The Visual Computer - CGI2016
The final publication is available at Springer via http://dx.doi.org/10.1007/s00371-016-1252-x

Hybrid fur rendering: combining volumetric fur with explicit hair strands

Tobias Grønbeck Andersen* · **Viggo Falster*** · **Jeppe Revall Frisvad** · **Niels Jørgen Christensen**

Abstract Hair is typically modeled and rendered using either explicitly defined hair strand geometry or a volume texture of hair densities. Taken each on their own, these two hair representations have difficulties in the case of animal fur as it consists of very dense and thin undercoat hairs in combination with coarse guard hairs. Explicit hair strand geometry is not well-suited for the undercoat hairs, while volume textures are not well-suited for the guard hairs. To efficiently model and render both guard hairs and undercoat hairs, we present a hybrid technique that combines rasterization of explicitly defined guard hairs with ray marching of a prismatic shell volume with dynamic resolution. The latter is the key to practical combination of the two techniques, and it also enables a high degree of detail in the undercoat. We demonstrate that our hybrid technique creates a more detailed and soft fur appearance as compared with renderings that only use explicitly defined hair strands. Finally, our rasterization approach is based on order-independent transparency and renders high-quality fur images in seconds.

Keywords Fur · Hair strand · Rasterization · Ray marching · Photorealistic rendering · Order-independent transparency · Shell volume

1 Introduction

Modeling and rendering of fur is essential in a number of applications within the entertainment and visual effects industry. In addition, digitally created fur is useful within the cosmetic and clothing industry as a tool for designing new products. This work aims at fur visualization for applications such as virtual clothing, where the goal is to come close to the appearance of real fur while retaining an aspect of interactivity in the rendering. It is a great challenge to semi-interactively generate and render fur which is qualitatively comparable to real fur. The challenge arises from the considerable number of hair strands present in fur. Mink fur like the dressed pelt in Fig. 1 (left) has around 20 thousand hair strands per square centimeter [13].

There are two standard ways to model fur: explicit models, where every hair strand is represented geometrically; and implicit models, where hair strands are represented by volume densities. Explicit models excel at creating visually distinguishable hair strands, but are often prone to aliasing artifacts due to the thinness of individual strands [3]. For dense animal furs, the aliasing artifacts become prohibitively processing intensive to deal with. In contrast, implicit methods excel at representing dense furs by treating the fur as a participating medium with scattering properties based on the hair density. Implicit models, however, lack the ability to represent an individual hair strand so that it is visually discernible from other nearby hair strands [11]. With high-resolution volumes, implicit models can represent individual hair strands [8], but this becomes too processing and memory intensive for our application. As exemplified in Fig. 1, we present a hybrid technique that enables simultaneous rendering of explicitly and implicitly modeled hair.

In Fig. 2, photos of a brown mink fur underline the visual importance of undercoat and guard hairs: thick, long guard hairs protrude from a fine, soft layer of undercoat fur (right). The guard hairs exhibit a shiny appearance as opposed to the diffuse undercoat hairs. The

* Joint primary authors.

T. G. Andersen · V. Falster · J. R. Frisvad (✉) · N. J. Christensen
Technical University of Denmark, Kgs. Lyngby, Denmark
E-mail: jerf@dtu.dk

Reference photo of a brown mink fur | Explicit hair strands | Hybrid fur (strands and volume)

Fig. 1 Most animal furs consist of both guard hairs and undercoat (*left*, as an example). These two fur layers are very different and hard to model and render believably with explicit hairs only (*middle*) or volumetric fur only. We present a pipeline for hybrid fur rendering that enables efficient combination of explicit hair strands with volumetric fur (*right*).

Fig. 2 Close-up photos of a brown mink fur. The fur skin is bent to better illustrate the visual differences of the undercoat hairs and the guard hairs. Both fur layers should be modeled to qualitatively match the appearance of real fur.

undercoat hairs have a tendency to "clump" together in cone-like structures (left), which resemble the appearance of a smooth noise function. As in previous work [11,4], we find that both undercoat and guard hairs have a significant impact on the overall appearance of the fur. Kajiya and Kay [11] render both fur layers in the ray marching pipeline used for volumetric fur, but cannot clearly produce individual hair strands. Bruderlin [4] renders both fur layers using explicit hair strands and a micropolygon rasterization pipeline. However, the undercoat fur seems too sparse in comparison with real animal fur. In addition, it appears too dark when the general direction of the hair strands is close to being parallel with the viewing direction.

We model the guard hairs with explicit camera-facing geometry. For the undercoat, we extrude a shell volume from the original polygonal model and dynamically choose the subdivision level of each individual triangle (dynamic resolution). We facilitate ray marching of this shell volume by a new polygon neighbor data structure and an on-the-fly lookup functionality for associating a position in space with an element of a volumetric data set (a voxel). To combine the volumetrically modeled undercoat with geometrically modeled guard hairs, we present two techniques using order-independent transparency (OIT) [39] in a new way. These techniques enable us to blend the implicit and explicit fur in a physically plausible way while retaining an aspect of interactivity (high-quality frames render in seconds).

2 Prior work

Early examples of fur rendering were based on rasterization of an explicit polygonal fur model [5,22]. Such techniques easily lead to aliasing problems. To overcome these problems, Kajiya and Kay [11] place voxels on the surface of a parameterized model to form a volumetric fur shell. They render this using ray marching [12]. Perlin and Hoffert [31] also use a volumetric fur model rendered by ray marching, but they compute fur volume densities using noise-based procedural techniques. Using a rasterization-based approach and blurring techniques to include approximate multiple scattering, Kniss et al. [14] present an interactive version of this noise-based volumetric fur. Their way of generating volumetric fur densities is similar to ours. However, they use a full volume representation, where we use a shell volume, and they do not include explicitly modeled hair strands.

As an extension of Kajiya's and Kay's [11] shell volume, Neyret [29] shows how mipmapping is useful for multiscale representation of geometry stored in shell volumes. Neyret [30] also suggests adaptive resolution by compression of the mipmap to suppress unvarying information. This compressed mipmap corresponds to a sparse voxel octree. Heitz and Neyret [9] present efficient use of a sparse voxel octree that accurately represents tiny opaque geometry. Although this is a volume representation, it is more suitable for rendering voxelized explicit fur [8]. As opposed to this, we use our shell volume to represent an anisotropic participating

medium defined by hair densities and scattering properties. Using a shell as in earlier work [11,29] instead of a sparse volume encapsulating the entire scene, we more easily obtain the orientation of the hair medium. In addition, our use of triangular prisms instead of boxes enables us to build the shell volume for a triangle mesh in a straight forward way.

There are many ways to model explicit hair strands. LeBlanc et al. [15] model each hair strand as a tube (a curved cylinder). At render time, these hair primitives are broken up into line segments and rendered as camera-facing geometry. In this way, each line segment appears as a cylinder. Goldman [7] slightly modifies this concept by using tubes with variable radius such that the hair segments become truncated cones. We use these tubes with variable radius for the modeling of our explicit hair strands (Fig. 3(3)). Following the approach of Van Gelder and Wilhelms [35], we generate explicit fur on a triangle mesh by randomly placing hair strands according to density and bending them according to gravity. We use cubic Bézier curves to shape the hair strands [1], and we apply grooming based on texture maps [27].

Rendering of explicitly defined hair strands entails a number of problems in aliasing, shadowing, and lighting [15,41]. The large number of rendering primitives is also a challenge with respect to memory and processing efficiency. To address these problems, LeBlanc et al. [15] describe a rasterization pipeline specifically for rendering of explicit hair. In recent years, this pipeline has been implemented to run efficiently on modern hardware. Yuksel and Tariq [41] describe how to deal with the large number of rendering primitives using geometry and tessellation shaders. Yu et al. [40] use order-independent transparency [39] to depth sort fur fragments and accurately blend the contributions from the hair strands seen in a pixel. We use similar techniques to efficiently render explicit hair strands, but our pipeline is different as it includes rendering of implicit volumetric fur (Fig. 3).

The light scattering model used for scattering events in volumetric fur, or to shade individual hair strands, is important with respect to the realism of the rendered result. Kajiya and Kay [11] presented the first hair shading model and used it with volumetric fur. Their model also applies to polygonal fur [15], and it is often used due to its simplicity. Several authors suggest different improvements for the Kajiya–Kay model [10,2,26], whereas others consider hair microstructure to create more physically based shading models [20,42,34,6,37]. For our volumetric undercoat fur, we use the standard Kajiya–Kay model. For our explicit guard hair strands, we use the single scattering component of the artist-friendly shading model [34], and we add the diffuse component of the Kajiya–Kay model to approximate multiple scattering between the guard hairs. We could trade efficiency for accuracy by using a physically accurate model [37] both for explicit hair strands and as the phase function in the rendering of volumetric fur.

Offline techniques include multiple scattering to improve realism [42,23,24,34,6,32,37], but spend minutes per rendering. Interactive techniques do exist that include approximate multiple scattering effects [14,43,33,36]. Such a technique could be combined with ours. Again, it is a trade-off between efficiency and accuracy.

Lengyel [18] and Lengyel et al. [17] propose an approach where fur appearance is precomputed and stored in 2D textures referred to as shells and fins. This technique enables real-time frame rates and has been improved in different ways [38,16]. However, it can ultimately only produce the rendering quality delivered by the technique used for the precomputation. Our hybrid technique is useful as a fast technique for precomputing shells and fins in high quality.

3 Method

We model guard hairs *explicitly* with camera-facing geometry and undercoat hairs *implicitly* with a volumetric shell wrapped around a polygonal base mesh. The volumetric shell is created by extrusion of the base mesh along smoothed vertex normals, resulting in a number of triangular prism shaped volumetric elements (prismatic voxels) located between the base mesh and the shell. To enforce the strengths and limit the weaknesses of these two approaches, we combine them into a single-pass rasterization-based technique.

Fig. 3 provides an overview of our rendering pipeline, which has the following steps. (1) We distribute hair roots on a polygonal base mesh in a geometry shader using the method outlined by Van Gelder and Wilhelms [35]. However, we modify the distribution with a 2D texture in order to control local density variations. In addition, the geometry shader generates the prismatic shell geometry and associated meta-data, which is stored in global buffers. (2) We use a compute shader to generate the volumetric data set based on simplex noise [21]. (3) We generate the camera-facing geometry for the explicit hair strands in a tessellation stage, formed by a tessellation control shader and a tessellation evaluation shader. The tessellation control shader is executed once per hair strand *root*, and the tessellation evaluation shader is executed once per hair strand *segment*. Data relating to the hair strand geometry, such as strand vertices and strand indices, are stored

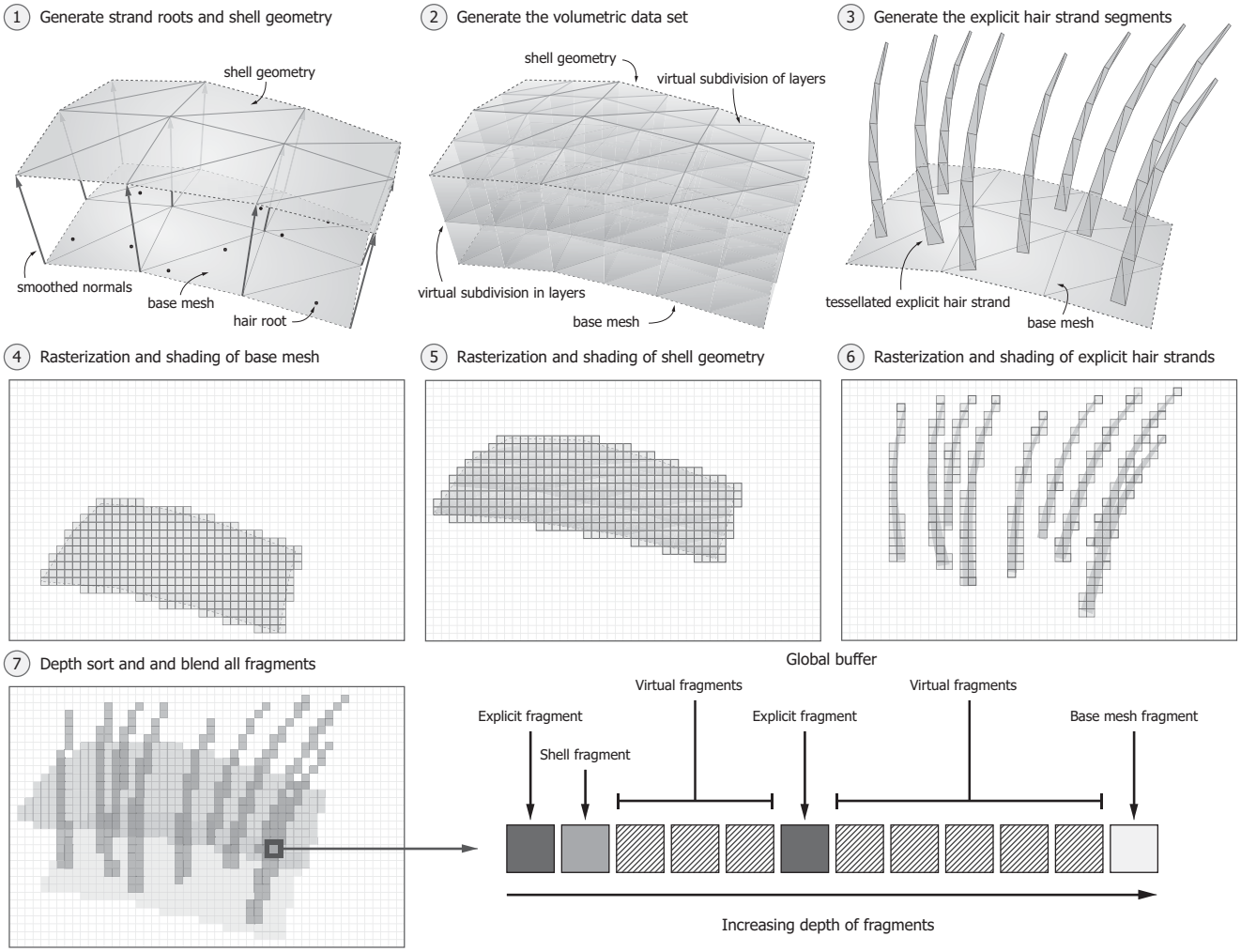

Fig. 3 Overview of our shader pipeline. In steps *1–3*, we generate explicit hair strand and shell geometry. In steps *4–6*, the generated geometry is rasterized and shaded (see also Fig. 4 and Section 3.1-3.2). In step *7*, the shaded fragments are depth sorted and blended in accordance with our blending algorithm (see also Fig. 5 and Section 3.3).

in global buffers. (4–6) We perform the actual rendering using three shader programs: one for the rasterized base mesh, one for the rasterized shell geometry (implicit fur), and one for the rasterized explicit fur. Finally, (7) we depth sort and blend the fragments.

3.1 Rendering of explicit hair strands

As described in Section 2, we use existing shading models to shade explicit hair strands. However, it is also important to consider self-shadowing [19]. We use a local, approximate self-shadowing model based on an exponentially increasing darkening of hair strands near the base mesh [28,25]. In addition, we let the transparency of the explicit hair strands increase exponentially towards the hair strand tips, resulting in a softer and more natural appearance of the fur.

3.2 Modeling and rendering of implicit hair strands

When rendering volumetric fur, we use the single scattering ray marching algorithm by Kajiya and Kay [11]. The algorithm approximately solves the volume rendering equation [12] by considering light which has been scattered towards the eye in a single scattering event. In the ray marching, we account for attenuation both from the light source to the scattering event and from the scattering event to the eye.

We initialize ray marching in the shader for the rasterized shell geometry (step 5). The shell fragments act as entry points to the volumetric shell (Fig. 4) and enable us to render the fur data stored in the prismatic shell volume. The three fragment shaders all access global buffers for shading calculations, and the processed fragment information (depth, color, and type) is output to another global buffer. This global buffer is

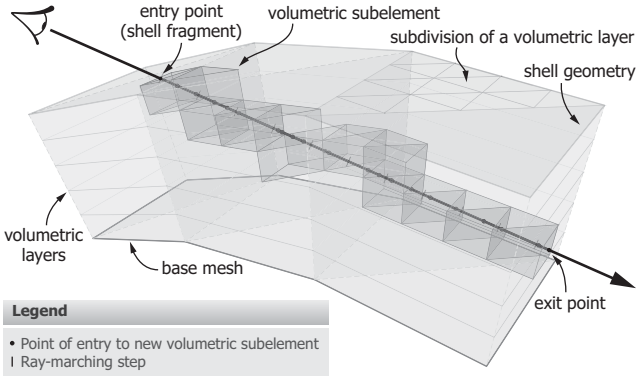

Fig. 4 The ray marching in step 5 of our shader pipeline.

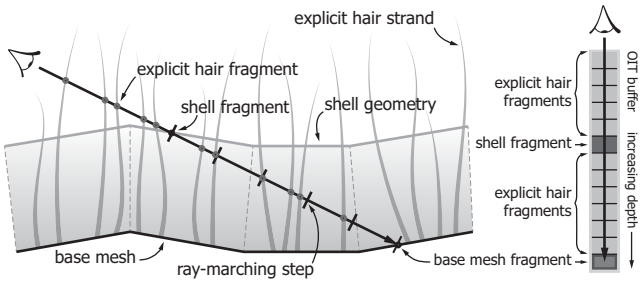

Fig. 5 The fragments generated by the three rasterization steps of our pipeline, seen from the side.

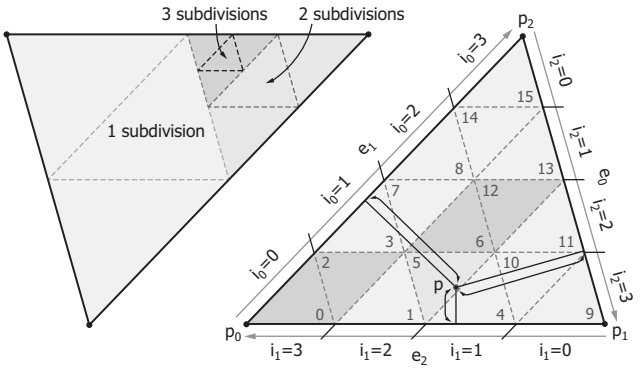

Fig. 6 Our indexing scheme for a volumetric layer.

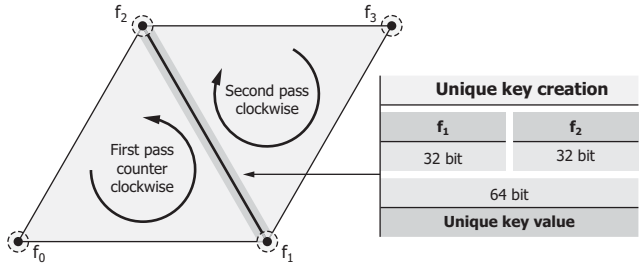

Fig. 7 Creation of unique keys for edges in order to build a neighbor data structure.

accessed in the final step (7) during the depth sorting and blending of the fragments (Fig. 5).

Generating the volumetric data set. To imitate the appearance of undercoat fur, we generate a 2D noise texture and apply it across all volumetric layers. We combine this with an increasing randomization of uv-coordinates as we approach the topmost layers of the shell volume. The randomization provides a more soft and fuzzy appearance. We also calculate tangent vectors by increasingly randomizing the interpolated, smoothed vertex normals towards the topmost volume layers.

Position to volumetric subelement. To associate a position in 3D space with an element of volumetric data, we *virtually* subdivide the prismatic voxels into finer volumetric *sub*-elements. We first slice the voxels into a number of layers defined by the perpendicular distance to the top of the voxel in question. We then subdivide each triangular layer into smaller triangles by repeatedly connecting edge centers (see Fig. 6, left).

Consider a layer of a given voxel (a triangle) defined by the points $\mathbf{p_0}$, $\mathbf{p_1}$ and $\mathbf{p_2}$, and its corresponding edges e_0, e_1 and e_2 (see Fig. 6, right). In order to calculate a local index of a given position $\mathbf{p}$ in this layer, we define three local sub-indices i_0, i_1 and i_2 by

$$i_x = \left\lfloor \left(1 - \frac{\mathrm{dist}_\perp(\mathbf{p}, e_x)}{\mathrm{dist}_\perp(\mathbf{p}_x, e_x)}\right) 2^d \right\rfloor , \tag{1}$$

where $x \in \{0, 1, 2\}$, d is the number of subdivisions along each edge and $\mathrm{dist}_\perp(\mathbf{p}, e)$ is the perpendicular distance from a point $\mathbf{p}$ to an edge e. Given the three local sub-indices, we now determine the local index i of a point $\mathbf{p}$ in a given layer as follows:

$$i = i_0(1 + i_0) + (i_1 - i_2)\,, \tag{2}$$

where the first term specifies the local index of the elements with $i_1 = i_2$ (highlighted in Fig. 6, right). Hence, it can be interpreted as an offset to which we need to add the (signed) difference between the remaining two local indices, $i_1 - i_2$. This results in a local index i for the subelement of a given layer to which $\mathbf{p}$ belongs. In combination with the voxel index i_v and the layer index i_l, the global index i_g is now given by

$$i_g = (i_v n_l + i_l)4^d + i\,, \tag{3}$$

where n_l is the number of layers in each voxel.

Neighbor data structure. To march through the voxels, we need the ability to go from one primitive in a triangle mesh to its neighboring primitives. This requires that we store the indices of the three neighboring faces with the vertex data of each face in the mesh. We obtain the indices of neighboring faces by first iterating through all faces and building a map that associates each edge with a face that it belongs to. An edge is uniquely identified by two 32-bit vertex indices. Hence, we can create a unique key for all edges by bit-shifting one of the two

vertex indices, and combining them into a single 64-bit integer. This is illustrated in Fig. 7. We then build the neighbor data structure by iterating over all primitives again, but now forming keys in reversed order. A lookup with a reversed key into the map that associates edges with faces provides the index of the neighboring face, which we then store with the vertex data of each face.

Ray marching in a fragment shader. Each shell fragment has access to the global index of the voxel to which it belongs. We step into the voxels along the direction of the eye ray. At each step, we try to find a local index within the current voxel. If such a local index exists, we combine it with the global index of the voxel in order to identify the volumetric subelement closest to the current step position. As we can identify the volumetric subelement, we can also associate each step position with an element of the volumetric data set. If the calculation of an index within the layer of the current step position fails, we are no longer within the bounds of the current voxel. In this case, we find the edge of the current layer that is closest to the current step position and refer to it as the exit edge. Following this, we attempt to find a local index within one of the three neighboring voxels. The relevant neighbor is given by our neighbor data structure in combination with the exit edge. This enables us to ray march through multiple voxels.

We continue this process iteratively until we find a valid index in a voxel, or until we exit the volumetric shell. Each time we find an index, we associate the step position with an element of the volumetric data set and do the standard computations of ray marching [11]: accumulate attenuation, ray march toward external light sources, evaluate the phase function, and calculate the radiance towards the eye.

Attenuation towards the eye is based on the density of the current and previously processed volumetric subelements. We apply the Kajiya–Kay phase function [11], which determines the fraction of incoming light scattered towards the eye at the current step. Finally, multiplication of the incoming light with density, accumulated attenuation, and phase function results in the radiance towards the eye from the current step.

3.3 Hybrid fur rendering

The explicit hair strands generate a vast number of fragments (many per strand), whereas the shell volume generates just a few fragments where we start ray marching into the underlying volume. One shell fragment represents *all* undercoat hair strands intersected by a given eye ray (Fig. 5). As a consequence, we cannot treat the shell fragments as other fragments. If we did so, all fragments *behind* a shell fragment would appear occluded by all the undercoat hairs it represents. Hence, the undercoat hairs would appear too dominant. Based on these thoughts, a proper combination of the two rendering schemes requires that the visualization of the volumetric data set and the explicit hair strands affect each other.

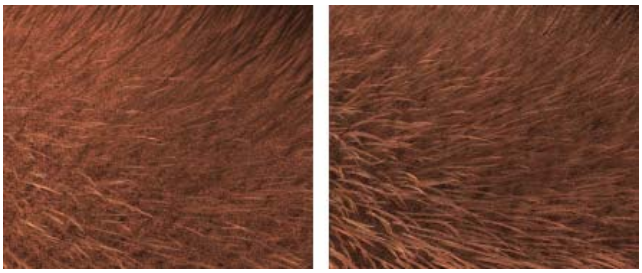

Fig. 8 Simplified blending (*left*) and blending with virtual fragments (*right*).

Virtual fragments. We achieve a physically plausible combination of the two schemes by storing multiple virtual fragments instead of only one per shell fragment. When ray marching through the volume, we compute the radiance towards the eye along the viewing ray. Instead of only calculating a final radiance result, we combine a small set of the ray marching steps into an intermediate result and store this as a virtual fragment. Additionally, each virtual fragment contains positional information so that it can be depth sorted accurately together with other non-virtual fragments (Fig. 3(7)). This accurate depth sorting produces visually pleasing results on close-up as demonstrated in Fig. 8. However, due to the added virtual fragments, the total number of fragments per pixel can easily exceed one hundred. This larger set of fragments requires additional sorting and more allocated memory, which becomes a limiting factor in oblique views where rays take a longer path through the shell volume.

Simplified blending. As a faster and less memory-intensive alternative to virtual fragments, we extend the fragment information to include the fragment type (in addition to color and depth). The fragment information describes whether a fragment stems from explicit hair geometry, the shell geometry, or an opaque object. We use this information in the per pixel post-processing step of our implementation of order-independent transparency (see Section 4), where we have access to all fragment information for all pixels.

With the fragment type information, we can reduce the alpha of a shell fragment based on the number of unoccluded explicit hair strand fragments located directly behind it and their depth (Fig. 5). If no such

explicit hair strand fragments exist, the shell fragment keeps its original alpha value as determined by our volume rendering algorithm. For each explicit hair strand fragment that we find, we subtract from the alpha:

$$\alpha_p = \exp(-a d_{pe}) \,, \tag{4}$$

where a is an attenuation parameter and d_{pe} is the distance from the shell fragment to the hair strand fragment. In this way, the explicit hair strands affect the visual impact of the shell fragments. This is reasonable as the attenuation accumulated during a ray marching will increase for each intersection of the viewing ray with an explicit hair strand. In addition, the effect is local as α_p decreases with increasing distance to the shell geometry. While this simplified blending is clearly an approximation, it enables us to avoid the memory and performance hit of virtual fragments. When we render a full mesh, the fur is not up close, and the results seem reasonable with this approximation, see Section 5. Another aspect is that simplified blending is more user friendly, as it provides direct control of the balance between the visual impact of each type of fur.

4 Implementation

Global buffer memory with custom layout is part of the OpenGL 4.3 core specification. The *Shader Storage Buffer Object* (SSBO) enables sharing of memory between multiple shader programs. We utilize this feature by storing all data generated on the GPU in SSBOs. In this way, we avoid any overhead of copying data between CPU and GPU, which is important in our pipeline as we distribute rendering tasks into different shader programs (Fig. 3).

To blend fragments ordered by depth, we store all generated fragments and use order-independent transparency (OIT) [39]. This means that we do not need a pre-sorting of scene geometry to have correct blending. Fragment information is stored in a pre-allocated global buffer, which is sliced into smaller pieces of varying sizes. Each slice is a linked list storing information such as fragment color and depth for all fragments of a given pixel. As a result, we have access to all fragments relating to each pixel of the final frame. We use the information in a per pixel post-processing step where all fragments are depth sorted (back to front) and blended with over-operations (see Section 3.3).

5 Results

Subdivision shell with neighbors. We first compare our ray marching method with a more conventional ray–voxel intersection technique. Both approaches produce the same results but with significant differences in performance. Ray marching with ray–prism intersections accelerated by a binary space partitioning (bsp) tree requires the time $t\,(3 + 0.4\,d)$ for processing, where t is the time required by our method and d is the number of subdivisions. The bsp tree stores index and bounding box for every subelement in the shell and a large number of splitting planes. The memory consumption of this acceleration structure thus grows exponentially with d. Our method is faster, does not depend on d, and requires no acceleration data structure.

Rendering a bended fur skin. In Fig. 1 (left), we show a reference photo of a cylindrically shaped brown mink fur skin shot in a light controlled environment. We created a polygonal model to roughly match the shape of the skin, and added fur rendered with our hybrid rendering technique, see Fig. 1 (right). We applied tone mapping and depth of field as a post-processing effect. If we only render explicitly modeled hair strands, the fur lacks depth and softness, as seen in Fig. 1 (middle). In comparison, our hybrid technique achieves an increase in softness and a higher level of detail in the fur. As a consequence, the qualitative appearance of the hybrid fur is in our opinion closer to the qualitative appearance of the real fur.

Since our rendering technique is deterministic and performs a complete depth sorting of all fragments, our results do not suffer from temporal instability as can be observed in the supplementary video. The fur in the video differs slightly in appearance from that of Fig. 1 as we did not apply the post-processing effects.

Rendering a fur hat. Fig. 9 shows implicit, explicit, and hybrid fur applied to a polygonal hat model. As for the fur skin shown in Fig. 1, the addition of implicitly rendered undercoat hairs qualitatively enhances the sense of depth in the rendered fur.

Fig. 10 is an example of how the fur hat can be integrated into a live action background image in a qualitatively realistic fashion. We applied depth of field and tone mapping as a post-processing effect. In the bottom row of Fig. 10, we show an environment map captured in the location where the background image was shot. We created the environment map by unwrapping an HDR photograph of a mirror ball with HDR-Shop 3.0. We also used HDR-Shop to sample 16 directional light sources (similar to the approach described by Hiebert et al. [10]), which we used to simulate the effect of environment lighting of the *explicit* hair strands. We applied this environment lighting in all our results. We did not use environment lighting for the shading of the implicit fur, as this environment lighting has a relatively small visual effect on the implicit fur.

Fig. 9 Renderings of a brown mink fur hat illuminated by the HDR environment map shown in Fig. 10: **a** implicitly rendered fur, **b** explicitly rendered fur, **c** hybrid rendered fur.

Fig. 10 *Top*: brown mink fur hat integrated into a live-action background image. *Bottom left*: chrome ball image of an environment. *Bottom right*: unwrapped environment map with extracted light sources indicated as *colored dots*.

	Cylinder	**Hat**	**Bunny**
Triangle count	3584	5632	69451
1. Mesh and textures, GB	0.37	0.33	0.37
2. Prism volume densities, GB	1.06	0.77	0.59
3. Strand geometry, GB	0.06	0.52	0.28
4. Shell fragments, GB	0.20	0.20	0.20
5. Base mesh fragments, GB	0.20	0.20	0.20
6. Explicit fragments, GB	2.60	2.60	2.60
Total memory, GB	4.49	4.62	4.24
1. Process base mesh, ms	0.2	1.7	1.4
2. Prism volume densities, ms	2405	1740	1327
3. Generate expl. strands, ms	1.1	8.0	4.3
4. Rasterize shell volume, ms	635	519	755
5. Rasterize base mesh, ms	87.6	139	147
6. Rasterize explicit fur, ms	428	1270	795
7. OIT post-processing, ms	255	812	447
Preprocess time (1–3), ms	2406	1750	1333
Online render time (4–7), ms	1407	2740	2143
Total render time, ms	3813	4490	3476

Table 1 Memory consumption and timings for each step of our pipeline when rendering a single frame of one of the images in this paper with resolution 3840×2274.

Rendering a furry bunny. Fig. 11 shows explicit and hybrid fur applied to the Stanford bunny (69,451 triangles). This demonstrates that our technique also applies to larger meshes.

Rendering performance. We used an NVIDIA Quadro K6000 GPU with 12 GB memory for rendering. Table 1 presents memory consumption and timings for each step of our hybrid fur rendering pipeline. As we used 4× supersampling, the *online render time* should be multiplied by four. The rendering resolution is 3840× 2274, which we downsample to get a high-quality fur image of resolution 1920×1137. These settings were used for all the rendered images that we present. Their total rendering time is thus well below 13 seconds for all meshes. In previsualization, the user would rarely need a high-resolution image. We therefore also generated an image of lower resolution: 512×512 after downsampling from 1024×1024. At this resolution, a single frame of the cylinder mesh can be computed in a total of 489 ms (not including preprocessing).

Using virtual fragments (as in Fig. 8), the number of fragments generated in step 4 increases from one to x per pixel on average. The OIT post-processing time increases with x, and the limit is $x = 30$ in our examples, as the memory consumption of step 4 then becomes 7.8 GB. Further increasing x requires a better GPU or a trade-off such as lowering the resolution or computing densities on the fly. Quality-wise virtual fragments carry a great potential, but they also require the user to carefully consider the available GPU resources. A similar memory and performance hit applies if we increase the number of explicit hair strands. Using explicit fur only, we could trade the 635 ms spent on ray march-

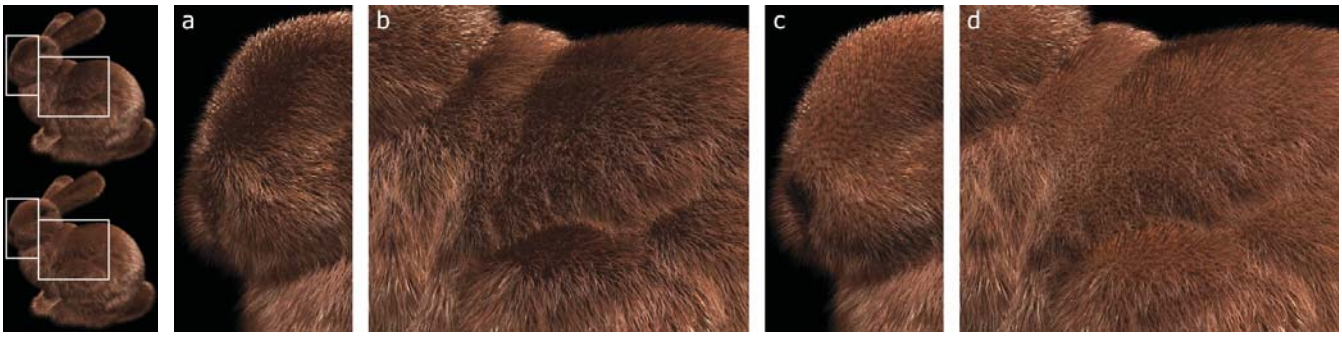

Fig. 11 Comparison of explicit fur and hybrid fur. Magnified nose and back of the furry Stanford bunny seen in full in the *leftmost column*: **a**, **b** explicitly rendered fur; **c**, **d** hybrid rendered fur.

ing in Fig. 1 for 82.432 undercoat hair strands. This is however far from the 3.8 million hair strands in the undercoat of the real mink fur. The use of volumetric fur is thus important as here each scattering event represents the net effect of scattering by many hair strands.

6 Discussion and conclusion

We combine geometrically modeled guard hairs with volumetrically modeled undercoat hairs. The result is a new fur rendering technique for accurate modeling and rendering of the two layers present in nearly all animal furs. With this hybrid technique, we are able to render images that in our opinion are a good approximation of the qualitative appearance of real animal fur. The explicit representation of the guard hairs enables us to imitate the visually distinguishable single strands of hair. At the same time, the volumetric component of our solution enables us to imitate the very dense undercoat. We thus overcome the problems related to an explicit representation of dense furs as well as the problems related to visualization of distinguishable hair strands with implicit approaches.

Our implementation requires less than a second to render previews and less than a quarter of a minute to generate fur and render it in high quality. Thus, we believe that our technique retains an aspect of interactivity that makes it suitable for virtual clothing or fashion CAD systems. As we can regenerate the fur from an arbitrary mesh and also render it in only seconds, our technique fully supports fur animation. This includes the ability to animate both undercoat and guard hairs.

There are many ways to improve our results. First of all, we believe that virtual fragments have an interesting potential as they lead to more accurate evaluation of the volume rendering equation. Other important ways to improve our work is by more accurate self-shadowing, more physically accurate hair reflectance and phase function, and inclusion of multiple scattering. This could further improve the qualitative similarities of reference photos and rendered images.

Our prismatic shell volume with dynamic resolution has many potential applications beyond fur rendering. We especially believe that it is useful in multiscale modeling of bark, dust, dirt, fibers, or other phenomena that stick to surfaces or grow on surfaces.

In conclusion, we presented a hybrid fur rendering technique that forms a solid foundation for improving the quality of rendered fur in applications that require an aspect of interactivity. In particular, we believe that our technique is an important step toward more widespread use of fur in virtual clothing, fashion CAD, and digital prototyping.

Acknowledgements We would like to thank the creative workshop Kopenhagen Studio at Kopenhagen Fur (http://-www.kopenhagenfur.com/kick/kopenhagen-studio) for kindly loaning us the mink fur sample appearing in the reference photos. The Stanford Bunny model is courtesy of the Stanford University Computer Graphics Laboratory (http://graphics-.stanford.edu/data/3Dscanrep/).

References

1. Ando, M., Morishima, S.: Expression and motion control of hair using fast collision detection methods. In: Image Analysis Applications and Computer Graphics, *Lecture Notes in Computer Science*, vol. 1024, pp. 463–470. Springer (1995)
2. Angelidis, A., McCane, B.: Fur simulation with spring continuum. The Visual Computer **25**(3), 255–265 (2009)
3. Barringer, R., Gribel, C.J., Akenine-Möller, T.: High-quality curve rendering using line sampled visibility. ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2012) **31**(6), 162:1–162:10 (2012)
4. Bruderlin, A.: A basic hair/fur pipeline. In: N. Magnenat-Thalmann (ed.) Photorealistic Hair Modeling, Animation, and Rendering, no. 34 in ACM SIGGRAPH 2003 Course Notes. ACM (2003)
5. Csuri, C., Hackathorn, R., Parent, R., Carlson, W., Howard, M.: Towards an interactive high visual complexity animation system. Computer Graphics (Proceedings of SIGGRAPH 79) **13**(2), 289–299 (1979)

6. d'Eon, E., Francois, G., Hill, M., Letteri, J., Aubry, J.M.: An energy-conserving hair reflectance model. Computer Graphics Forum (Proceedings of EGSR 2011) **30**(4), 1181–1187 (2011)
7. Goldman, D.B.: Fake fur rendering. In: Proceedings of SIGGRAPH 1997, pp. 127–134. ACM Press/Addison-Wesley (1997)
8. Heitz, E., Dupuy, J., Crassin, C., Dachsbacher, C.: The SGGX microflake distribution. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2015) **34**(4), 48:1–48:11 (2015)
9. Heitz, E., Neyret, F.: Representing appearance and pre-filtering subpixel data in sparse voxel octrees. In: Proceedings of High-Performance Graphics (HPG'12), pp. 125–134 (2012)
10. Hiebert, B., Dave, J., Kim, T.Y., Neulander, I., Rijpkema, H., Telford, W.: The Chronicles of Narnia: the lion, the crowds and rhythm and hues. In: ACM SIGGRAPH 2006 Courses, Article 1. ACM (2006)
11. Kajiya, J.T., Kay, T.L.: Rendering fur with three dimensional textures. Comptuter Graphics (Proceedings of SIGGRAPH 89) **23**(3), 271–280 (1989)
12. Kajiya, J.T., Von Herzen, B.P.: Ray tracing volume densities. Computer Graphics (Proceedings of SIGGRAPH 84) **18**(3), 165–174 (1984)
13. Kaszowski, S., Rust, C.C., Shackelford, R.M.: Determination of the hair density in the mink. Journal of Mammology **51**(1), 27–34 (1970)
14. Kniss, J., Premože, S., Hansen, C., Ebert, D.: Interactive translucent volume rendering and procedural modeling. In: Proceedings of IEEE Visualization 2002, pp. 109–116 (2002)
15. LeBlanc, A.M., Turner, R., Thalmann, D.: Rendering hair using pixel blending and shadow buffers. The Journal of Visualization and Computer Animation **2**(3), 92–97 (1991)
16. Lee, J., Kim, D., Kim, H., Henzel, C., Kim, J.I., Lim, M.: Real-time fur simulation and rendering. Computer Animation and Virtual Worlds (Proceedings of CASA 2010) **21**(3–4), 311–320 (2010)
17. Lengyel, J., Praun, E., Finkelstein, A., Hoppe, H.: Real-time fur over arbitrary surfaces. In: Proceedings of Symposium on Interactive 3D Graphics (i3D 2001), pp. 227–232. ACM (2001)
18. Lengyel, J.E.: Real-time fur. In: Rendering Techniques 2000 (Proceedings of EGWR 2000), pp. 243–256. Springer (2000)
19. Lokovic, T., Veach, E.: Deep shadow maps. In: Proceedings of SIGGRAPH 2000, pp. 385–392. ACM Press/Addison-Wesley (2000)
20. Marschner, S.R., Jensen, H.W., Cammarano, M., Worley, S., Hanrahan, P.: Light scattering from human hair fibers. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003) **22**(3), 780–791 (2003)
21. McEwan, I., Sheets, D., Richardson, M., Gustavson, S.: Efficient computational noise in GLSL. Journal of Graphics Tools **16**(2), 85–94 (2012)
22. Miller, G.S.P.: From wire-frames to furry animals. In: Proceedings of Graphics Interface (GI '88), pp. 138–145 (1988)
23. Moon, J.T., Marschner, S.R.: Simulating multiple scattering in hair using a photon mapping approach. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006) **25**(3), 1067–1074 (2006)
24. Moon, J.T., Walter, B., Marschner, S.: Efficient multiple scattering in hair using spherical harmonics. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008) **27**(3), 31:1–31:7 (2008)
25. Neulander, I.: Quick image-based lighting of hair. In: ACM SIGGRAPH 2004 Sketches, p. 43. ACM (2004)
26. Neulander, I.: Fast furry ray gathering. In: ACM SIGGRAPH 2010 Talks, Article 2. ACM (2010)
27. Neulander, I., Huang, P., Rijpkema, H.: Grooming and rendering cats and dogs. In: ACM SIGGRAPH 2001 Sketches, p. 190. ACM (2001)
28. Neulander, I., van de Panne, M.: Rendering generalized cylinders with paintstrokes. In: Proceedings of Graphics Interface (GI '98), pp. 233–242 (1998)
29. Neyret, F.: A general and multiscale model for volumetric textures. In: Proceedings of Graphics Interface (GI '95), pp. 83–91 (1995)
30. Neyret, F.: Synthesizing verdant landscapes using volumetric textures. In: Rendering Techniques '96 (Proceedings of EGWR 1996), pp. 215–224. Springer (1996)
31. Perlin, K., Hoffert, E.M.: Hypertexture. Computer Graphics (Proceedings of SIGGRAPH 89) **23**(3), 253–262 (1989)
32. Qin, H., Chai, M., Hou, Q., Ren, Z., Zhou, K.: Cone tracing for furry object rendering. IEEE Transactions on Visualization and Computer Graphics **20**(8), 1178–1188 (2014)
33. Ren, Z., Zhou, K., Li, T., Hua, W., Guo, B.: Interactive hair rendering under environment lighting. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010) **29**(4), 55:1–55:8 (2010)
34. Sadeghi, I., Pritchett, H., Jensen, H.W., Tamstorf, R.: An artist friendly hair shading system. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010) **29**(4:1–4:10), 56 (2010)
35. Van Gelder, A., Wilhelms, J.: An interactive fur modeling technique. In: Proceedings of Graphics Interface (GI '97), pp. 181–188 (1997)
36. Xu, K., Ma, L.Q., Ren, B., Wang, R., Hu, S.M.: Interactive hair rendering and appearance editing under environment lighting. ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2011) **30**(6), 173:1–173:10 (2011)
37. Yan, L.Q., Tseng, C.W., Jensen, H.W., Ramamoorthi, R.: Physically-accurate fur reflectance: Modeling, measurement and rendering. ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2015) **34**(6), 185:1–185:13 (2015)
38. Yang, G., Sun, H., Wu, E., Wang, L.: Interactive fur shaping and rendering using nonuniform-layered textures. IEEE Computer Graphics and Applications **28**(4), 24–32 (2008)
39. Yang, J.C., Hensley, J., Grün, H., Thibieroz, N.: Real-time concurrent linked list construction on the GPU. Computer Graphics Forum (Proceedings of EGSR 2010) **29**(4), 1297–1304 (2010)
40. Yu, X., Yang, J.C., Hensley, J., Harada, T., Yu, J.: A framework for rendering complex scattering effects on hair. In: Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (i3D 2012), pp. 111–118. ACM (2012)
41. Yuksel, C., Tariq, S.: Advanced techniques for real-time hair rendering and simulation. In: ACM SIGGRAPH 2010 Courses, Article 1. ACM (2010)
42. Zinke, A., Sobottka, G., Weber, A.: Photo-realistic rendering of blond hair. In: Proceedings of Vision, Modeling, and Visualization (VMV 2004), pp. 191–198 (2004)
43. Zinke, A., Yuksel, C., Weber, A., Keyser, J.: Dual scattering approximation for fast multiple scattering in hair. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008) **27**(3), 32:1–32:10 (2008)