Furry, Floppy, Lovable
Once Upon a Monster's
Fur Pipeline

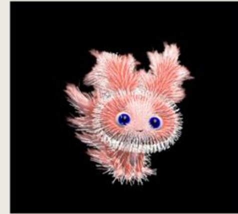
Pete Demoreuille
Lydia Choy
Oliver Franzke

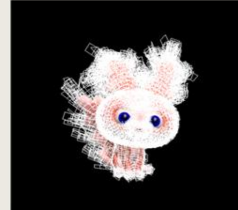
- Fur Pipeline developed at DF for Sesame Street: Once Upon a Monster

(Introduce ourselves and also special thanks to Tim Schafer, and to Raymond Crook for modelling and animation used this talk)

Motivations

- High-ish fidelity (for games) and flexible
  - Used on all characters
  - One (shared) artist for all fur
    - 17 characters
    - 166 total fur layouts!
- GPU: XBOX 360 (8 year old DX9 part)
  - 5-8 ms budget for *all* characters
  - Prefer CPU work over GPU work (!)
  - Modern approach would be all-GPU





Our basic motivations were to make a system that could produce pretty decent quality fur, but most importantly that this system be flexible and easy to use from an authoring standpoint. We had many characters using fur, and very little art time to do all the authoring.

The other large restriction placed on us was the fairly old GPU in the Xbox-360. This was also a Kinect title, which meant it would also take a chunk of our frame time, and we needed to keep everything very low-latency (as little buffering as possible to reduce input latency). The GPU time restrictions ended up ruling out many of the more modern approaches used when rendering and shadowing fur, and forces us to do GPU-suited operations like skinning, sorting and tessellation on the CPU. We believe that none of these things would be necessary on a modern GPU (and in fact, most of the algorithms would be very well suited for GPU implementation).

One might assume we went with shells and fins (often seen in games) for our fur. We did not.

We simulate, sort and render thousands of individual “fur strands”.

Motivations

- No Shells and Fins?
  - Inherited code from Brutal Legend
  - Though we could improve visual quality

Why did we not go with shells and fins?

We had a fur system inherited from Brutal Legend, which was developed under some similar and some very different constraints. Most differently, the old system had to be very cheap to render, as there could be many furry characters on screen at once. But it also needed to be high-quality and suitable for "hero" shots, as the player had to ride the beast in several missions. It turns out this ability to LOD, even in a title when there were far fewer characters rendered at once, would be fairly useful. And even though we were rendering fewer characters for Once Upon a Monster, they were generally a lot larger on screen, so their fur needed to be a lot higher in detail than the Brutal Legend characters. All that said, we thought it would be worth trying to increase the quality of the system to that needed by Once Upon a Monster and save the time building a new system (as with artist time, there was very limited programmer time).

One of our major motivations when examining what approach to take for for OUaM was to ensure we could faithfully represent Sesame Workshop characters.

We suspected they would be fairly particular about how their characters were represented on-screen, and knew we couldn't afford to custom-author individual fur geometry for each character in the game, mainly due to our small team size. Our solution needed to be semi-automated, mainly due to that bandwidth constraint, but also to make it easier to handle performance optimizations later down the road.

They gave us a lot of reference material, and there was a lot of back and forth iteration between us and Sesame Workshop. For example, early versions of Elmo matched the more haphazard fur configuration that he had in earlier Sesame Street episodes, but then later we were told that Kevin Clash, the puppeteer behind Elmo, liked grooming the fur on Elmo's head and face in a specific, slicked back way in later episodes, so we needed a system that gave us finer control.

We also made the mistake of telling our concept artists that we could do some amount of fur.

Which ended up with every single one of the concept characters being covered in fur.

Marco concepts + in-game shot.

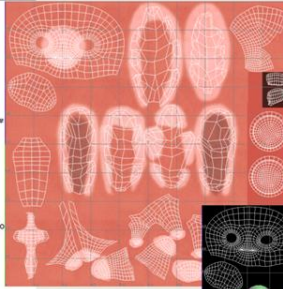
Tallulah concept and in-game shot.

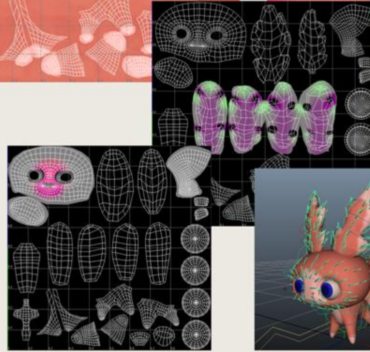
Seamus concepts and in-game shot.

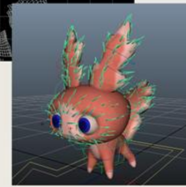
Additional monsters, in-game shots.

Authoring Pipeline

```
FurSetup
{
    Mesh=@Characters/Puffalope/Rig/Puffalope;
    # Subset = 1;
    FurMaterials = [@Characters/Puffalope/Materials/Puffalope.Mtrl];

    # for debugging
    #DebugRender = true;
    #DebugJoints = true;

    Layers = [
        ################################################
        ## BODY appearance params
        ##
        FurSetup::Layer {
            RandomSeed = 6;
            Enabled=1;

            SortBias = 1;

            Material=@Characters/Puffalope/Fur/PuffalopeFurLayer0.mtrl;
            DensityTexture=@Characters/Puffalope/Fur/Textures/Density_Bo
            CombTexture=@Characters/Puffalope/Fur/Textures/FurComb0;

            # This is the distance of the fur from the sk
            # first at the st
            Offset=0.000;

            # These are th
            Length=.04;
```







Basic fur is made up of a few pieces of data:

- Data Files
- Control Maps
- Maya scripts
- Shader Textures and Parameters

Authoring: Layers

- Authored by spatial region: "fur layer"
  - Generally spatial regions
  - Share defaults and shader properties
  - All defaults set per-layer, overridden with textures
    - Density
    - Length, Width
    - Surface Offset
    - Curl, Twist

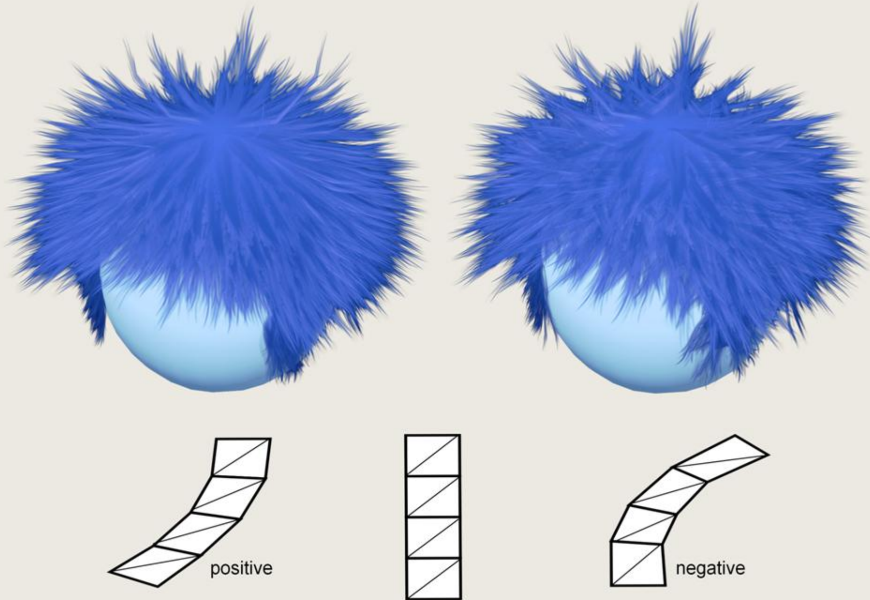
- Fur is authored as a collection of several distinct layers
- Generally these correspond to spatial regions on a character, both to aid performance and to group collections of fur strands that have a similar appearance.
- Each fur layer has a list of global parameters that drive anything from the appearance of the fur to its simulation properties
- Each layer also has a set of control maps whose different color channels modulate specific parameters

-Each fur layer (the top of the head, the eyebrows, and the mutton chops) has a list of default values for properties such as the overall density of fur strands, the width and length of each strand (with a variance), and the position offset from the underlying mesh surface.

- top texture corresponds to top of head, different channels modulate the default values set in the data file
- red channel scales the density value---fully red scales that region to twice the default density value, fully black scales that value to 0
- blue channel scales length, green channel scales width, and so on

Once the general size and density of the fur is set, additional layer properties provide controls over how the fur conforms to the character's body.

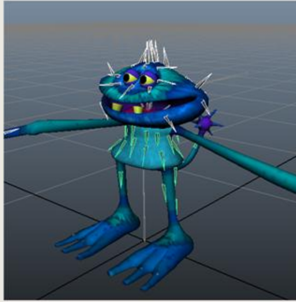
- strands are composed of 3 segments
- Curl value controls the polynomial that defines how the curvature is applied along these segments each strand.
- Positive values increase the amount that the fur curls "towards" the surface, and negative values push it away.
- This also allows the fur to better fit the shape of the character and prevents many of the strands from intersecting the body.



- To allow for more gnarled, matted fur, twist provides controls how each strand rotates in the plane of the normal to the surface.
- Both curl and twist have variance range values, and also control map channels that scale the default value

Authoring: Combing

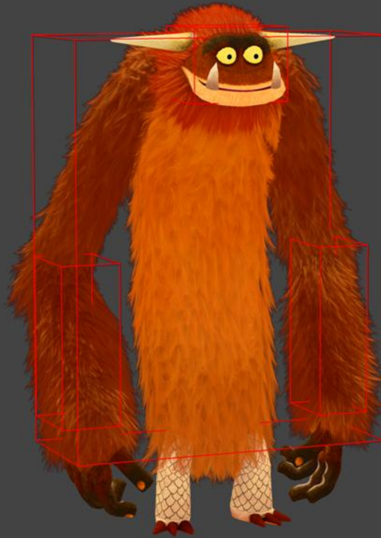
Vector field markers
determine strand direction



After setting global layer properties, we move onto defining the direction of the fur strands

- didn't have time to author the direction and placement of each individual strand (had 1 to 2 working days per character from start to final version, on avg)
- tried a bunch of different things (global direction per layer, joint-aligned directional vectors per layer), but they all didn't give fine enough control, esp for Sesame characters
- Ended up with a method of defining comb vectors, on the mesh surface in Maya.
- Strands are interpolated using both the relative position and normal of the strand. The normal is especially important for regions where surface direction changes quickly, such as armpits and shoulders
- Strands outside of the set distance range would default to a joint direction, so we only needed to add comb vectors where we needed finer control

Authoring: Layer Sorting



Head
Sort Bias 0

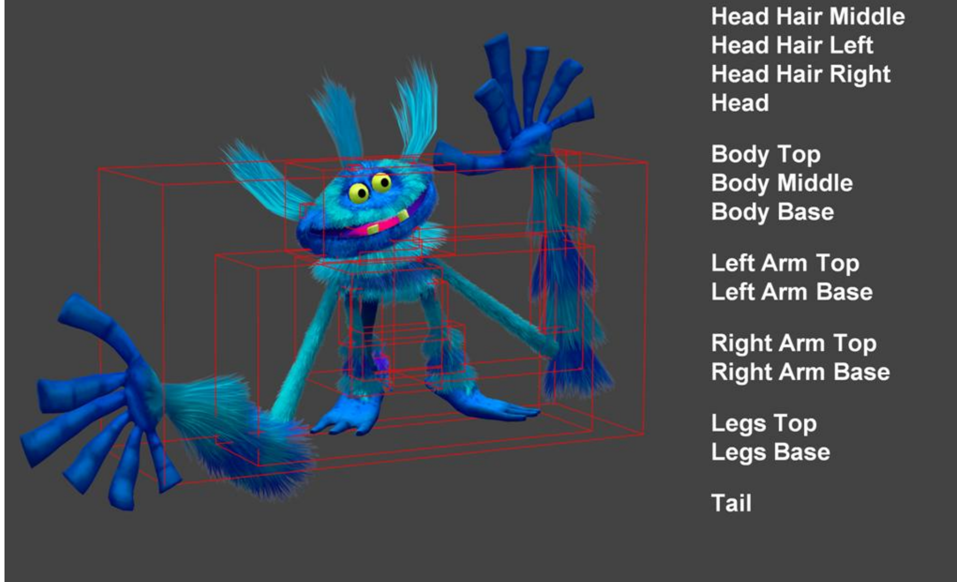
Left Arm
Sort Bias 1

Right Arm
Sort Bias 1

Body
Sort Bias 2

- For performance and artistic reasons, only fur strands within a layer are sorted against each other. Otherwise, layers are bounding box sorted--we also had control over the sort bias of each fur layer

In this example you can see the bounding boxes of the 4 layers, in addition to a sort bias value for each layer (the head, the body, and the forearms)



- It was very easy to organize and control the placement of each sort layer, and it made it possible for more complicated layouts such as this one

Authoring: Appearance Textures

- Base Color

- After authoring the overall layout, comb, and length/width properties of the fur, we now need to author the appearance of each strand of fur
- The fur strand shader has the following inputs:

- Diffuse color of the strand is sampled from underlying mesh diffuse texture (or some simplified version of it)
- Overall tint and specular values

Authoring: Appearance Textures

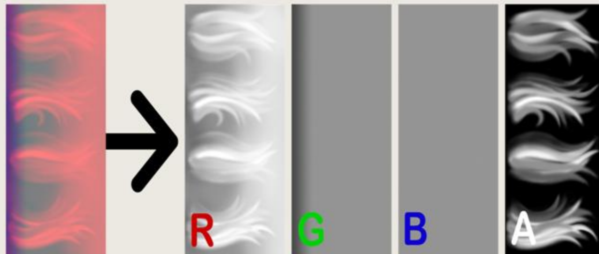
- Base Color
- Strand Texture
- Strand Variation

- After authoring the layout, comb, and length/width properties of the fur, we now need to author the appearance of each strand of fur
- The fur strand shader has the following inputs:

- Diffuse color of the strand is sampled from underlying mesh diffuse texture (or some simplified version of it)
- Overall tint and specular values
- Also a texture for the fur strand itself, whose various channels define alpha, and additional tint and specular
- Film strip texture varies the silhouette of each strand (picked randomly for each strand0

Authoring: Appearance Textures

- Base Color
- Strand Texture
- Strand Variation
- Shader Inputs
  - Specular
  - Shadowing
  - “Misc”
  - Alpha



- The fur strand texture encodeds a number of properties for the shader, including alpha, and additional tint and specular.

Here are some sample alpha channels of fur strand textures so you can see that there can be a lot of variety--the second one from the left was for Oscar the Grouch, and was extracted directly from high res photographs of Oscar.

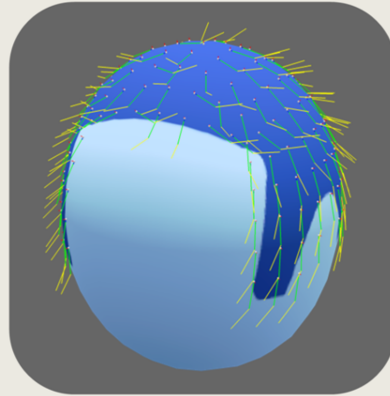
Because the system ended up being really flexible, we used it in some unexpected ways, such as making the fur on this character look like it was covered in mud, dirt and leaves. For this activity we also had a gpu painting system that would allow the player to dynamically clean the muddy fur off of the character.

And here we made Elmo and Cookie's fur look like it had been dusted with snow

Authoring: Simulation

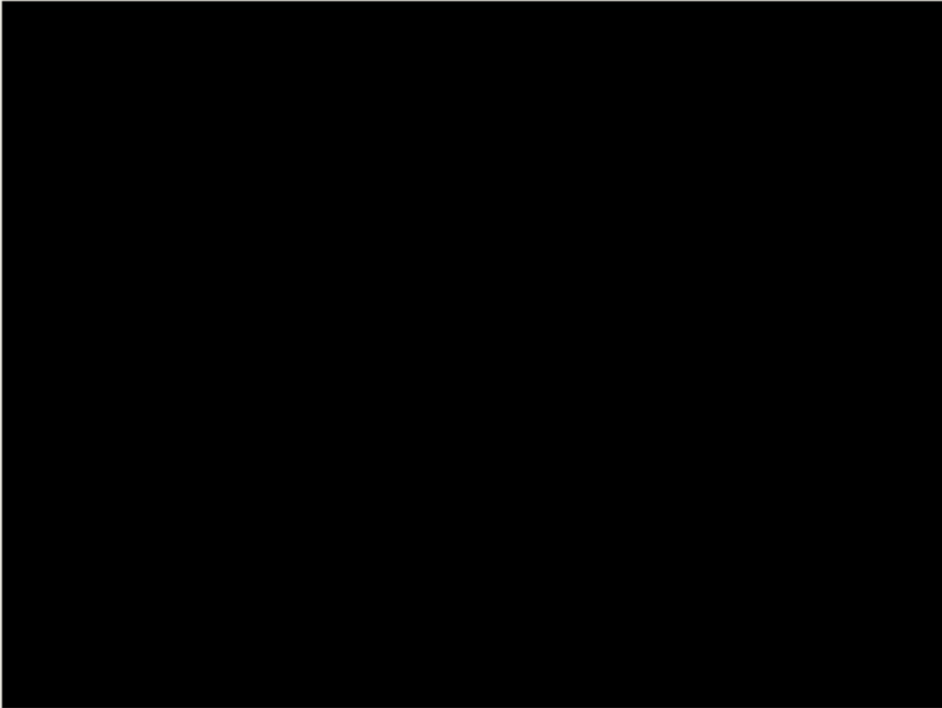
Many controls for simulation:

- Velocity Damping
- Motion Damping
- Segment Length
- Mid Angular Hardness
- End Angular Hardness
- Surface Hardness



Details to come

- Simulation for the fur is done with a 3-particle chain at each vertex
- Particles are not allowed to move beneath the skin, and are not allowed to move more or less than a certain amount apart
- As an artist, for each fur layer I have control over the length of the particle chain, the stiffness of each segment, and also how much secondary motion, or damping, is added based on the velocity of the particle when the character is animating or external forces like wind are applied



- Here is an example of the particle simulation layered on the character animation (Colored lines are the particle chains, while white lines are the interpolated strands)
- And here is the simulation with final rendering
- thanks to Ray Crook for the test rig and animation used in this talk
- And now Pete will go into more detail about the offline and runtime process

Offline Preprocess

- Preprocess data for runtime
  - Iteration time 10-15 sec max
- Basic process for each layer:
  a. Use density texture to determine strand locations
     - Rejection sampling to distribute points
  b. Sample comb data to determine primary direction
  c. Sample additional maps to generate curl/twist/size
  d. Pick shading params (filmstrip texture index, etc)
  e. Extract mesh vertices used for each strand

The offline preprocess uses the layer definitions, layout textures, and combing vector field to generate a list of strands and strand properties.

The most important bits of this are the various ways data is packed and split up so it can be processed in parallel by several threads, sorted efficiently, and vertices skinned in software as quickly as possible.

Offline Preprocess: Data

- Per-strand data
  - Position, direction, width, length, offset
  - Barycentric uvs, mesh verts
  - Simulation stength, curl, twist, random indices
- Per-vertex data
  - position, normal, skinning data
- Per-Layer data
  - The kitchen sink

- Compact data per strand, per vertex and per layer

NEXT: runtime

Runtime: Overview

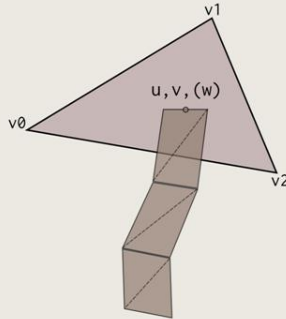
1. Pose mesh vertices
2. Compute level of detail
3. Update strand positions and velocities
4. Simulation update
5. Generate sorted strand indices
6. VB, IB generation and rendering

- Roughly reverse of preprocess

Runtime: Mesh Vertex Update

Skin all mesh vertices used by strands

- Strand stores `[u,v]` and `[v0,v1,v2]`
- Used to compute root position and normal
  - Also store frame-to-frame motion



In order to update all the strand positions, we first skin the vertices each strand are constrained to, updating both position and normals.
From this, each strand uses the vertex indices and barycentric coordinates to find the position of their root, posed normal and tangent.

Shows the mesh vertices and simulation chains.

The mesh vertices don’t need to be extracted from the highest level LOD of the character – they very often be produced from a lower-level LOD without any visual change, reducing memory, skinning and simulation costs.

Runtime: LOD

Utilize distribution of strands in buffer

- Strands added uniformly w.r.t. density
  - Progressively smaller distance between points
- At runtime, drop last N% of strand buffer
  - Allows nearly continuous LOD
  - Extremely cheap, requires no more memory
  - Optionally fade out last few strands

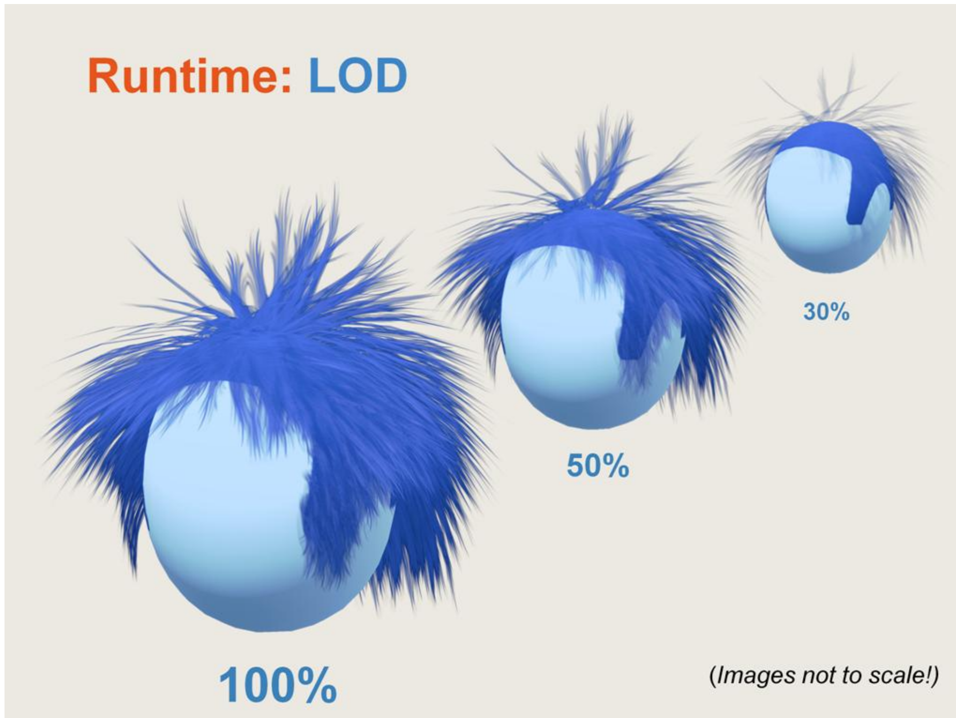
LOD Strand Buffer

Before each of the individual strands are updated, we determine how much LOD should be applied when updating the character and rendering. This can use any technique (screen size, budget, etc) and will just tell us approximately how many strands should be rendered.

Because we generate the strands in such a way that it progressively covers the character's surface (where strands later in the buffer are relatively closer to their neighbors than strands earlier in the buffer), we can progressively drop the last N strands from the update to perform LOD. This prevents them from being rendered (the update generates an buffer of indices and sort keys that is used when generating an index and vertex buffer).

This gives us nearly completely continuous LOD, is extremely cheap, and requires no additional memory (it does cause some of the underlying updates to run a little slower due to less coherent access to simulation vertices, but that was acceptable for us). If the occasional popping out of indivudal strands is a problem, you can fade out the last few rendered in the buffer.

The ability to very cheaply pick a subset of the strands to render is also very helpful when rendering shadows.



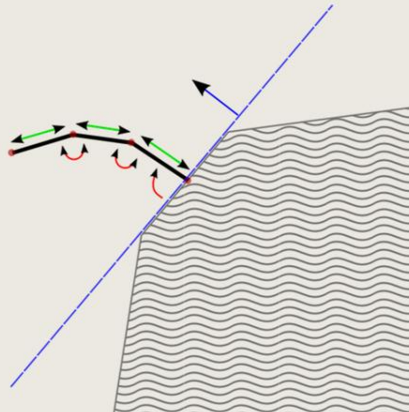
-Exaggerated example -- coverage remains pretty uniform when we reduce the number of strands rendered. (images not to scale)

Also a good example why your lower mips should have their alpha changed so everything doesn’t go too transparent.

Runtime: Simulation

Capture frame-to-frame movement of root

- Feed to particle simulation
  - Distance constraint
  - Angular constraint
  - Plane constraint (collision)



Customizable

- Per-layer strengths
- Per-strand damping

Once we have determined how many strands to update and determined the root position and normal of each strand, we update and interpolate the particle simulation performed at each vertex.

Three basic constraints suffice to give fairly believable motion, provide some approximation of fur-to-body collision, and give enough flexibility that the various fur styles can be modeled.
The simulation uses one fixed and three free particles
- each has a distance constraint between subsequent particles in the chain
- an angular constraint that limits the angle between subsequent segments (and the first segment + normal)
- and a plane constraint that prevents the chain from coming too close to the body

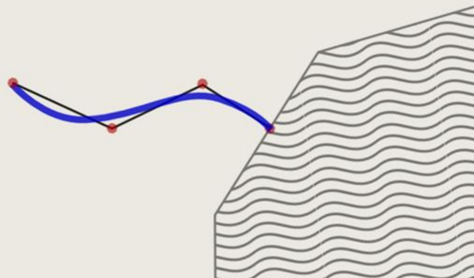
Each of these has customizable parameters for the entire layer, and the artist can specify different strengths for each particle in the chain.
In addition, strands have an "simulation" map which determines how much the simulation is allowed to influence the motion of the strand.

This gives really flexible results, but the one thing they are not is very high resolution. We need to fix that before rendering.

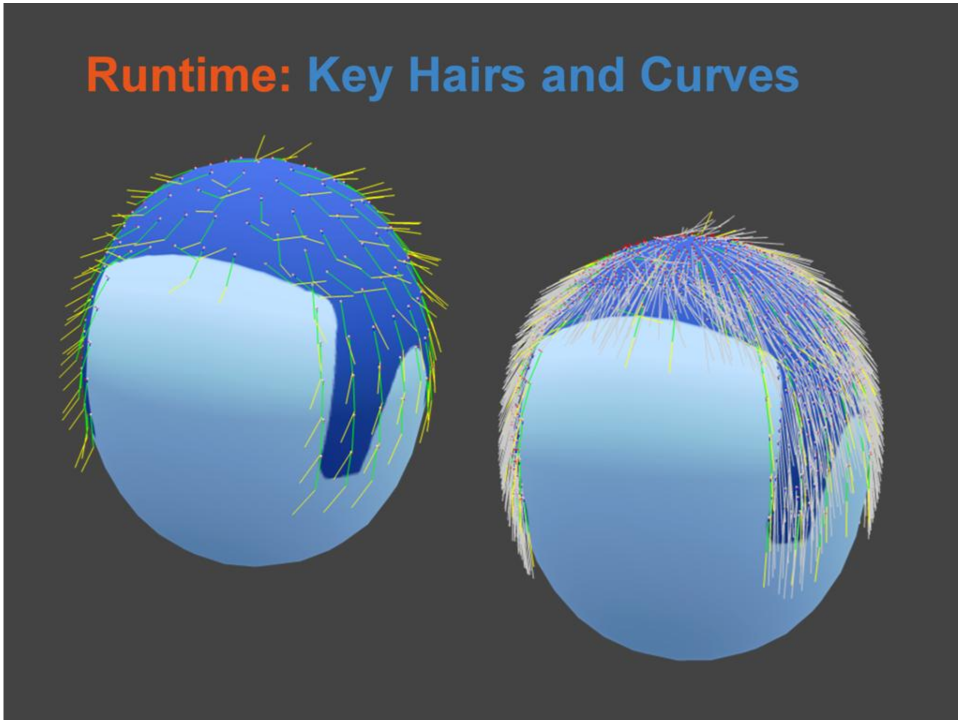
Runtime: Simulation

Compute curve coefficients from sim particles

- Used in VS to deform strip geometry
  - Per-strand scalar controls amount of deformation
- Overall fairly pleasing motion
  - Despite simple + low-resolution sim!



- Simple sim with 3 points can't be used directly
- Fit a curve to the simulation
- Pass coefficients to vertex shader
- Vertex shader uses curve to deform the strip geometry



- Left: raw keyhairs
- Right: interpolated strand curves

Runtime: Sorting

Need key to sort strand

- Perfect value not possible, but we try
- Hand-coded approximation of shader
  - Yields a single Z value per strand
  - Endlessly tweakable, best approx. depends on fur

Radix sort Z values, generate sorted indices

- Strand indices used to generate index buffer
- Sorting also used for backface culling

Making sure the strands render roughly back-to-front is critical for the fur transparency to look good and be stable when the camera or character moves.

To do so, during update of their root we compute a Z value for each strand that will be used when sorting (distance between the camera and a location on the strand). Choosing a reasonable Z value for the entire strand is somewhat finicky, we use a hand-coded approximation of the vertex shader to compute the position roughly 75% of the way down the strand (this is also quite expensive!), which is then used to determine a sorting key. This value and computation required a bit of tweaking, and given time, it would probably have been best to customize this per-character (and probably per-layer).

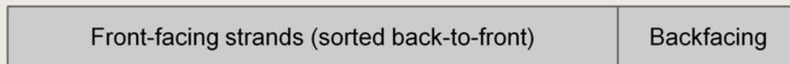
Once these values are computed, a fast radix sort is used to simultaneously sort them and permute a strand index buffer that will be used to generate the fur vertex buffer.

Nice thing about the sorting is that it lets you do some additional culling.

Runtime: Culling

Sort key is changed for backfacing strands

- Layers are tagged to cull backfacing or not
  - Expose tolerance to determine back-facing strand
- Keep track of front-facing count (N)
- Only render first N sorted strands



The sorting process is tweaked to allow us to simultaneously cull backfacing strands, saving CPU time generating the vertex buffer and GPU time rendering them.

Using a configurable tolerance value set per-layer, each strand is tested if the normal is backfacing. If so, it will generate a special sorting key to force it to the end of the sorted strands.

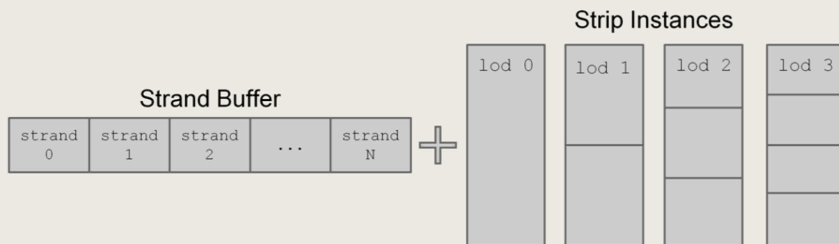
If we keep track of the number of front facing (or back facing) strands, this allows us to skip the last N backfacing strands, saving CPU and GPU time.

NEXT: RENDERING

Runtime: Rendering

Use sorted indices to generate vertex buffer

- XBOX vertex shader instancing used
    - Generate single "instance" vertex per strand
    - Use precomputed VB and IB based on tessellation
    - Shader deforms strip using simulation curve



Using the sorted strand index list, we generate a vertex buffer with one vertex per strand. This is used in conjunction with the Xbox's vertex shader instancing support to combine this data with pre-tesselated strip geometry. The instanced vertex data includes everything needed to deform the strip, generate UV coordinates, scale the strip, etc.

All that gives us this. And now all we need to do is make it look nice.

Like this.

Runtime: Lighting

Modified Kay-Kajiya cheapest and controllable

- Directional (shadowed) sun term
- Project point and area lights to SH
- Rim lighting term for highlights + darkening

Though we originally started to use Marschner et al's hair shading model (similar to the GPU Gems 2 article by Donnelly and Nguyen), we found Kay-Kajiya was the most flexible, artist friendly and cheapest solution that gave good results. Thorsten Scheuermann's presentation in GDC 2004 and in ShaderX3 also provided some good inspiration. We were lucky as the title's art and lighting style was very simple.

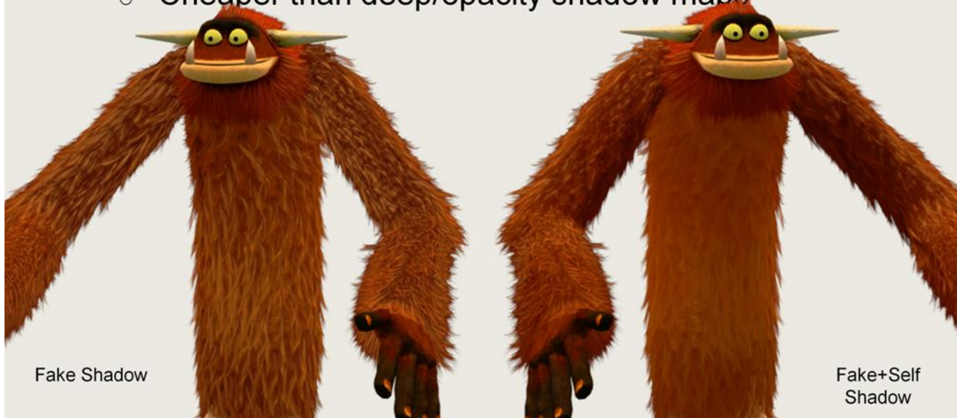
The fur lighting is fairly simple: all point and ambient light sources are combined into single band-4 spherical harmonics. Ambient sources were artist placed "irradiance" samples with two colors and a falloff radius which were interpolated as the characters moved. This allowed easy authoring of lighting gradients as the character moved in the scene. Lighting in the title is fairly uncomplicated so no gradients/more complex techniques were necessary to handle quickly varying lighting environments or large objects.

To help give a slightly backlit look, we also added an analytic (completely fake) rim lighting term that is configurable dynamically by the lighting environment. It also can be used with a "negative" lighting intensity to give a nice defined dark highlight around the edge of the fur (helps distinguish the silhouette of the character).

- Rim/Backlighting,
- Sun
- Subtle gradients
- "Scary" monster that needs a shower is dark + backlit

- Shadows make a huge difference in hair/fur
- Simple approaches add something, but not enough
- Fake shadow added based on distance between mesh and strand position (dot with normal)
-Exaggerated effect on right (too dark under chin)

-Important to bake the fake shadowing into the base texture or else your character skin will look too light.



To get better quality self-shadowing, we used a very simple process that gave us the "transparent-edged" shadows we were after. We couldn't use more complex approaches like deep shadowing/opacity shadow maps/etc, and they might not even be all that necessary as the fur was often not that thick.
-Render the strands approximately back-to-front into a small buffer, writing z
-Write Z to RGB, and set alpha to additively accumulate a "transparency" value
-A will not saturate to white in around the edges/lightly covered areas
-Use this plus the depth to give soft shadows that get darker the further from the front-most strand.

Very cheap and gave a nice looking shadow when using low-resolution maps: success.

Overview of system: start with mesh vertices, do simulation.

Overview of system: strand lines showing the interpolated and smoothed simulation

Wireframe

Base lighting

Final result.

DOUBLE FINE
pbd@pod6.org
lydia@doublefine.com