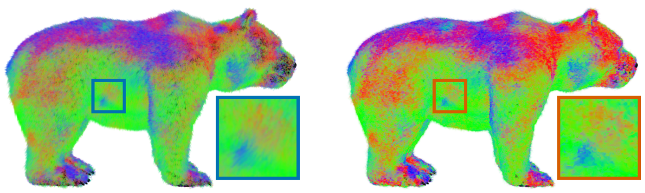

Figure 4: *Comparison between the rasterized tangents (on the left), and their volumetric approximation (on the right). We find it by first quantizing the tangents, voxelizing them, and then finding the mean tangent vector after de-quantization for use in Kajiya-Kay shading.*

Strand Voxelization. Finding the strand density can be done fast with GPU compute. Each voxel stores a 8-bit counter that tells us how many strands are in it. This means we can keep track of up to 255 strands of hair per voxel. The voxelization happens by translating the hair style's vertices to volume-space (with just a coordinate-change) and increasing the counter of the voxels it's mapped to. For GLSL, the `imageAtomicAdd` function can be used to increment this counter asynchronously. This is an approximation, and leads to "gaps" in the volume (because of line spacing), which we cover by using a filter. We have found this to work well in practice for our hair styles, but for the cases where segments are long, this approach will break down, as repairing the volume will require bigger filters.

To find the tangent volume we use something similar, but we find the average direction instead of the sum. We start by quantizing our tangents from $[-1.0,+1.0]$ into $[0,255]$ per-channel, and summing those values up, just as for the density volume. The trick is then to divide this by the number of hair stands in each voxel, which is the same as our density volume, and then de-quantizing it from $[0,255]$ back into $[-1.0,+1.0]$ when shading. See Figure 4 for these results.

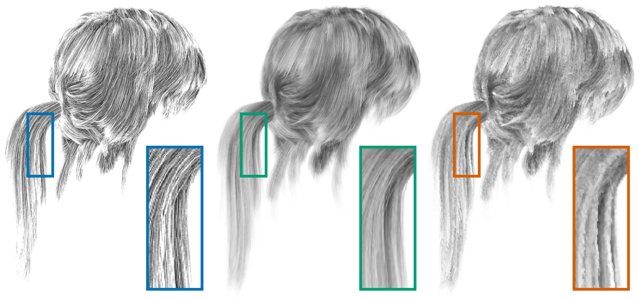

Figure 5: *Differences between the raytraced ambient occlusion on the left, and our volume-based approximation of it on the right. The middle result is the AO from our strand-based rasterizer, which also uses a volume. We have used a 256^3-sized hair volume for this case.*

3.4. Self-Shadowing

After solving transparency and light scattering, our results shown in Figure 3 will still be flat looking, because strand self-shadowing is an essential part of rendering realistic hair [WBK*07]. There are several self-shadowing algorithms in related work, but we need one that is fast, and which maps to both rasterization and raymarching.

We have chosen to use a novel technique due to Lacroix [Lac13], that is used in Tomb Raider and TressFX [MET*14; SD15], as it is very fast, and only requires one single shadow map per light source, which is not true for Deep Opacity Maps [YK08]. It's based on an approximation of the Deep Shadow Map (DSM) [LV00], called the Approximated DSM (ADSM). First it finds the traversed depth d in the hair by the distance between the strand fragment f and the depth in the shadow map s. It uses the distance d to estimate the number of strands in the way: $d \cdot r$, where r is the expected spacing in-between hair strands. These are used to find the occlusion: $(1-\alpha)^{dr}$, that we use to find the shadows for a strand with translucency α. This will result in a smooth looking shadow, that becomes darker the further we get inside the hair. An issue with this technique is that it assumes constant strand spacing, which is generally not true for most styles.

Because we now have a hair density volume, we can actually use it to find the real number of hair strands in the way without making assumptions about the strand spacing. We raymarch in the direction of the light source, and accumulate the number of strands we have passed. This value is then used to replace the ad-hoc $d \cdot r$ in ADSM.

Ambient Occlusion. This only accounts for directional occlusion, but we would also like to model ambient occlusion (AO), based on nearby hair strands. This is not possible in real-time for a rasterizer, and can't be pre-calculated since the hair could be animated. For a raytracer, the general approach is to shoot rays in random directions around the sample point, and find the ratio that are being occluded. By using our volume, we can find this AO in real-time rates as well.

Our ambient occlusion is similar to Hernell et al's LAO [HLY10] and Kanzler et al. [KRW18]. A voxel with few hair strands in it will occlude less than a voxel with many hair strands in it. We raymarch in a sphere around our sample point, accumulating the total number of possible occluding hair strands inside it. This ratio of occluding hair strands is our ambient occlusion, but it needs to be tweaked to match the raytracer's intensity as shown in Figure 5. Our rasterizer also uses this volume-based AO, which is calculated in every frame.

4. Results

We have implemented our hybrid hair rendering pipeline in Vulkan, and evaluated its performance on an AMD Radeon Pro WX 9100. Our evaluation consists of comparing our strand-based rasterizer, and volume-based raymarcher, in order to determine in which cases they perform the best. Because our rasterizer is based on TressFX 3, which in turn is derived from [YYH*12], the results we show here should also translate over to other hair renderers built on the same set of techniques. The benchmark scenes we have built consists of a single directional light source pointed at a hair style, layered on top of a polygon mesh. The hair styles we have used in our evaluations are shown in Table 1, and are of comparable size with what is used in hair rendering research [YK08; RZL*10; AFFC16], and is $\approx$ 7x more strands of hair than used in games [Lac13]. In our benchmark we render all of our scenarios in 1280x720 resolution (without any MSAA), render into 1024x1024-sized 3x3 PCF shadow maps, use voxelizations of 256^3-sized volumes, and raymarch with 512 steps.

Hair Style	Strands	Vertices	Segments	Size (MB)
Ponytail	136,320	1,772,160	1,635,840	62.71
Bear	961,280	4,806,400	3,845,120	165.34

Table 1: *Summary of the data sets we are using for our evaluation.*

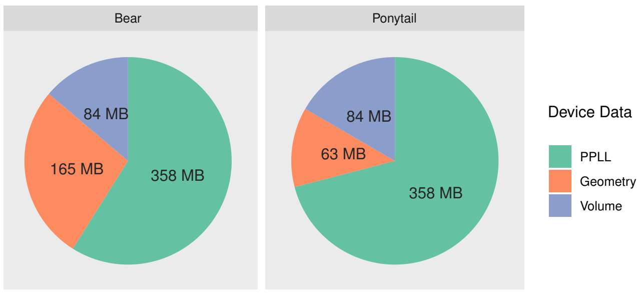

Figure 6: *GPU memory usage breakdown. Our volumes don't have to use inordinate amounts of memory when compared to its original geometry, and can be adjusted by voxelizing at lower resolutions. The primary "bottleneck" is the PPLL that's used for transparency.*

We start off with comparing the visual fidelity between our rendering solutions: the rasterizer and raymarcher, and see how well they transition in-between each other. In Figure 1 are screenshots from our hybrid hair renderer, using the same rendering conditions we detailed above. The left side in each individual image shows the result of the rasterizer, while the right side shows the raymarcher's result. These results are alpha blended in the middle of the image to demonstrate a level-of-detail transition. The rasterizer produces fine details up-close as hair strands are individually distinguishable. This is not true for the raymarcher, as it is easy to tell which of the sides are rasterized or raymarched for close-up shots. For far away to medium distances however, these details are not noticeable, and the raymarched result can be used without much loss in visual fidelity. Because these level-of-detail transitions are smooth, both of these solutions can be used, the rasterizer for close-up shots, and the raymarcher for far away shots. The reason we use a raymarcher in these cases is because it is cheaper to compute, scales better with distance, and is more runtime configurable, than our raster solution.

4.1. Performance

To show the performance benefits of using our hybrid approach, we have constructed a benchmark to compare the timing and scaling of our rasterized and raymarched solutions. We have gathered these by using Vulkan timestamp queries, which are averaged over a period of 60 frames, with VSync turned off, with no other GPU workloads.

We begin by discussing the frame timings for each solution, and show where these numbers come from. In Figure 7 we see that the raster solution spends most of its time shading (*Draw Hair Styles*), sorting, and blending transparent fragments (*Resolve the PPLL*). In *Bake Shadow Maps* the cost may seem small, but it scales with the number of lights in a scene, and also requires extra GPU memory. The raymarcher on the other hand, spends most of its time shading and raymarching (*Raymarch Strands*), and since it doesn't need any shadow maps, no extra memory is needed (besides the volume). In both cases though, *Voxelize Strands* is cheap, and only takes a small part of the rendering time, or none, if the hair can be assumed static.

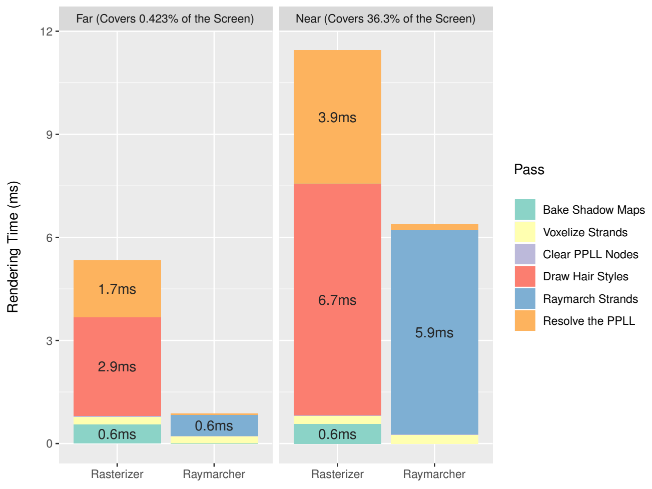

Figure 7: *Averaged render times divided into rendering passes for each renderer type. Voxelization accounts only for a fraction of the rendering time. We rendered the Ponytail in near and far distances, and our raymarcher outperforms the rasterizer, especially in "far".*

We have also measured performance from different distances to see how our solutions scale. For *"near"* distances, e.g. in a game's cutscene (covering 36.3% of the screen), our raymarcher is around twice as fast as our rasterizer, and allows twice the number of game characters to be on the screen at once, for the same time budget. In the *far away* case (0.423% of the screen), it is around 5x faster. This performance scaling difference can more easily be seen in Figure 8. The raymarcher scales linearly, while the rasterizer doesn't, and the former has a lower constant cost attached to it. While both rasterizer and raymarcher achieve real-time frames, our results point towards our raymarcher being more apt for level-of-detail minification than our rasterizer, for characters with a realistic amount of hair strands.

As can be seen in Figure 10, the rasterizer scales linearly with the hair strands, and so does our raymarcher, but with a "shallow" slope, caused by the voxelization in Figure 9. The quality of our raymarch and its performance can be tuned by varying the step size in Figure 9, making it possible to change the intercept of Figure 10.

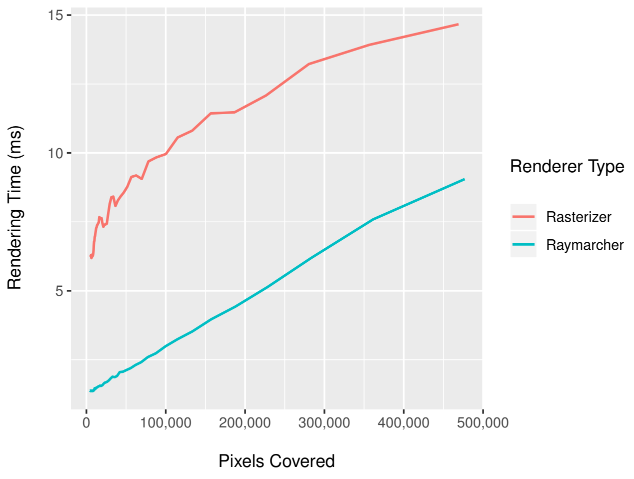

Figure 8: *Average rendering time of both solutions with decreasing distances (i.e. increasing screen space) for the Bear hair. Note that our raymarcher scales linearly, while the rasterizer does not, for far away hair. The raymarcher also has a lower constant cost attached.*

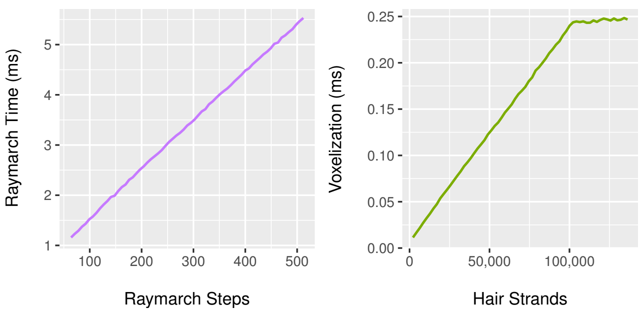

Figure 9: *Some performance scaling aspects in our raymarcher for Ponytail hair. Voxelization is cheap, and performance can be tuned. This seemingly flat region in the voxelization is not a measurement error, we believe it to be caused by the GPU cache being populated.*

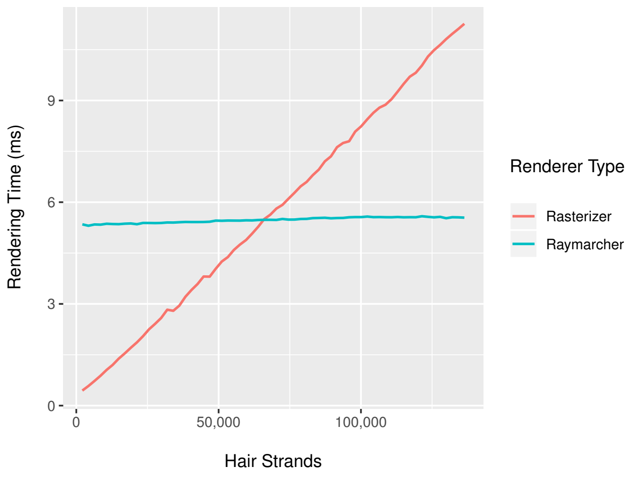

Figure 10: *Average rendering time of our solutions with increasing number of hair strands for the Ponytail hair. The raymarcher is only slightly affected because of the voxelization cost, and will otherwise be constant. This constant can be tuned with the raymarching steps.*

4.2. Limitations

While our strand voxelization strategy is quite fast, it does not work well for all cases. If the spacing between strand vertices is too large, or the grid resolution too granular, even a large reconstruction filter won't work, and will also yield diminishing returns in performance. For those cases, lowering the volume resolution will help, but it will result in a lower-quality result. A solution would be to use e.g. DDA and rasterize the lines using compute. Our experiments have shown that this is quite expensive, and not suitable for real-time rendering, as confirmed by Kanzler et al. [KRW18]. A promising method that was suggested to us is to sample the strand with the right frequency.

For mobile targets our solution may be too memory intensive, as we saw in Figure 6, the PPLL consumes a lot of GPU memory that is not readily available on these memory constrained platforms. We also haven't tried our solution with more advanced lighting models, like [ND05], and it would be interesting to see if our raymarcher is able to account for these in real-time as well, while still performing better than the rasterizer. Another issue with our raymarcher is that it uses constant step size, and may result in staircase artifacts. There is also a fixed cost attached to clearing the volumes for each frame; but the cost can be "hidden" by doing asynchronous compute work.

5. Conclusion

In this work we have presented a novel real-time hair renderer that is based on a new hybrid approach. It uses a strand-based rasterizer for close-ups and a volume-based approximation for level-of-detail minification. We have shown that our raymarcher consistently outperforms our rasterizer, especially for far away cases, since it scales better at increasing distances. The level-of-detail transition between them is smooth, since we estimate: light scattering, self-shadowing, transparency, and ambient occlusion for both solutions. We find this volumetric representation with a fast strand voxelization algorithm that works well for many hair styles. It can be found in real-time to support fully animated or simulated hair, which means it can easily be added to existing hair rendering frameworks, like TressFX, that already uses strand-based representations for their simulation pass. We have also shown that this volume can be used to estimate global effects, such as ambient occlusion, in real-time. This is not possible in purely raster-based solutions, but it is with our hybrid approach.

More broadly, our paper has shown that volumetric methods are a viable alternative to raster-based hair rendering when it comes to real-time applications. This has largely been unexplored in related work, as most volumetric methods are either interactive, or offline.

We expect future work to focus on improving the volume-based approximation we have presented. The raymarcher we have shown spends most of its time finding an isosurface, because we are doing constant sized steps. It would be very interesting to see if an signed distance field can be constructed in real-time, as that would improve our raymarching time considerably, and also fix the artifacts we get when not sampling the volume at a high enough frequency. Another aspect that could be further explored is to try other lighting models besides Kajiya-Kay, as we do not know if our raymarched solution will generalize to more advanced models. Finally, it would also be interesting to look at what other global effects can be approximated using our density volume, and if they can still be done in real-time.

Acknowledgments

We would like to thank AMD for providing us the ponytail and bear hair styles bundled in TressFX 3.1. We would also like to thank our anonymous reviewers at LiU and AMD for their valuable feedback. I would like to thank: Matthäus, Jason, Ingemar, and Dominik, for their help and support during my thesis at LiU, on this same subject.

The authors would like to thank the anonymous reviewers at EGSR for their great feedback on this paper. They were much appreciated!

References

[AFFC16] Andersen, Tobias Grønbeck, Falster, Viggo, Frisvad, Jeppe Revall, and Christensen, Niels Jørgen. "Hybrid Fur Rendering: Combining Volumetric Fur with Explicit Hair Strands". *The Visual Computer* 32.6-8 (2016), 739–749 2, 5.

[BCL*07] Bavoil, Louis, Callahan, Steven P, Lefohn, Aaron, et al. "Multi-Fragment Effects on the GPU using the K-Buffer". *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*. ACM. 2007, 97–104 3.

[CMS98] Cignoni, Paolo, Montani, Claudio, and Scopigno, Roberto. "A Comparison of Mesh Simplification Algorithms". *Computers & Graphics* 22.1 (1998), 37–54 1.

[dFH*11] d'Eon, Eugene, Francois, Guillaume, Hill, Martin, et al. "An Energy-Conserving Hair Reflectance Model". *Computer Graphics Forum*. Vol. 30. 4. Wiley Online Library. 2011, 1181–1187 3.

[Han14] Han, Dongsoo. "Hair Simulation in TressFX". *GPU Pro* 5 (2014), 407–418 1, 2.

[HDCD15] Heitz, Eric, Dupuy, Jonathan, Crassin, Cyril, and Dachsbacher, Carsten. "The SGGX Microflake Distribution". *ACM Transactions on Graphics (TOG)* 34.4 (2015), 48 2.

[HKSB06] Hadwiger, Markus, Kratz, Andrea, Sigg, Christian, and Bühler, Katja. "GPU-Accelerated Deep Shadow Maps for Direct Volume Rendering". *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. ACM. 2006, 49–52 4.

[HLY10] Hernell, Frida, Ljung, Patric, and Ynnerman, Anders. "Local Ambient Occlusion in Direct Volume Rendering". *IEEE Transactions on Visualization and Computer Graphics* 16.4 (2010), 548–559 1, 5.

[KK89] Kajiya, James T and Kay, Timothy L. "Rendering Fur with Three Dimensional Textures". *ACM SIGGRAPH Computer Graphics*. Vol. 23. 3. ACM. 1989, 271–280 3, 4.

[KRW18] Kanzler, Mathias, Rautenhaus, Marc, and Westermann, Rüdiger. "A Voxel-Based Rendering Pipeline for Large 3D Line Sets". *IEEE transactions on visualization and computer graphics* (2018) 5, 7.

[Lac13] Lacroix, Jason. "A Survivor Reborn: Tomb Raider on DX11". 2013 2, 4, 5.

[LN17] Loubet, Guillaume and Neyret, Fabrice. "Hybrid Mesh-Volume LoDs for All-Scale Pre-Filtering of Complex 3D Assets". *Computer Graphics Forum*. Vol. 36. 2. Wiley Online Library. 2017, 431–442 2.

[LTT91] LeBlanc, Andre M, Turner, Russell, and Thalmann, Daniel. "Rendering Hair using Pixel Blending and Shadow Buffers". *The Journal of Visualization and Computer Animation* 2.3 (1991), 92–97 2.

[LV00] Lokovic, Tom and Veach, Eric. "Deep Shadow Maps". *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. ACM Press/Addison-Wesley Publishing Co. 2000, 385–392 5.

[MCTB11] Maule, Marilena, Comba, João LD, Torchelsen, Rafael P, and Bastos, Rui. "A Survey of Raster-Based Transparency Techniques". *Computers & Graphics* 35.6 (2011), 1023–1034 3.

[MET*14] Martin, Timothy, Engel, Wolfgang, Thibieroz, Nicolas, et al. "TressFX: Advanced Real-Time Hair Rendering". *GPU Pro* 5 (2014), 193–209 1–3, 5.

[MJC*03] Marschner, Stephen R, Jensen, Henrik Wann, Cammarano, Mike, et al. "Light Scattering from Human Hair Fibers". *ACM Transactions on Graphics (TOG)*. Vol. 22. 3. ACM. 2003, 780–791 3.

[MWM08] Moon, Jonathan T, Walter, Bruce, and Marschner, Steve. "Efficient Multiple Scattering in Hair Using Spherical Harmonics". *ACM Transactions on Graphics (TOG)*. Vol. 27. 3. ACM. 2008, 31 2.

[ND05] Nguyen, Hubert and Donnelly, William. "Hair Animation and Rendering in the Nalu Demo". *GPU Gems* 2 (2005), 361–380 1, 7.

[Per12] Persson, Emil. "Geometric Antialiasing Methods". *GPU Pro* 3 (2012), 71–88 3.

[PHA05] Petrovic, Lena, Henne, Mark, and Anderson, John. "Volumetric Methods for Simulation and Rendering of Hair". *Pixar Animation Studios* 2.4 (2005) 2.

[Ric14] Riccio, Christophe. *How Bad are Small Triangles on the GPU and Why?* [Online; accessed 08-April-2019]. 2014. URL: `https://www.g-truc.net/post-0662.html` 1.

[RZL*10] Ren, Zhong, Zhou, Kun, Li, Tengfei, et al. "Interactive Hair Rendering under Environment Lighting". *ACM Transactions on Graphics (TOG)*. Vol. 29. 4. ACM. 2010, 55 2, 5.

[SA08] Sintorn, Erik and Assarsson, Ulf. "Real-Time Approximate Sorting for Self shadowing and Transparency in Hair Rendering". *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*. ACM. 2008, 157–162 2.

[Sch04] Scheuermann, Thorsten. "Practical Real-Time Hair Rendering and Shading". *ACM SIGGRAPH 2004 Sketches*. ACM. 2004, 147 3.

[SD15] Steward, Jason and Doyon, Uriel. "Augmented Hair in Deus Ex Universe Projects: TressFX 3.0". 2015 1, 2, 5.

[WBK*07] Ward, Kelly, Bertails, Florence, Kim, Tae-Yong, et al. "A Survey on Hair Modeling: Styling, Simulation, and Rendering". *IEEE Transactions on Visualization and Computer Graphics* 13.2 (2007), 213–234 1–5.

[WLJ*03] Ward, Kelly, Lin, Ming C, Joohi, Lee, et al. "Modeling Hair using Level-of-Detail Representations". *Proceedings 11th IEEE International Workshop on Program Comprehension*. IEEE. 2003, 41–47 2.

[XDY*12] Xing, Xiaoxiong, Dobashi, Yoshinori, Yamamoto, Tsuyoshi, et al. "Real-Time Rendering of Animated Hair Under Dynamic, Low-Frequency Environmental Lighting". *Proceedings of the 11th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and its Applications in Industry*. ACM. 2012, 43–46 2.

[XDY*15] Xing, Xiaoxiong, Dobashi, Yoshinori, Yamamoto, Tsuyoshi, et al. "Efficient Hair Rendering under Dynamic, Low-Frequency Environmental Light Using Spherical Harmonics". *IEICE Transactions on Information and Systems* 98.2 (2015), 404–411 2.

[YHGT10] Yang, Jason C, Hensley, Justin, Grün, Holger, and Thibieroz, Nicolas. "Real-Time Concurrent Linked List Construction on the GPU". *Computer Graphics Forum*. Vol. 29. 4. Wiley Online Library. 2010, 1297–1304 3.

[YK08] Yuksel, Cem and Keyser, John. "Deep Opacity Maps". *Computer Graphics Forum*. Vol. 27. 2. Wiley Online Library. 2008, 675–680 5.

[YT10] Yuksel, Cem and Tariq, Sarah. "Advanced Techniques in Real-Time Hair Rendering and Simulation". *ACM SIGGRAPH 2010 Courses*. ACM. 2010, 1 1–3.

[YYH*12] Yu, Xuan, Yang, Jason C, Hensley, Justin, et al. "A Framework for Rendering Complex Scattering Effects on Hair". *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM. 2012, 111–118 2, 3, 5.