WHITE PAPER

UNREAL ENGINE

GROOMING FOR REAL-TIME REALISM: HAIR AND FUR WITH UNREAL ENGINE

TABLE OF CONTENTS

Introduction

A head of hair and a pelt of fur are complex organic structures made up of thousands of colored strands with specific characteristics. The particular shine of hair or fur, which we instantly recognize as such, is the result of the strands' unique translucent and reflective properties, including their tendency to clump and curl together. These strands also react to forces like gravity or wind in ways that we, as lifelong observers of hair and fur, instinctively recognize as natural.

The production of realistic hair and fur for computer-generated imagery has been a topic of research and study for several decades, not only for animated characters in games and films, but also for training simulations for military, industry, and animal science, among other fields.

Figure 1: Guide hairs in DCC application and final render in Unreal Engine

The challenge of creating computer-generated (CG) hair and fur lies in fooling the human eye. However, due to the limitations of computer processing power, a CG artist traditionally could not see the CG hair's final colors or shadows in context in the scene, nor the hair's reactions to light and movement, until after a time-consuming rendering process. After rendering, the artist, seeing room for improvement, would have to tweak the settings and render again. Thus, the process of getting CG hair and fur to look just right has traditionally been long and arduous.

For this reason, Epic Games recently introduced the ability to render hair and fur in real time in Unreal Engine, giving the artist the ability to see the final results in action, on screen, without a long wait for rendering.

Why this paper?

While real-time rendering presents a welcome relief from long render times normally associated with hair and fur, the setup in Unreal Engine requires careful consideration of a number of parameters to work efficiently, and to produce the most realistic hair or fur possible.

The purpose of this paper is to lay out the process of working with real-time hair and fur in Unreal Engine, going beyond the documentation to discuss artistic and technical concerns that artists might encounter while using this feature.

Hair and fur in DCC applications

Hair and fur for Unreal Engine is set up in a digital content creation (DCC) application such as Maya, 3ds Max, or Houdini. The hair or fur is attached to a polygonal surface, such as a model of a human head or an animal.

Common tools for generating hair and fur include DCC plugins such as **XGen**, **Yeti**, and **Ornatrix**, and the Hair Utils tool included with **Houdini**. In addition to hair and fur, these tools can also be used to generate feathers, such as those on a bird.

After the hair/fur setup is complete in the DCC application, it can then be imported to Unreal Engine for real-time rendering.

Terminology

To introduce the methodology and considerations for DCC applications and the extended toolset in Unreal Engine, here we will do a brief review of the terms and processes involved in creating CG hair and fur.

Guide hair, guide curve - A spline primitive that represents a grouping of hair or fur strands. The hair/fur tool includes a mechanism for manipulating the guide hairs, which in turn determine how the final hairs (interpolated hairs) behave—the interpolated hairs follow the shape of the nearest guides based on a weighted average of the distance between them. The lengths of the interpolated hairs are usually determined by the lengths of the guides.

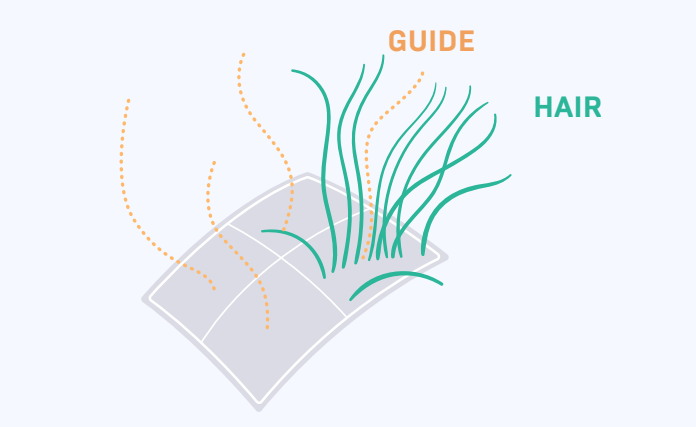


Figure 2: Relationship of guides and hairs

Control vertices (CVs) - One or more control vertices are placed along the guide curve, allowing for control at that point on the curve. The higher the CV count on a guide hair, the more the guide hair can be molded into a precise shape. However, a higher CV count also increases processing time, which in turn reduces performance.

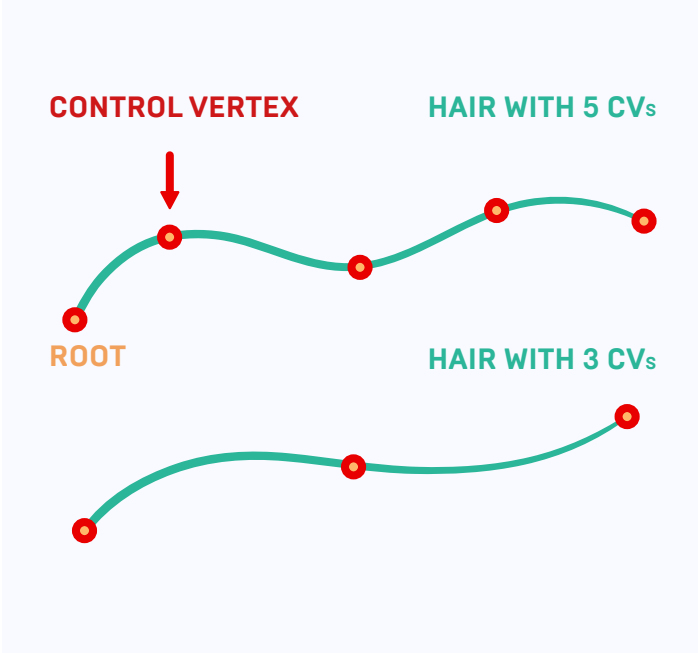


Figure 3: Control vertices on guide curves

Styling - The practice of pushing/pulling guide hairs or interpolated hairs in a particular direction, shaping them, curling them, etc. to make the CG hair or fur match reference images.

Grooming - A combination of defining hair growth and natural hair properties, and the styling process. This includes setting the thickness of strands and their tendency to curl or clump together. The resulting settings, and a file that holds them, is called a *groom*. Grooming is accomplished through the use of guide hairs and parametric settings in the DCC application.

Grooming workflow

The grooming workflow takes place in a DCC, before the model is imported to Unreal Engine. As a starting point, a set of guide hairs that determine the shape and the volume is created on a mesh area.

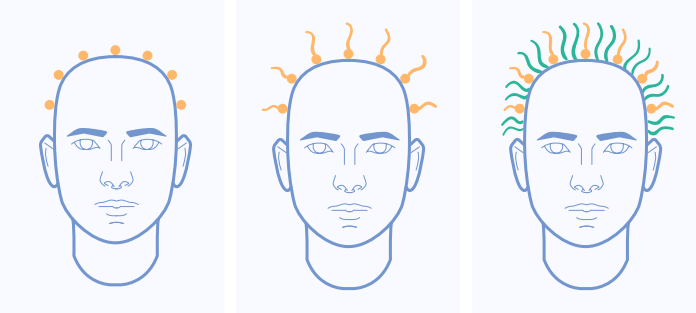


Figure 4: Guide hairs for hair on human head shape

Grooming brushes are used to position, sculpt, and scale groomable guide curves. Grooming brushes can curl, trim, or even remove hairs.

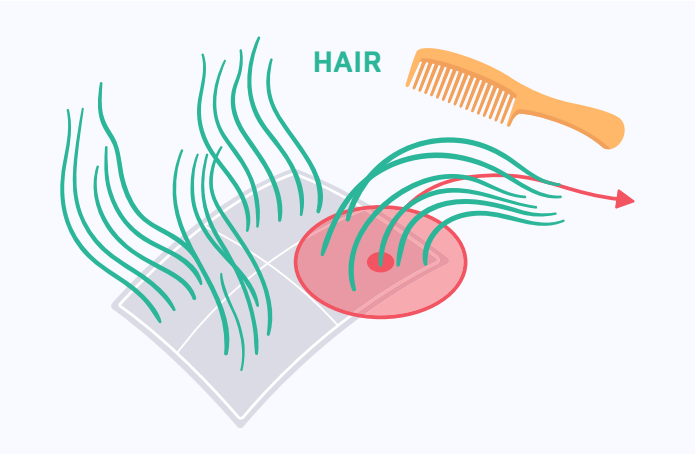


Figure 5: Virtual grooming brush

With the help of sculpting and combing brushes, the guides are shaped into the hairstyle or pelt style. Color for the rendered hair can be defined via material parameters or textures sampled onto the hair.

Since different areas of a character have distinct features, a groomer might create separate grooms for the head, eyebrows, beard, and body hair for a human, or for an animal's head, body, legs, and tail.

As a complex groom is created, splitting it up into different sections makes it easier to manage various groups once the asset is imported into Unreal Engine.

Procedural styling

Hair and fur tools within DCC applications include parameters for procedural styling, which adjust guide hairs or interpolated hairs. These parameters often have names that mimic real-life styling terms like *trim, curl, and frizz*. One or more numerical values are associated with each procedure to indicate the relative degree of alteration to the guide hair or strands.

A *trim* value will limit the length of guide hairs or interpolated hairs. A *curl* setting will ask for the number of curls and the radius, and will act accordingly. *Frizz*, or *noise*, will randomize the positions of the curve's CVs a little or a lot, depending on the number value entered.

For such parameters, the number of CVs directly affects how much detail the curves can support. Since complex shapes require a larger number of CVs, you might find it necessary to rebuild the curves from time to time to ensure that the CVs are distributed equally along the length of the hair.

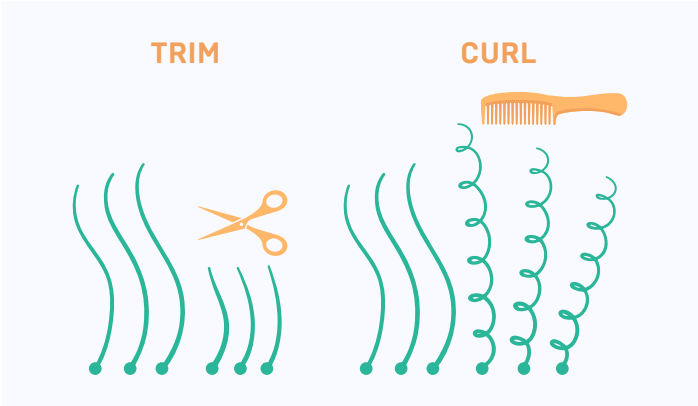


Figure 6: Procedural styling tools

Rendering an image with hair/fur requires the system to render multiple visible strands within each pixel. A human head has hundreds of thousands of hair strands—a reasonable strand count for human hairstyles in Unreal Engine is around 50k—each with an average diameter of about 100 micrometers. (For animal fur, the number of strands and average diameter varies widely from one animal to the next.)

An optimal groom sets the right balance between the density and thickness to balance realism with performance.

Clumping

Clumping is a phenomenon we see naturally occurring in hair and fur, such as on a lock of hair, an individual curl, or wet fur. The principal reason for adding levels of clumping is to bring detailing, variation, and breakup to the larger forms, to achieve a more realistic look.

In CG, the clumping algorithm creates the effect by pulling the tips of strands together during rendering.

Figure 7: Clumping pulls tips of strands together

To give the hair a breakup effect and natural structure, clumps work in a three-tiered process: large, medium, and small clumps. We also refer to them as primary, secondary, and tertiary clumps. Primary clumps are the main group of hair strands from which the secondary and tertiary are derived.

There is no real set of rules when using these extra levels of clumping—it just comes down to the reference that is being matched. Clumps in real life can be highly detailed, and achieving a look similar to the reference can be challenging. It's best to start with the larger clump structures first, then layer in sub-clumping structures until the overall structure matches.

Procedural versus manual approaches

A procedural approach is to use parameters (modifiers, expressions) to place hairs and to set the curl, length, etc. of one or more hairs. Conversely, a manual approach is to place hairs by hand, and sculpt individual hairs or small groups of hairs with a mouse-based brush to match reference.

A groom is generally built using a combination of these two methods; the choice of when to move from procedural to manual manipulation is up to the groomer. A simplistic set of guides can have its final state defined procedurally via a series of styling modifiers, while individual guides can be placed manually and intricately sculpted into a final shape.

Procedural approaches tend to lend themselves to shorter hair and fur. It is usually good to start out using as much of a procedural approach as possible to block in areas of a groom before starting to hand-place clumps and individual hairs.

If a character is far from the camera or is in a high-action sequence, a procedural approach usually works best. Procedural grooms, for the most part, have fewer layers or node networks than assets built using a manual approach.

A manual approach can be essential for hero assets that are seen up close, even if the process is considerably more time-consuming. With this technique, the groomer can hand-place hairs to match to specific detail to achieve the desired result. In visual effects for feature films, for example, groomers often move single eyelashes to match a live actor's appearance, or move individual fur strands to match to a very specific fur structure from reference.

Grooming best practices

References

Sourcing reference, and paying close attention to fine detail, is the best way to achieve the highest level of fidelity. Gathering high-resolution images ensures that close-up details such as fine hairs or subtle breakup are represented clearly. Looking for references that cover various angles is critical since they will inform the overall volume and shape of the groom.

Figure 8: Reference photos of meerkats (left) and renders (right)

Combining multiple references by choosing interesting structures from several different sources helps create a groom that is unique yet based on reality. The groomer can separate those key reference images as primary reference, referring to the rest as secondary throughout the grooming process.

For grooms with a lot of directional changes, relying on image planes and sketching out the directional changes can be a useful trick. This technique can speed up the process of laying out guides more quickly and in the correct areas.

Figure 9: Sketches of directional changes

Growth mesh

While it is possible to attach the groom to the hero version of the model, sometimes it can be easier to have a growth mesh—a separate piece of geometry that exists for the sole purpose of emitting the hair or fur. When you later import the groom into Unreal Engine, you will be able to bind the groom to the hero mesh's skin.

If the model is still undergoing design changes, using a growth mesh can reduce the number of transfers (refitting/transferring the groom to the mesh). Generating more than one growth mesh, each one representing a separate area of the groom, can make it easier to focus on the groom for each area.

Another advantage to separate groom meshes is that geometry in areas that won't take any hair or fur—eyeballs, eye sockets, inside of the mouth, teeth, clothing, and so on—can be removed or hidden.

If there are areas of the mesh's UV layout that lack sufficient resolution, it can be helpful to use a separate set of UVs for painting groom attribute maps.

Real-time dog fur in Unreal Engine courtesy of Jelena Jovanovic - Digitalbite

Real-time hair and fur in Unreal Engine

Unreal Engine primarily performs hair/fur optimization, rendering, simulation, and generation of LODs, with only a very few controls for styling. Ideally, you would have your groom styled exactly as you like before importing it to Unreal Engine. It is not unusual for styling changes to be required after the first import; in such a case, simply adjust the styling in your DCC application and re-import the groom.

Unreal Engine uses the **Alembic** file format as its hair/fur import language. A naming convention-based schema is provided to facilitate the import of static grooms from DCC application tools like **XGen**, **Ornatrix**, **Yeti,** and **Houdini** Hair Utils.

It is important to note that when creating grooms in a DCC application and exporting them to Alembic, you will want to make sure certain attributes are exported into the Alembic file, as described in the Import process section of this paper. Ornatrix and Yeti offer capabilities to export these attributes to the Alembic file for import into Unreal Engine.

Optimization

With the strand-based hair system described here, high-density grooms can easily contain hundreds of thousands or even millions of strands, and each strand can contain dozens of CVs. The combination of these two factors affects performance for import, rendering, and simulation.

With an offline rendering process for feature film VFX, real-world densities and widths are typically used to achieve the highest level of realism possible. But when working with real-time rendering, while it's reasonable to start with real-world densities and widths to match the reference as closely as possible, it is usually necessary to evaluate performance and eventually optimize from there as required.

Optimization for real-time hair and fur is accomplished in the DCC application, before import to Unreal Engine. To start optimizing, it can be helpful to look at the CV count first. If the hair is short and straight, the number of CVs can be significantly reduced without too much visual difference. If the hair is long, scraggly, and complex, there is a limit to how much the CVs can be reduced before loss of structure takes place.

The next step would be optimization through densities and widths. Start by halving the densities and doubling the widths, and re-evaluating as needed. For an average human hair width, a thickness of 0.008 centimeters is usually a good value to start with. Lower values are typically needed for fine hairs around the hairline, for sideburns, and for strays and flyaways.

There is a balancing act between the visual perception of density and the actual density. The illusion of a denser groom can be achieved by having thicker strands, as this will block both light and the direct visibility of the skin. Generally, as density is reduced, it's best to compensate with increased widths until a balance of performance and fidelity is achieved.

This doesn't always work well if the asset is very close to the camera, because the widths will look unrealistic. LODs can sometimes help with this by introducing strand culling and thickening as the camera gets further away, which will improve performance while still enabling your groom to get as close as possible to real-world widths and densities. For further optimization, you can also fall back on the traditional card-based method, where a small number of large, flat sheets is used to give the approximate shape and motion of a much larger number of individual hairs. Unreal Engine provides functionality for these approaches.

Import process

To bring a groom into Unreal Engine, you must first export it from your DCC application to Alembic format. You can then import it to Unreal Engine, where it becomes part of a Groom Asset.

Instead of reconstructing the groom based on guides and interpolation rules, Unreal Engine imports both guides and strands, which precisely preserves the authored groom.

During the import process, the importer will look for attributes and groups that follow the Alembic naming conventions outlined in the **Alembic for Grooms** specifications page, and will import them to a new Groom Asset. When the schema is implemented, it enables the transfer of attributes such as **width** and **color** into Unreal Engine, along with **guide** attributes that are identified for the simulation of interpolated hairs. Multiple hair groups within a single Alembic file are supported via **group_id**. Among these attributes, the **rootUV** attributes will fetch the UV of the underlying surface (e.g. skin surface). This enables the creation of some spatial variation across the groom, depending on the underlying surface.

If the Alembic file contains only curves but doesn't follow the Alembic export schema, the groom will still be imported into Unreal Engine, but without custom attributes.

To learn more about the import process, see the **Hair Rendering** topic in the Unreal Engine documentation.

Groom Asset Editor

The Groom Asset Editor is available when a Groom Asset is selected. The controls in this editor assist with coloration, optimization, and other aspects of the groom within Unreal Engine.

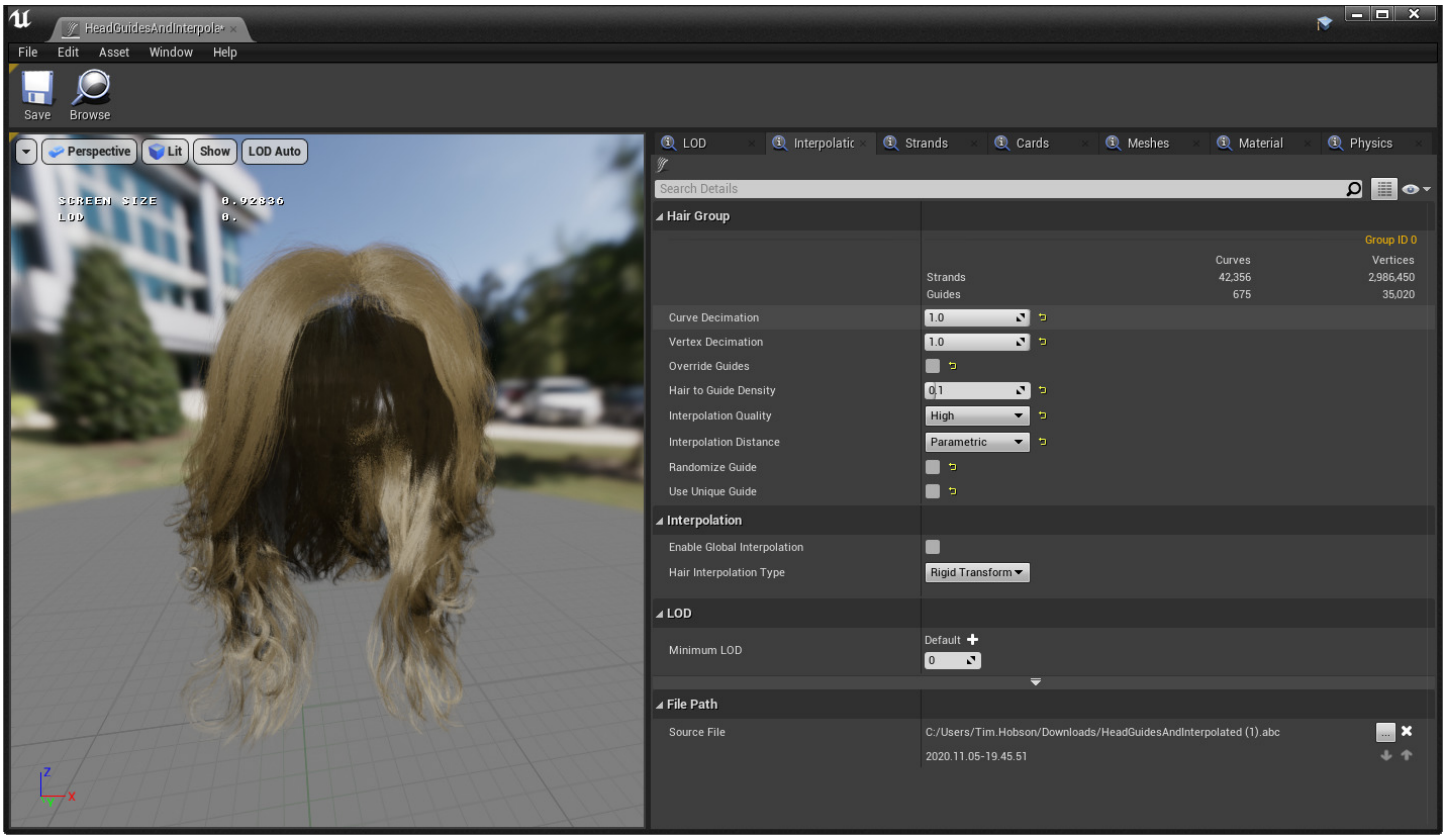


Figure 10: Groom Asset Editor in Unreal Engine

While the Groom system is primarily designed to handle grooms made of strands, alternative geometry representations such as cards and meshes are also supported for scalability purposes. For an easy setup, all these geometric representations are managed within a single asset (Groom Asset) and component (Groom Component). An LOD panel configures which geometrical representation should be used for each LOD, and how strands should be decimated. A groom asset can be made of several groups (a fringe group, a pony tail group, etc.) and each group can have its own LOD settings.

For more information on using this feature, refer to the **Groom Asset Editor User Guide** and the **Hair Properties Reference** in the Unreal Engine documentation.

Surface attachment

The groom is ordinarily attached to a **SkeletalMesh Component** which has been rigged.

To attach a groom to a skinned surface, a Groom Component needs to be attached under a SkeletalMesh Component in the hierarchy, and a Binding Asset needs to be provided (see the Binding Asset Options section of the **Hair Properties Reference** in the Unreal Engine documentation). This Binding Asset stores information about the hair strands' projection onto the targeted SkeletalMesh.

During the attachment process, **barycentric coordinates** are computed from each hair root to the closest triangle on the mesh. The hair roots stay constrained to the SkeletalMesh when the mesh deforms during animation.

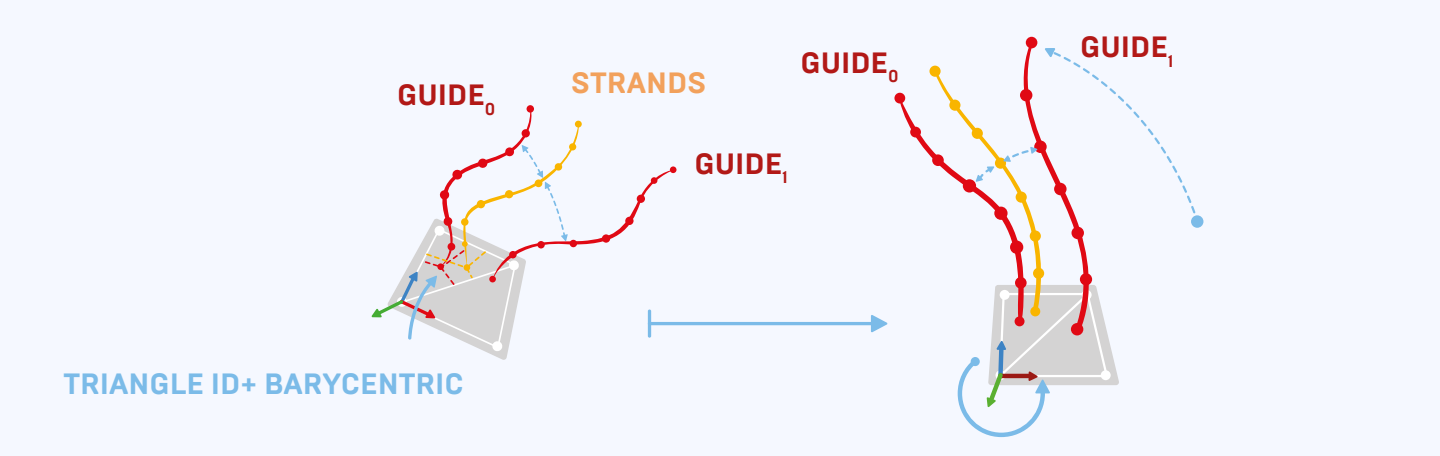


Figure 11: Illustration of how strands stay "rooted" to mesh triangles during skin deformation

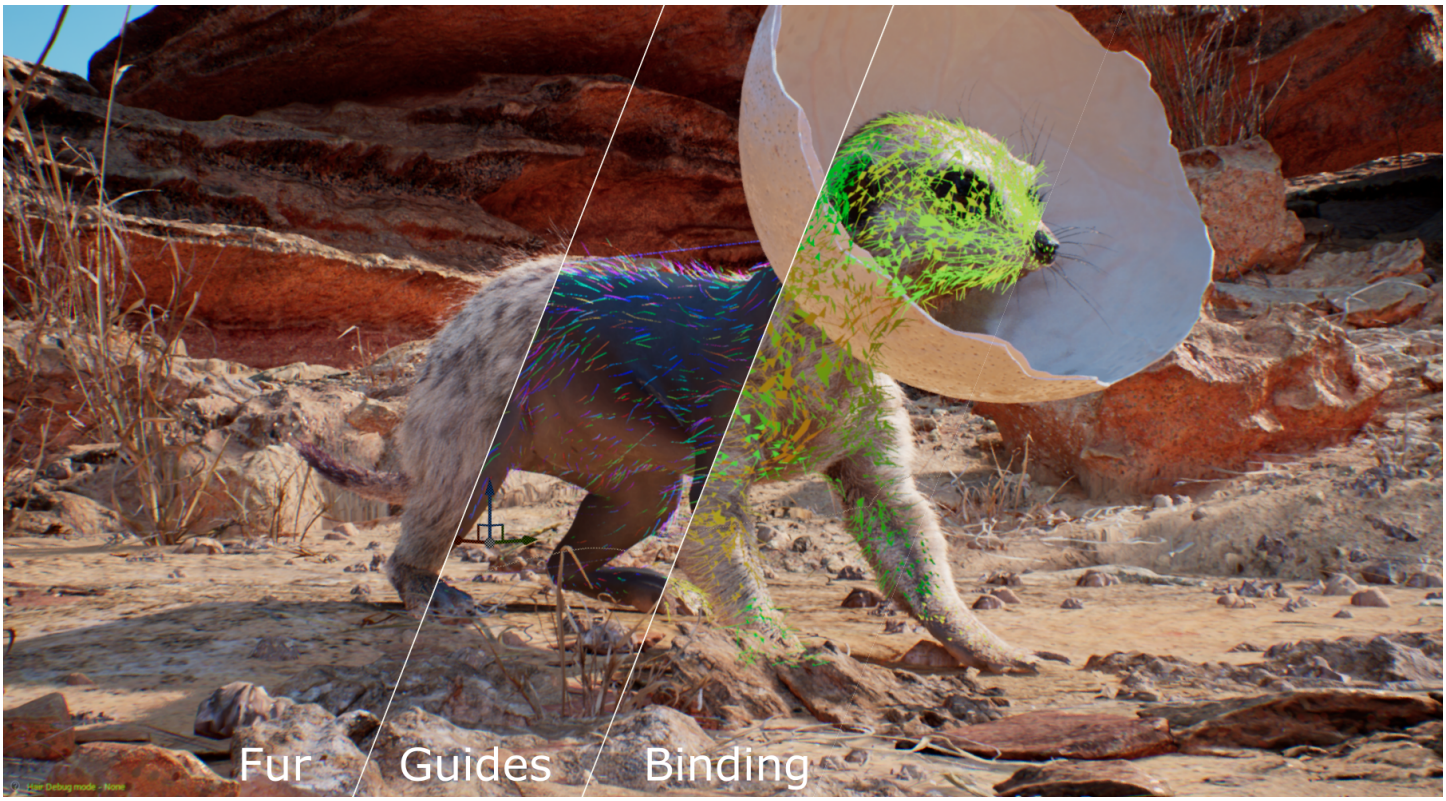


Figure 12: Breakdown of fur, guides, and binding on meerkat character

Skin deformation

When the skin surface is deforming, the strands attached to the surface can get pinched or shift into unnatural positions. Fine grooms such as eyelashes, eyebrows, or mustaches, for example, can be sometimes challenging to render realistically due to the fact that they are composed of thin strands on an uneven or creased skin surface.

To address these problems, grooms can opt into a global deformation constraint. This constraint forces hairs to remain as close as possible to their original shapes, even under heavy skin deformation. A set of radial basis functions (RBFs) is precomputed at several points on the attached SkeletalMesh, and applied at runtime to preserve the original groom position.

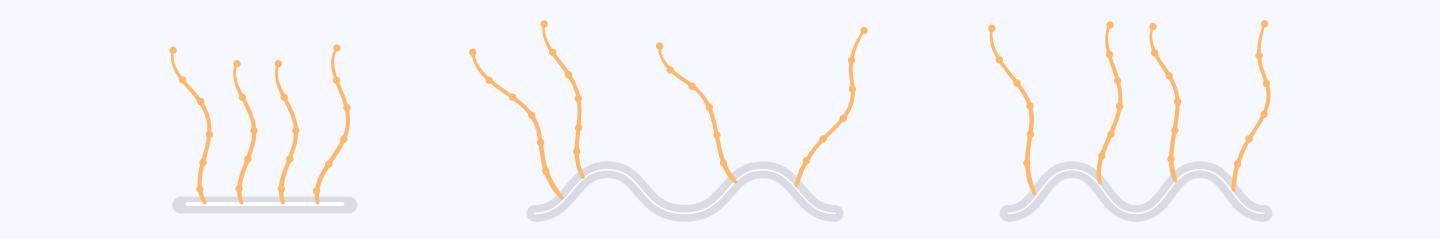


Figure 13: Hairs on skin at rest (left), hair on pinched skin deformed without RBF constraints (middle), same skin/hair with RBF constraints (right)

Light approximation

The shading method for hair and fur aims to approximate the complex behavior of light on real-life hair and fur while still staying within the bounds of real-time calculation.

In life, the color we see on hair is influenced by several factors: the color of the hair itself, the way light hits it, how it refracts and scatters light, and how it is affected by the refracted light bouncing off other strands.

The following diagram gives a simplistic view of what happens when light hits a strand of hair or fur. The strand is both reflective and translucent, so some light is reflected directly off the strand, forming the shine and highlights, while other light is absorbed by the strand and then scattered.

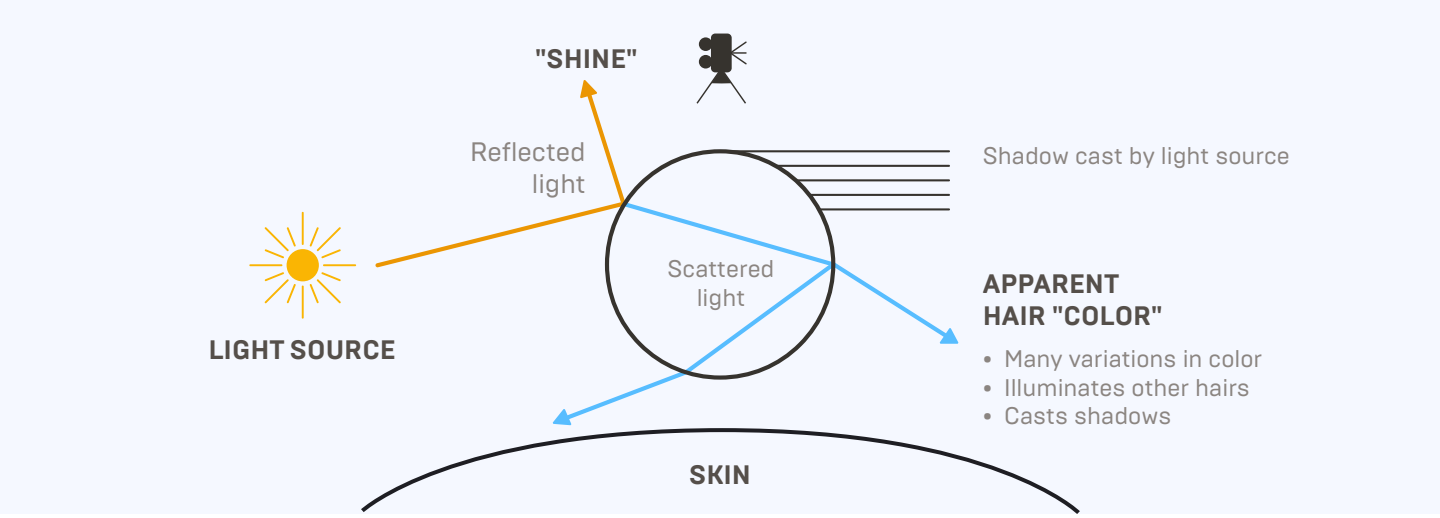


Figure 14: Cross section of a strand of hair or fur, and the activities of light that hits it.

For the light that the strand absorbs, it can be scattered in one of two ways: it can either go across to the other side of the strand and exit, or it can bounce inside the strand itself before exiting. These two methods of scattering light, referred to as "fiber scattering," account for the unique, complex appearance of hair and fur. This phenomenon is well known, and is well-described by the mathematical function called the Bidirectional Scattering Distribution Function.

In addition, light escaping from fibers in this manner can scatter light onto other fibers, causing what is called "fiber-to-fiber scattering." This effect is even more complex, and thus even more time-consuming to compute.

When you consider all the different ways that light can impact a rendering of hair and fur—shadows, ambient occlusion, backlight, and so on—the effects of this scattered light on thousands of strands of hair or fur can become very complex indeed. For example, it is relatively easy for human beings to tell the difference between fake and real animal fur just by its appearance, which is almost entirely due to how light affects it (as opposed to the coloration of the strands). If we are to create convincing CG hair and fur, we need to take all these lighting effects into account.

Early methods of shading CG hair and fur tried to simulate the physics of how light behaves when it hits a strand, which gave realistic results but was computationally intensive and caused exceedingly long render times. However, a number of faster "shortcut" computations have since been developed which yield results very close to those of real-world physics. Unreal Engine uses many of these approximations to achieve realistic hair and fur with real-time rendering.

Figure 15: CG rendering of hair with combination of reflected light (brightest areas), scattered light (medium brightness, apparent hair color), and dark areas where little light reaches

One example is an approximation called *dual scattering*. This method decomposes this complex phenomenon of scattering into two parts: global scattering, which is the coarse/macro estimate of how light travels through the many fibers/hair, and local scattering, which is a simple model of how the light bounces around just one fiber. The time needed to compute the effects of dual scattering is orders of magnitude faster than computing the effects of each and every light ray on a series of hairs, yet still yields results very similar to those of much more computationally intensive methods.

Another time-saving method for approximating realism in hair and fur is *deep opacity maps*. This approach creates dedicated shadow maps per light, storing hair absorption into several shadow map layers. Because of the way the maps follow the shape of the hair-emitting object, fewer maps are needed to accomplish the same level of realism as multiple opacity shadow maps, making this method faster to compute.

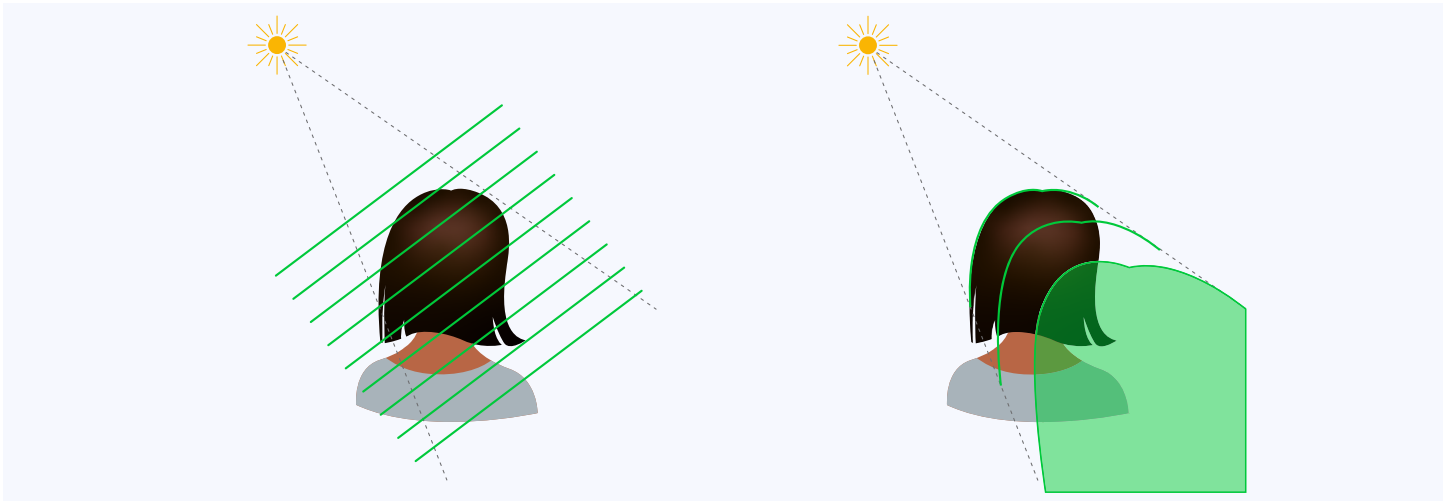


Figure 16: Representation of shadow opacity map (left) and deep opacity map (right)

For more information on these and other techniques Unreal Engine uses to approximate light behavior for hair and fur in real time, see the **Resource for further reading** section of this paper.

Density volume

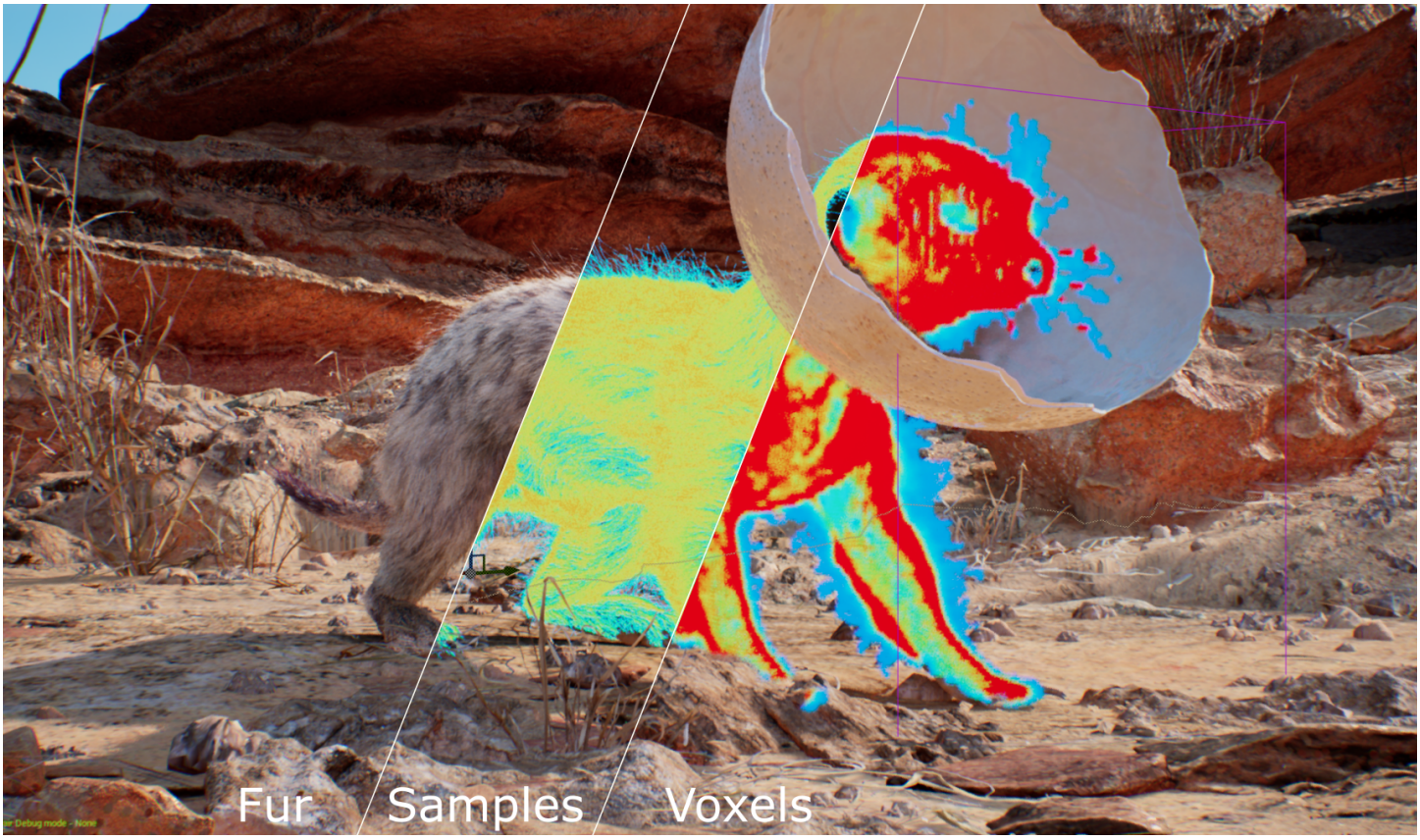


Figure 17: Density volume, samples, and resulting fur

Another part of Unreal Engine's approach to approximating light involves the creation of a virtualized density volume for each frame. The density volume provides information on the density or "thickness" of the fur from the current viewpoint (see Figure 17). For example, compare the "solid" appearance of the fur at the center of the meerkat's leg (green) as opposed to the flyaway tufts at the leg's edge (blue). These cues give the rendering system the information it needs to quickly approximate light behavior for that type of fur density. The density volume makes use of voxels (tiny cube-shaped 3D placeholders), with the strands voxelized into the density volume.

The density volume is the default mechanism for hair shadowing, ambient occlusion, and environment lighting, enabling plausible backlight, self-shadowing, and illumination. However, the artist can opt to also use deep opacity maps, which can be set per light source in Unreal Engine.

Once the lighting is computed for visible strands, the system moves to the next step: rendering.

Rendering

For rendering in Unreal Engine, the imported hair strands are converted into individual polylines. These curves are regrouped into small clusters of strands which are used for LOD selection and culling purposes.

When the real-time system renders multiple thin hair strands within each pixel, aliasing problems can arise. Depending on the performance and quality budget, different approaches can be used for rendering hair. For real-time rendering, Unreal Engine's default primary visibility algorithm uses a combination of a **MSAA** (multi sampling anti-aliasing) visibility buffer to extract the first visible strands, and also a sub-pixel coverage buffer. This combination provides a middle ground between performance and quality.

For cinematics, where we are willing to sacrifice performance in favor of quality, we can use a different approach, dynamically building a list of visibility strands. This helps for having a better sorting and transparency effect. For both algorithms, we output a list of visible strands for every pixel after a compaction pass to reduce the number of samples per pixel.

Grooms are split into small clusters for which LODs and culling information are computed. When a groom is rendered, these clusters allow fine-grained LOD selection based on the screen coverage and visibility/occlusion. This enables us to adapt the rendering and performance costs based on what is actually visible on screen. It is particularly helpful for fur on large animals, as some parts of the groom could be close to the camera at the same time that other parts are far away.

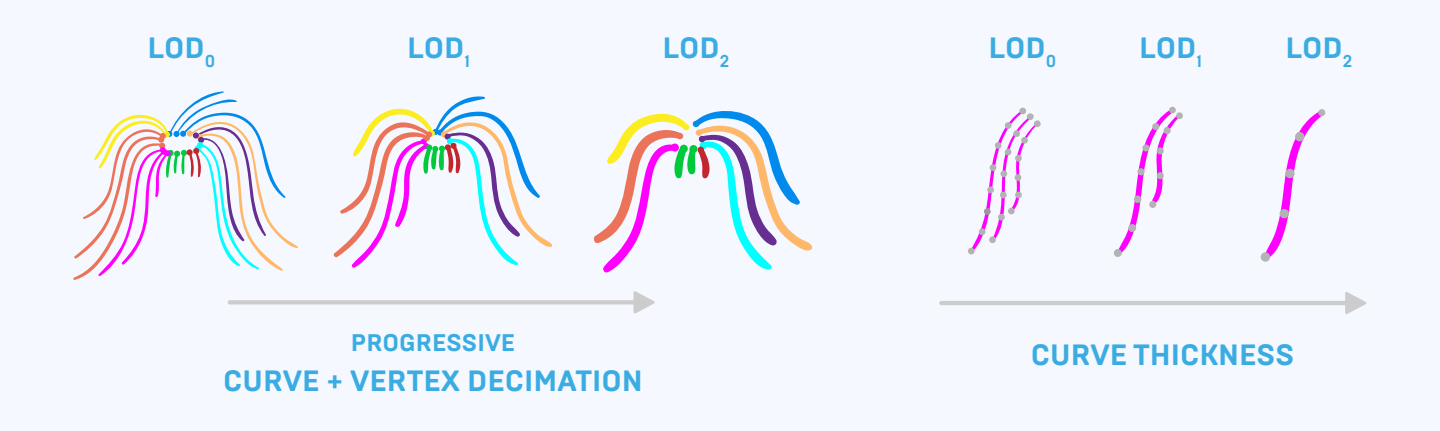


Figure 18: Methods of formulating LODs for hair/fur

For scalability purposes, the groom systems support alternative geometry representations, like cards and meshes, to switch to even simpler rendering techniques when the groom covers only a few pixels, or for less powerful platforms.

Light and shadows on thin hair

Shadows cast under directional lighting by thin hair strands, like eyelashes or whiskers, might not be well-preserved due to the lack of resolution of the shadow map or hair voxel resolution. To address this, when ray tracing is enabled, the exact hair geometry can be directly used for casting shadows, generating sharper shadows from these thin features. However, one needs to be careful, as the more complex the hair/fur geometry is, the more expensive this option becomes.

The human face's skin is entirely covered with thin hairs called "peach fuzz." Like other hairs, peach fuzz is lit by all the surrounding lighting. But due to their light-colored nature, a non-negligible part of their lighting comes from the light bouncing off the skin's surface. In Unreal Engine, grooms can opt into this special lighting contribution.

Simulation

Real hair and fur moves in a specific way when reacting to head and body movement, wind, gravity, etc. Animation of hair and fur due to these forces is generally accomplished through a physics-based simulation, which can be very computationally heavy. However, to achieve photorealism in animation, any CG hair or fur simulation must be able to generate believable animation from these forces and movements.

Hairs are challenging to simulate because of their large numbers and the complexity of the motion, both with object collisions and self-collisions. Hair dynamics rely on complex interdependencies. In the case of animal fur, muscle and flesh sims are responsible for a large portion of the fur motion, in addition to the hair simulation itself.

The guides used for simulation (which are not necessarily the same as the hair guides) drive the shape and motion of the associated interpolated curves. The intelligent choice of guides can be more important than the number of guides—quality is more important than quantity.

Usually the grooming guides provide some sort of structure in the groom, such as clumping, direction changes, flow, etc. The structures will usually need to be maintained during simulation, so the grooming guides are a good starting point for the simulation. If these guides don't provide enough resolution, the user can supplement them with additional simulation guides derived from the groom. If performance is the main focus, you can also decrease their number to reduce the computation time.

Hair simulation in Unreal Engine is implemented as part of the **Niagara VFX system**, and for performance reasons is available only on the GPU. The solver is based on XPBD (Position-Based Simulation of Compliant Dynamics). To solve all the different constraints, the user needs to provide the number of substeps and solver iterations alongside all the strands, collisions, and constitutive models parameters. The solver is targeting the original mesh under gravity (rest pose) to keep the final style as true to the original style as possible.

When a physics asset has been created and added to the SkeletalMesh, the simulation solver handles body collisions against the primitives of that physics asset.

Self-collisions are computed based on an average velocity field built from the rasterization of the particles' velocity onto a regular voxel grid. Regarding the constitutive models, the main challenge is usually finding a good hair constitutive material model that enables strands to behave realistically, at low solver iteration count. These models control how the strands will stretch, bend, and twist during simulation.

We are constantly testing new constitutive material models to see which ones offer the best balance between quality and performance for hair and fur. The Cosserat Rod and Angular Spring methods are already available within the groom asset physics properties. For more information on these techniques, see the **Resources for further reading** section at the end of this paper.

Skin darkening

As the camera moves closer to certain grooms where the human hair and skin colors are very different, the base of each hair shaft can be the most visible part of the hair strand. With human skin being relatively transparent by nature, one can notice the hair bulb beneath the skin, also called a follicle. This follicle causes a faint but noticeable darkening of the skin. To simulate this aspect, a follicle texture is generated for the groom and used within the skin shader (option available on the Groom Asset).

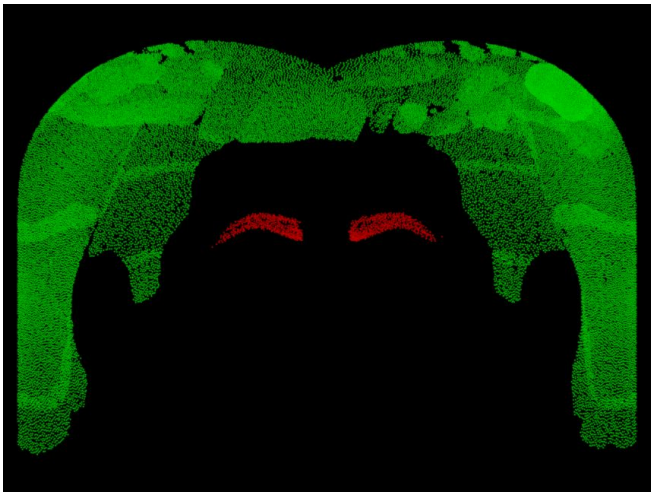
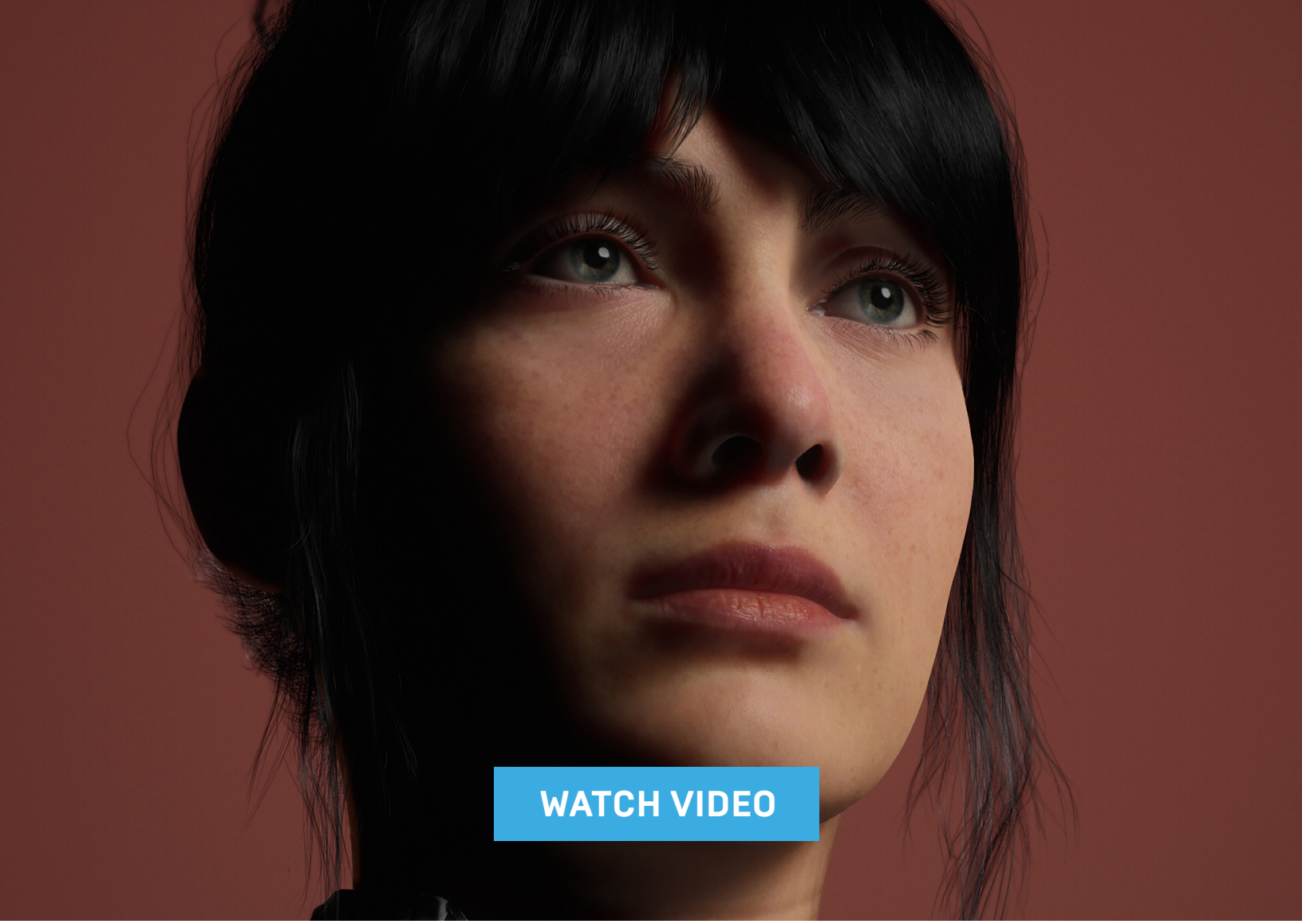


Figure 19: Follicle map



Use case: Human hair with MetaHuman Creator

To promote further understanding of real-time hair and fur, we present here some practical use cases in Unreal Engine along with details of their implementations.

To illustrate grooms for human hair, we will take a look at the grooms on the sample digital humans created with **MetaHuman Creator**, which are available for download.

MetaHuman Creator is a new tool for building digital humans complete with facial and body rig. The model, which can be animated in Unreal Engine, comes with a number of hair options, both for head and facial hair.

The hair choices in MetaHuman Creator are grooms pre-created in a DCC application and imported to Unreal Engine. With controls within MetaHuman Creator, you can mix and match to create the style for hair, eyebrows, mustaches, and beards. You can adjust the color of grooms within MetaHuman Creator prior to generating the character.

Both the grooms described here were created using the XGen plugin in Maya.

Side-swept fringe groom

The side-swept fringe groom consists of a sweeping fringe in the front and naturally shaped curls in the back.

Side-swept fringe groom

Facial groom

Groom artist: Marija Blašković

With the fringe in the front and curls in the back, it was logical to separate the style into layers (i.e. groups). Layers were built from the bottom up, similar to the workflow hairdressers use when cutting hair. This technique helped with the careful layering of hair as the style was built upwards.

By inspecting references and observing how curls formed in similar hairstyles, we came up with a few rules that helped in building this particular hairstyle; curls are mostly imperfect, and even though hair has a tendency to form clearly defined entangled spirals, they are still quite broken up, especially in the upper layers where the hair is longer. The curls tend to lose shape closer to roots, gradually twisting along the strand and interlocking with each other, forming interesting shapes.

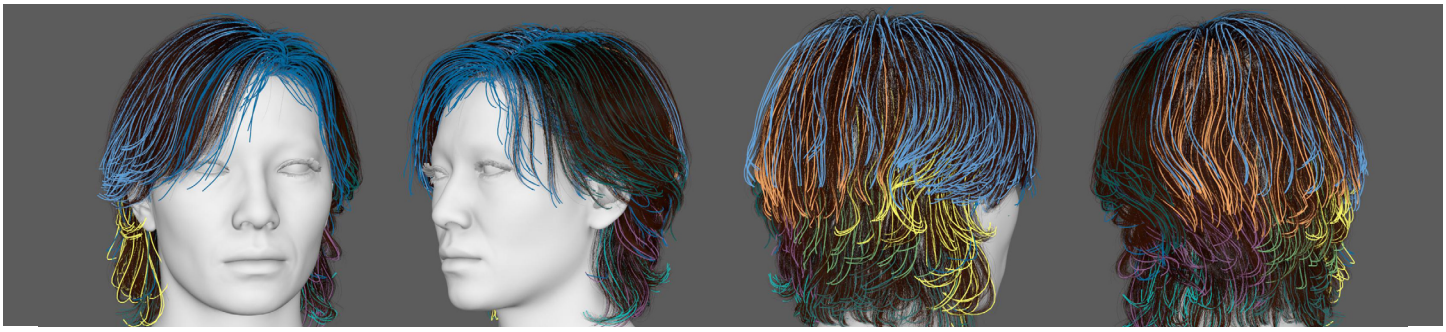


Figure 20: Guides with color-coded layers

The clumps were broken into two or three sub-clumping layers, with random offset values along each strand. In addition, in different sub-clumping layers, different percentages of clumps were set up to not follow the primary clump shape. This further broke the shape and added clump variations.

High-frequency noise helped break the clean and generic look of straight hairs, and additional noise was added to the tips to further break the clumping. Widening of the clumps at the tips, and randomizing length, also contributed to more realistic curls that felt more natural.

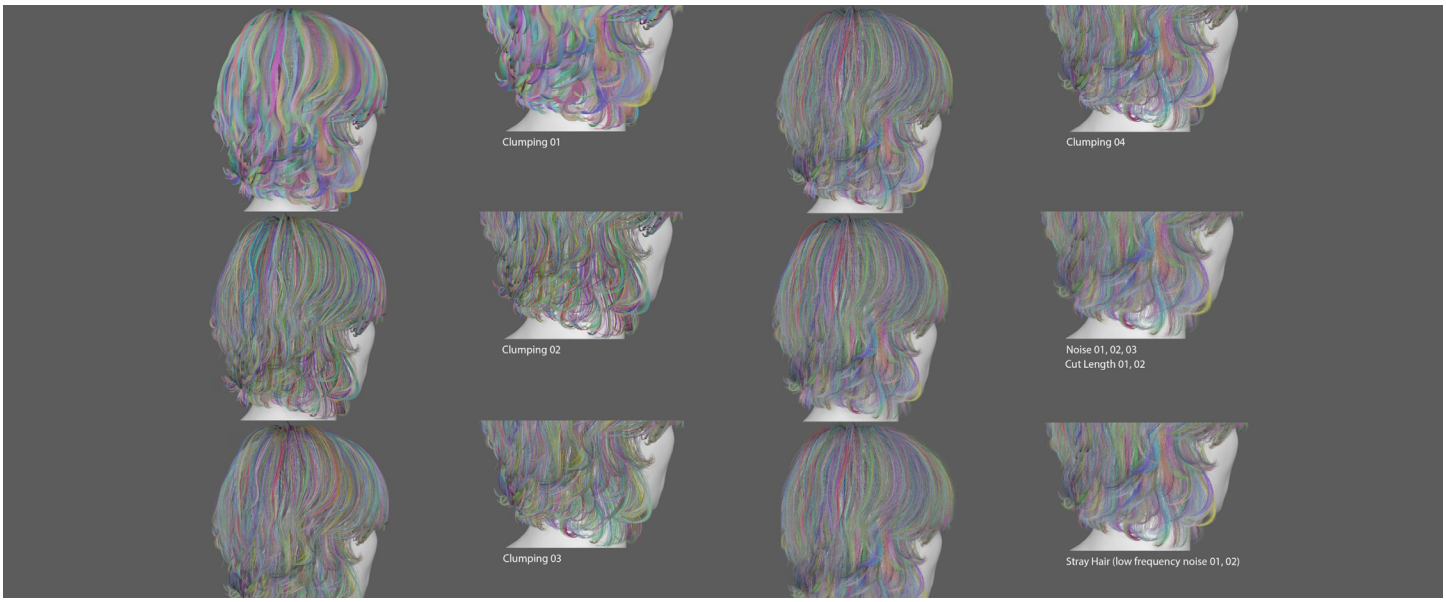
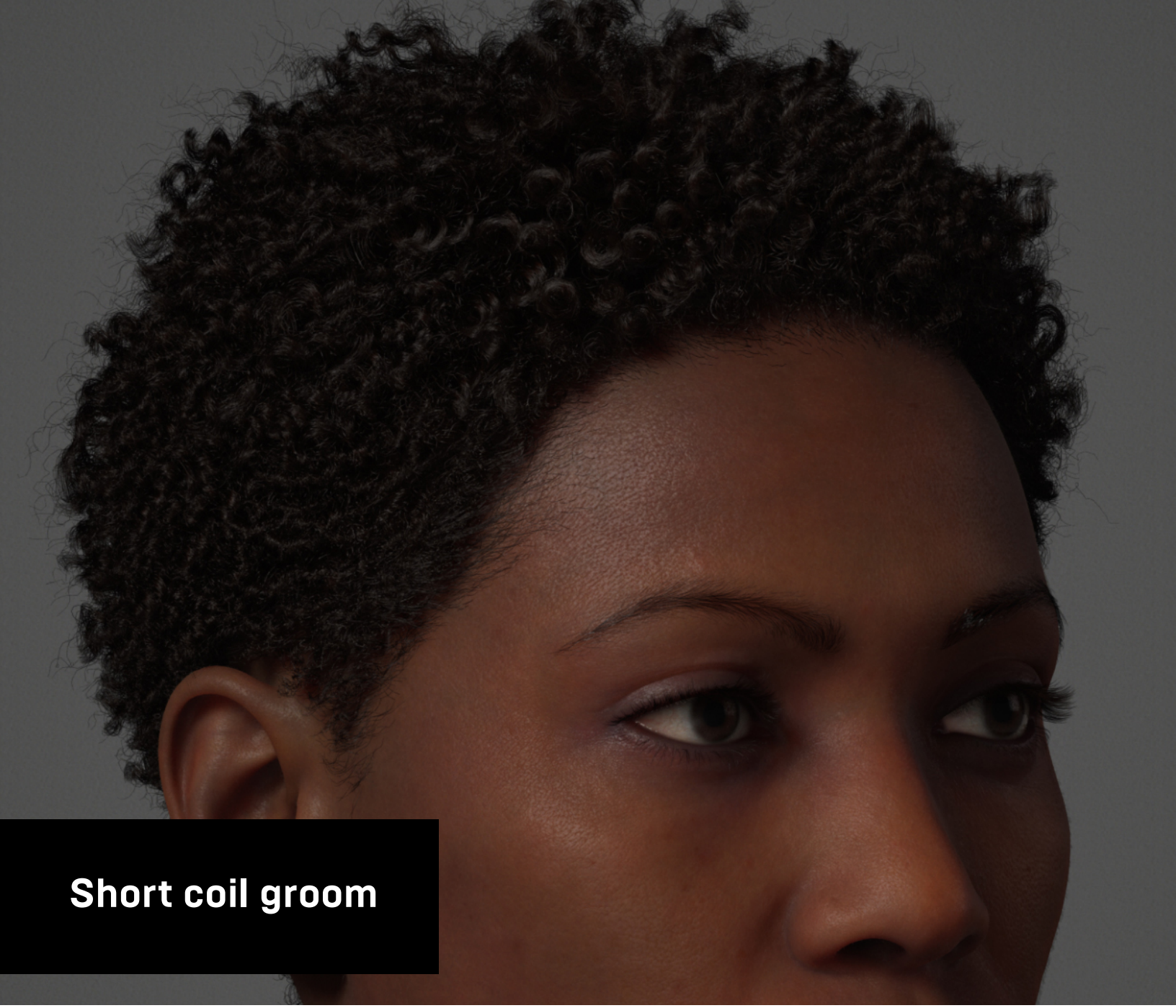


Figure 21: XGen modifiers stacked to produce clumping and noise

Finally, we added two or three layers of low-frequency noise that introduced layers of hairs that were completely disheveled. These layers significantly helped with creating a realistic look.

Short coil groom



Short coil groom

Groom artist: Nikola Milosavljević

Short coil groom

The short coil groom is a short hairstyle with thick, somewhat kinked corkscrew curls. To look natural, the curls must have a great deal of variation, with numerous individual curls visible in profile from every angle. The forehead hairline is also highly visible.

For the initial creation of guides, we opted for attribute-based grooming with a rough density mask and a sculpted volume mesh, which acted as a MeshCut modifier inside XGen. It helped to quickly block out the hair length and the overall shape of the groom.

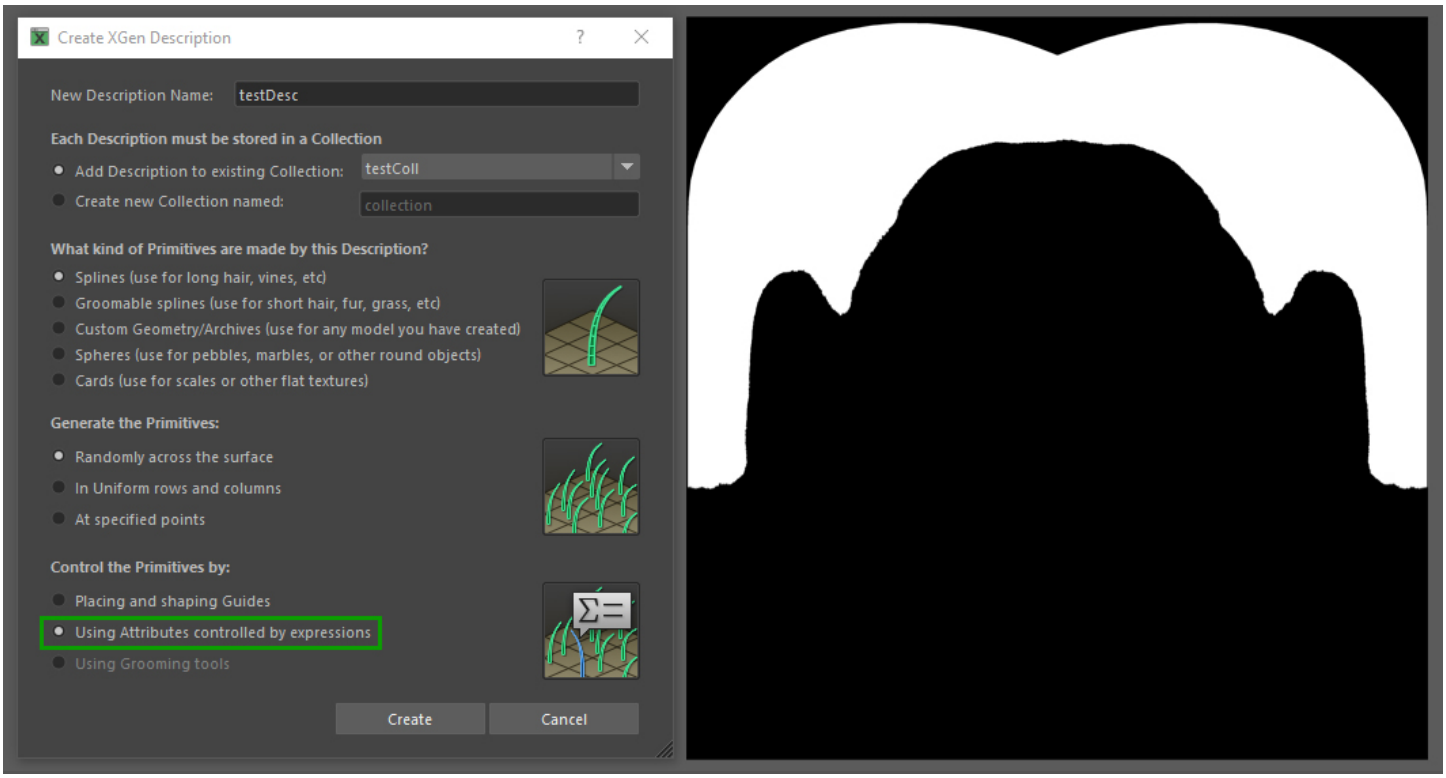


Figure 22: Density map in XGen

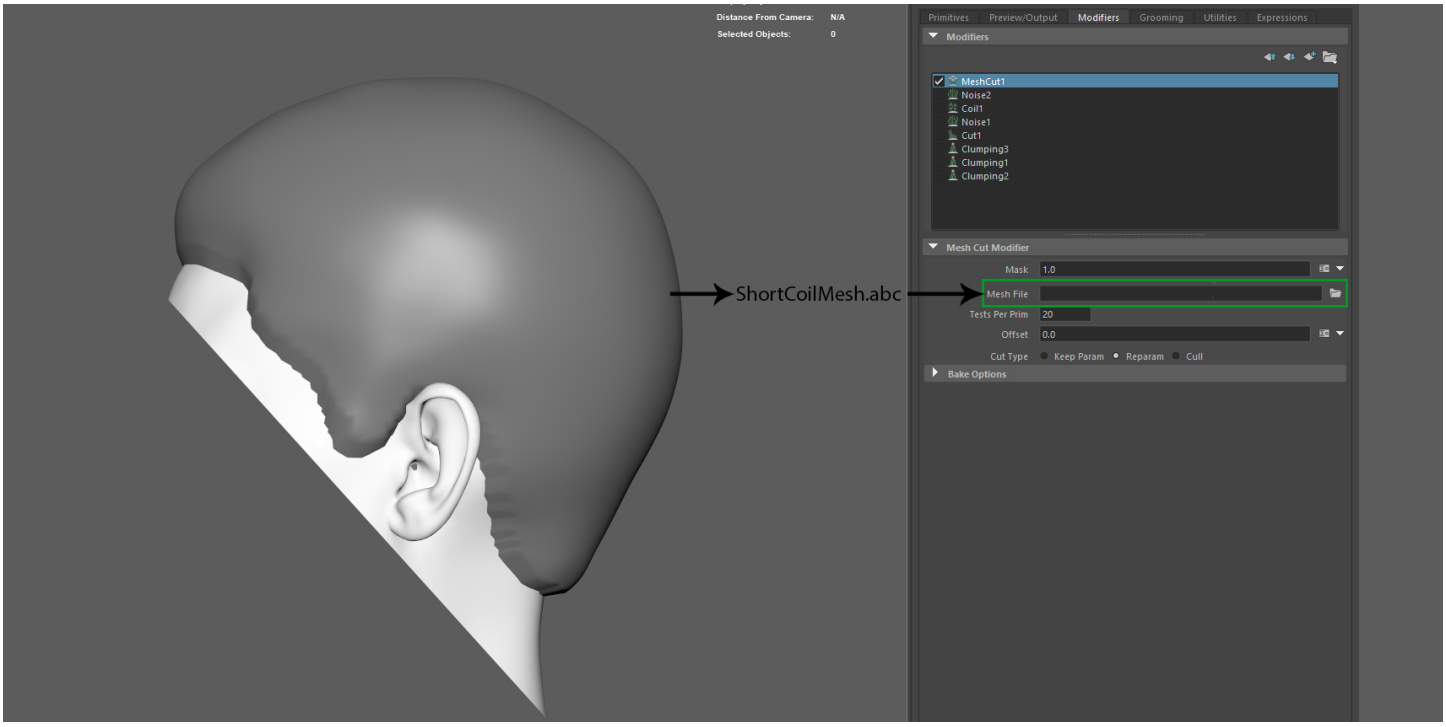


Figure 23: Section of head defined as initial area for groom

A percentage of hair was exported as guides to be used in a new spline description (a layer or group in a groom).

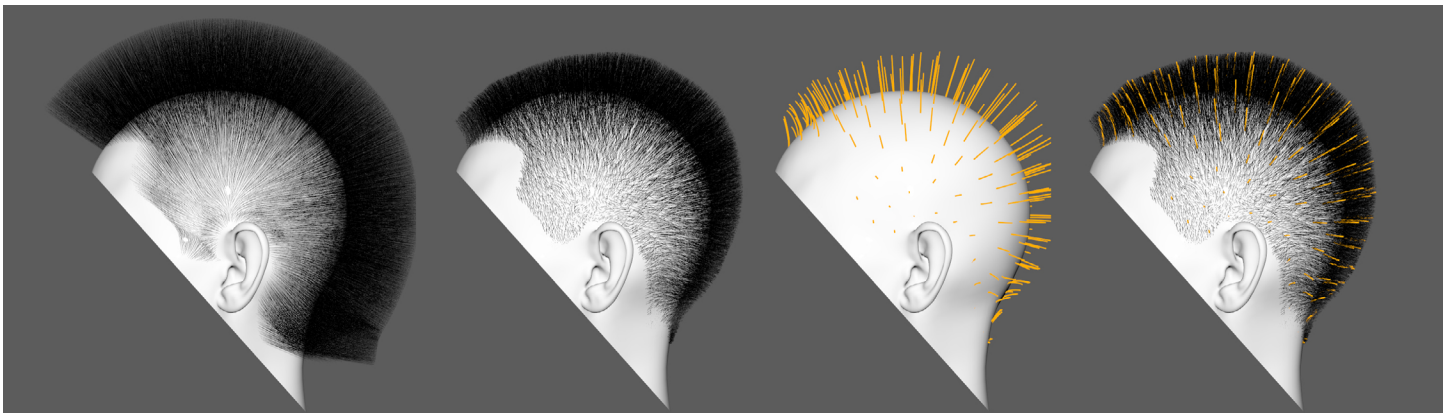


Figure 24: Rough groom exported for use as guide hairs

The new description was sculpted and a base clumping map was created. It was split into three separate layers to gain more control of parameters such as coil thickness, shape, number of turns, etc. Expressions were used to randomly mask out a percentage of coil guides to be exported to each new layer. These expressions created a mask from the base clumping map's IDs.

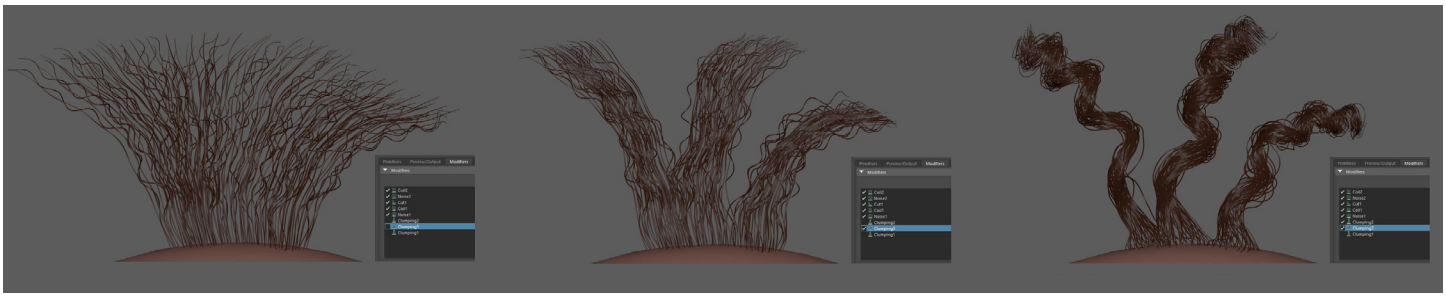


Figure 25: Three types of layers for coil groom

Coils were created procedurally, using expressions to blend the number of turns between the root and the tip, with some control of the location where blending occured along the length of the guide (closer to root or tip or somewhere in between). Disabling the **Uniform CVs** option yielded better results, where the shape was more precisely defined along the length of the coil. Enabling the **Uniform CVs** option tended to average out the shape.

The **Modifier CV Count** parameter also played an important role. A good value to start with was 80-100 CVs, where we could see the appropriate level of detail in the curls without using up an excessive amount of system resources. The CV count was later optimized to 60-70 for export.

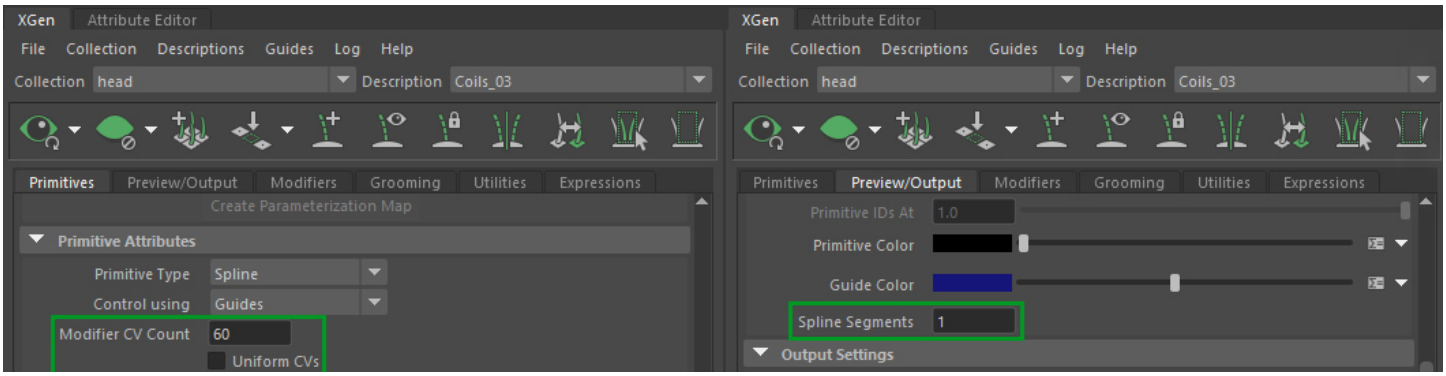


Figure 26: CV count modifications in XGen

Each coil description was controlled by two clumping modifiers. One modifier had a lower clumping intensity that just attracted hair closer to the guides, and one modifier did the actual curling. On top of that we used stacked noise, coil, and cut modifiers to break up the uniformity.

Clumping intensity was defined by a map which was derived from the description's base clumping map. The texture was loaded into the Mari texture-painting software, projected onto the scalp mesh, and the **Edge detect** filter was applied. This was followed by the **HSL** and **Contrast** adjustment layers, which produced a grayscale map with a structure like packed cells or reptile skin. The map was then exported from Mari and further manipulated in Photoshop, where we created a gradient within each cell from its center to the outer edges.

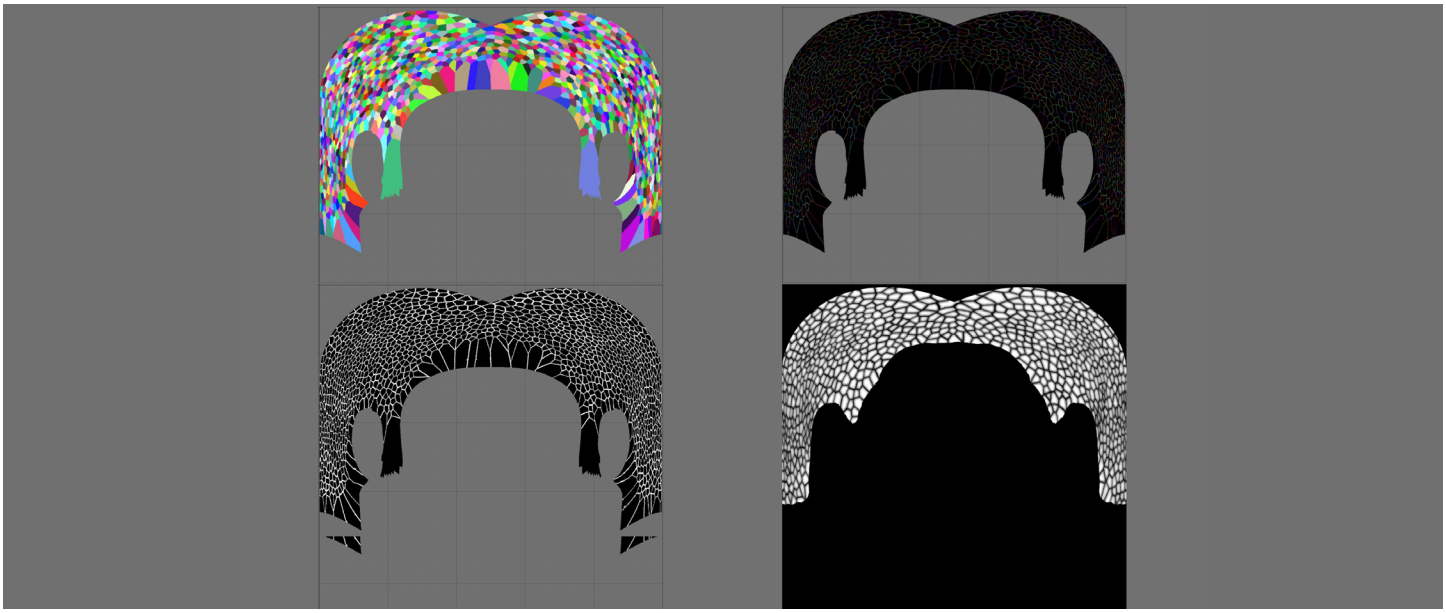


Figure 27: Clockwise from top left: clumping map, edge detect filter, HSL and contrast map, and final clumping map from Photoshop modification

Fill

Where curls do not fill in the scalp, we sometimes rely on fill hairs. The fill description's density map was created in the same fashion as the coil's clumping intensity map, except that cell borders are colored white and the cells are black. The map constrains hair growth to the white areas of the map, adding density only where necessary. This automates the painting that you would otherwise have to do manually. The map can be additionally modulated by XGen's noise functions.

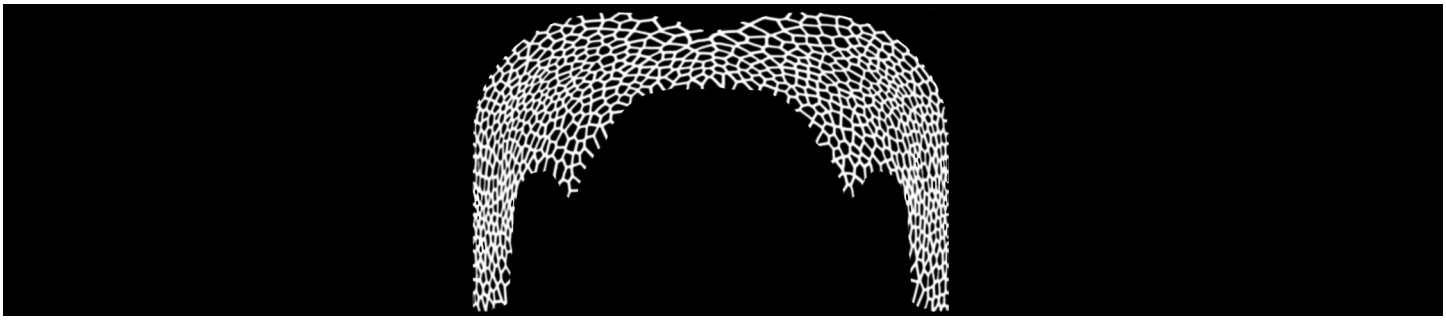


Figure 28: Fill map

The coiling effect on the fill hairs was looser than the coiling effect on the curls. Variation was generated by noise and coil modifiers.

Use case: Animal fur and feathers with *Meerkat* project

For fur and feathers use cases, we look at the meerkat and eagle from the animated real-time short ***Meerkat*** by Weta Digital. The film tells the tale of an amusing confrontation between a meerkat and a martial eagle. The meerkat required real-time fur, while the eagle required real-time feathers. The grooms for both animals were created in Maya with the Yeti plugin.

As much as possible, the shaders were built to mimic the same biological and morphological principles that govern real hair and feathers. Understanding growth patterns, proteins in charge of pigmentation, and pigment release timing mechanisms were key to replicating some of the patterns in living specimens of meerkat fur and martial eagle feathers.

For example, the mottled pattern seen on the back of the slender-tailed meerkat (*Suricata var suricatta*) is a result of specific hair growth pattern in conjunction with timed melanin pigmentation, which causes hair strands to have two or more bands of pigmentation caused by the agouti gene. Therefore, in the shader there needs to be control over the colors of the fur spatially throughout the body as well as control over the timing of the colors along the hair strands.

Figure 29: Reference photos of meerkats

The two main characters, the meerkat and the eagle, appear on screen together several times, but the fact that they are two very different-sized animals complicated the fur lighting and shadowing.

As described in the **Light approximation** section of this paper, the shadows cast by grooms and the light transmission within the groom relies on a voxel structure. The size of the voxels determines the accuracy of the lighting and shadowing. Having too large of a voxel will result in a "blobby" and incorrect look, but having too small of a voxel requires a lot of allocated memory.

For addressing this latter point, we rely on a sparse voxel structure, where allocation is done only where needed. The allocation is driven on the GPU-based on clusters built during the groom import. The bounding information of these clusters defines where the groom occupies space, and where memory needs to be allocated. In addition to saving memory, this sparse structure also enables faster tracing of rays by skipping empty space.

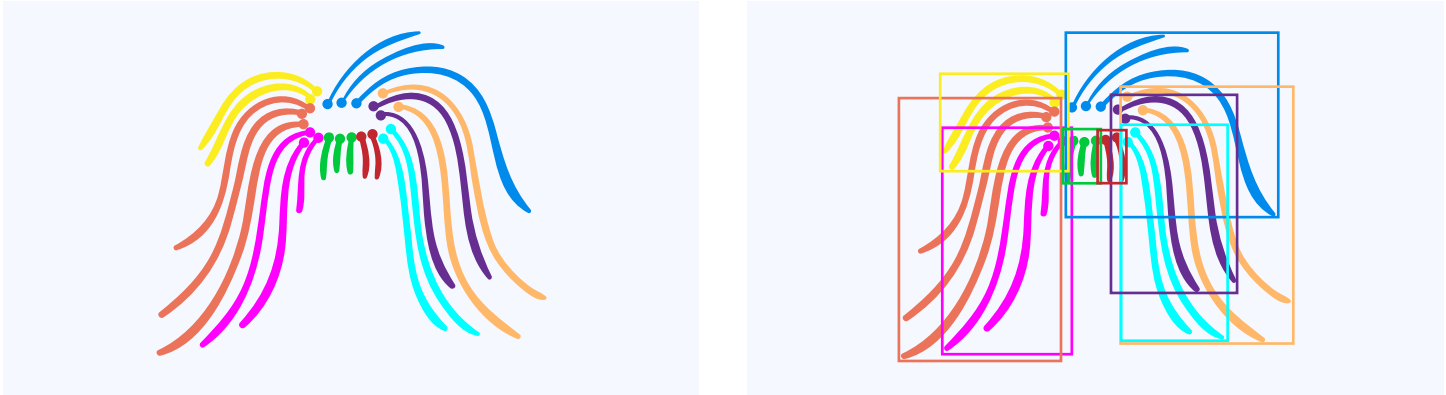


Figure 30: Sparse voxel structure allocates memory clusters only where needed

Ambient occlusion and sky lighting relies on this structure to cast correct occlusion and propagate light through the groom. Unreal Engine offers several ways to compute the sky lighting contribution, with tradeoffs between accuracy and speed. For the *Meerkat* project, we used the most accurate version, which properly integrates the sky contribution with the fur and feather material by casting several rays through the fur, and computes an accurate estimate of the scattered lights.

For both characters, textures were painted directly on the character's skin using Substance Painter. These textures are low-resolution images containing RGB masks which are then used to control the color and patterns in the shader. There are two main masks—one defines zones that control the color spatially throughout the body, while the other defines the zones that control the timing of color banding along the hair strands.

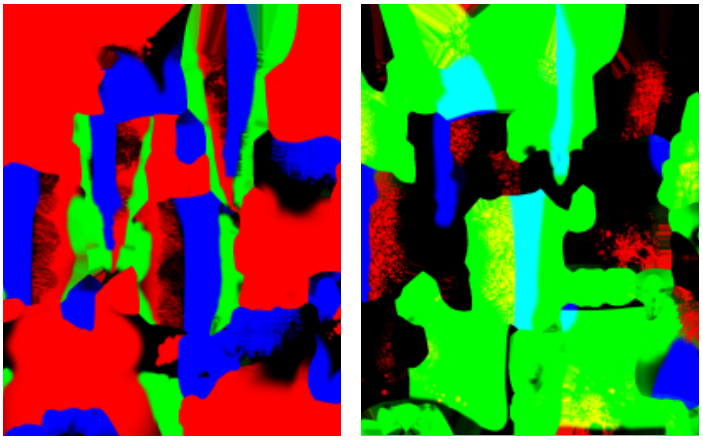


Figure 31: Spatial zone (left) and timed zone (right) masks for eagle

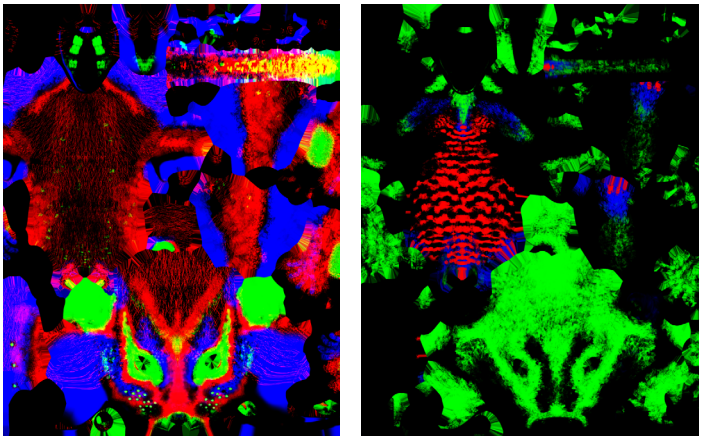


Figure 32: Spatial zone (left) and timed zone (right) masks for meerkat

Texture maps were generated using the skin's UV space. These maps define the color throughout the entire length of hair strands using their **rootUV** attribute.

Meerkat fur

As the meerkat moves quickly, having precise motion vectors was important for producing accurate motion blur, as well as for anti-aliasing, as we needed to re-project previous frame lighting onto the current frame. For this purpose, previous strands' positions are cached, and computing motion vectors becomes as trivial as computing the difference between positions.

Real-time performance of fur in Unreal Engine was critical, so reasonable fur densities were important. The meerkat used only about 25% of the density one would expect in feature film VFX work. Reducing densities and balancing the widths can help provide convincing and detailed fur in real time.

The meerkat body was groomed as a single groom, but we created separate groups for the fine face fur, the whiskers, eyelashes, and other unique elements.

Figure 33: Meerkat groom in Unreal Engine

With the meerkat, the whiskers and eyelashes were hand-placed guide curves, directly converted to strands inside the Yeti node network.

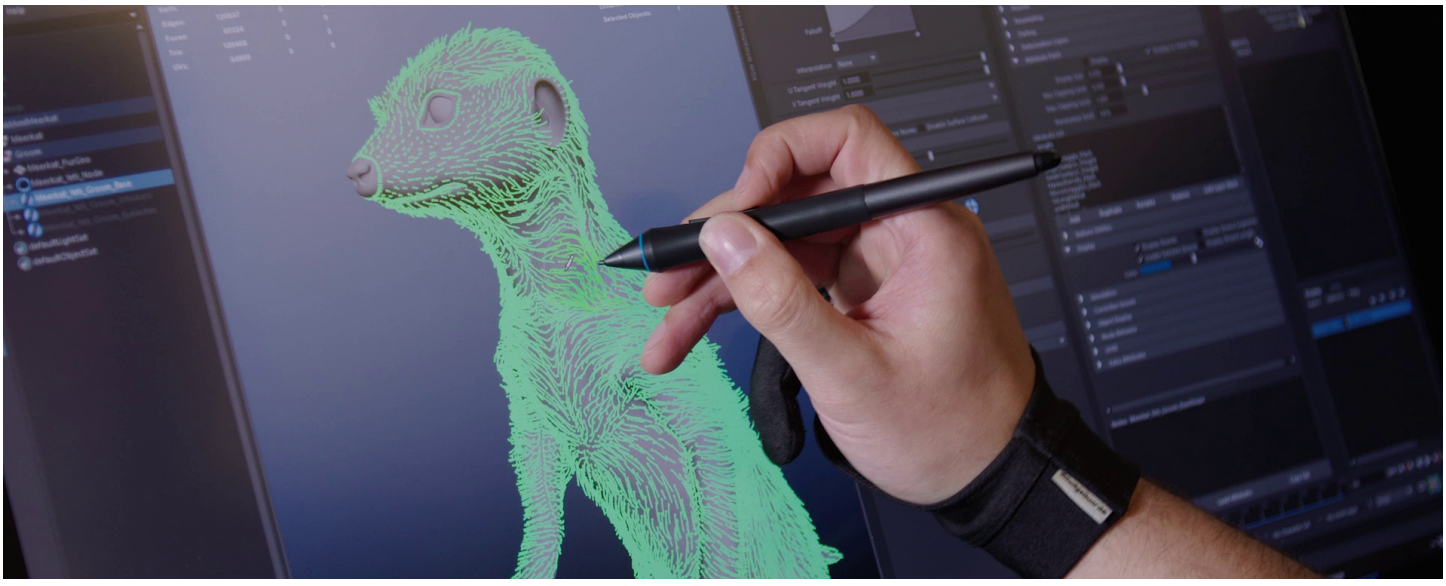


Figure 34: Sketching meerkat guide hairs

Eagle feathers

The eagle was a unique challenge due to the inherent complexity of feathers. Yeti builds feathers using a feather primitive, where the spine of the feather is effectively a “strand” and the barbs are grown along the spine. The feather is then instanced across the eagle’s body.

The animation called for deformation of individual feathers as the eagle flies, walks, and moves around, which meant a rig was needed for several feathers. In addition, strands will bind only to geometry in Unreal Engine. This led us to create a strip of instanced geometry for each feather spine, which we rigged in Maya. The barbs were imported to Unreal Engine as fur and bound to the geometry spines.

The eagle feathers required manual placement to ensure a specific structure and appearance as well as to meet rigging requirements.

Figure 35: Sketching rami placement

For feathers, the timing of the release of pigments along the rachis (spine) and then onto the rami (branches off the spine) is responsible for the seemingly complex patterns on feathers. The shader for the feathers follows the same principles of spatial versus timing zones as the hair strands for the meerkat. The main difference is the extra complexity of the orientation and position of the rami along the rachis. The final patterns seen on feathers are tightly dependent on the geometry of the feather, meaning careful attention needs to be given to their construction because they affect the final look more profoundly.

Originally, the shader for the feather was a projection of a color map painted on a flat polygonal plane in Substance Painter. There were 13 different types of feathers for the whole body of the martial eagle. These maps were then remapped using the UV coordinates of both the rachis and the rami.

Unfortunately, this immediately presented problems in getting natural variation in the patterns amongst the feathers, and limiting the ability to control the hue and value in a quick and efficient way. A more procedural approach of using the principles of pigmentation along the strands gave a much more organic look.

Figure 36: View of the martial eagle in the final real-time rendering

As discussed in the Import process section, the texture maps were generated using the skin's UV space, defining the color throughout the entire length of hair strands using the **rootUV** attribute. For the meerkat fur this was straightforward, but for feathers the **rootUV** parameter could only define the entirety of a single feather.

In order to achieve specific patterning, we required extra attributes for the hair strands: one to define the rami position along the rachis (0 at the base, 1 at the tip) and a second attribute to specify which side of the feather it was (medial versus lateral vane). Color ramps were used to create the patterns of color/value along each of the hair strands in order to control the banding along the strands.

Conclusion

The real-time hair and fur system is still fairly new, and there's lots of room to grow.

In future releases of the hair and fur system, our objective at Epic Games is to focus on quality and performance. For shading and rendering of hair/fur, we will be improving approximations of light by offering extra parametrization and better lighting response.

We will also aim to improve collision response and materials, and further optimize performance to handle larger and denser grooms.

We look forward to adding more features to future releases of Unreal Engine to advance the cutting edge of real-time hair and fur.

Resources for further study

Formulas

Bidirectional Reflectance Distribution Function (BRDF)

Radial Basis Function (RBF)

Technical papers

Light Scattering from Human Hair Fibers

Physically Based Hair Shading in Unreal

Dual Scattering Approximation

Deep Opacity Maps

Position and Orientation Based Cosserat Rods

Artistic Simulation of Curly Hair

Related software

MetaHuman Creator

About this document

Authors

Gaelle Morand

Charles de Rousiers

Michael Forot

Contributors

Shawn Dunn

Nathan Farquhar

Darrin Wehser

Marija Blašković

Nikola Milosavljević

Editorial

Editor: Michele Bousquet

Layout: Jung Kwak