Innovations Report: Procedural Fur System

Joe Withers*
National Centre for Computer Animation

Abstract

For this project I developed a procedural fur system, with the aim of exploring the feasibility of offloading computation onto the GPU within artist tools. The final artefact is an application which serves to interface with the fur system API which I have developed. This report primarily documents the implementation of the API, as well as my findings.

1 Introduction

Generating and rendering hair and fur content for use in visual effects and games is typically a very computationally heavy task, due to the inherent complexity of handling thousands if not millions of geometric strands. Fortunately, modern hardware provides flexibility in regards to accelerating such tasks, which we are able to leverage though the use of hardware acceleration APIs such as OpenCL and CUDA.

1.1 Related work

Whilst many larger companies opt to develop their own software for handling fur, such as MPC's Furtility [Fagnou and Leaning 2010], there are a number of commercially available solutions that are also widely used in production.

The most popular of these is XGen [Thompson et al. 2003], originally developed by Walt Disney Animation Studios, which has been included in Autodesk's Maya since 2014. XGen uses a 'Collection' and 'Description' paradigm; Descriptions store user controlled parameters which affect fur primitives (curves for example) on a select area of a mesh, and Collections act as a container for these Descriptions. There are two primary methods offered for distributing primitives on a mesh, Guide Curves can be used to manually control the placement and shape of primitives by placing 'Guides' on the mesh surface, whereas 'Groomable Splines' randomly distributes splines over the mesh surface which can be manipulated in real time using viewport manipulators that mimic the brushing and styling of fur. In both modes, 'modifiers' can be added to the modifier stack to further adjust the visual outcome, adding features such as clumping, noise, and coiling. Attributes within these modifiers can be further controlled using expressions or texture maps.

Another commercially available solution is Yeti [Peregrine*labs 2012], which is also offered as a plugin within Autodesk's Maya. Yeti offers a more procedural approach, giving the user a node graph interface in which to construct the desired look from individual components. For example, the most simple usage requires the user to first create a 'scatter' node to distribute points on the mesh surface, and then combine it with a 'grow' node to extrude curves from these points. Whilst at first this may seem unintuitive when compared to XGen, it offers greater flexibility and avoids the pitfalls associated with a destructive workflow.

Offering even more flexibility, recent versions of SideFX's Houdini include hair and fur operator nodes, which create predefined node networks for the generation of hair and fur. These networks use standard Houdini nodes internally, so experienced Houdini users are intuitively able to create the visual outcome they desire, using tools they already know. However, it may be quite unintuitive for asset artists (those who would be using the tool, Groom artists?), who are not used to Houdini's heavily procedural workflow.

1.2 GPU acceleration

Whilst it is not clear to what extent each of these solutions make use of GPU acceleration, they all perform fast enough to allow the user to manipulate them interactively. However, software such as the HairWorks [NVIDIA 2014] API and TressFX [AMD 2014] library are developed primarily for use in games, so they are designed to utilise the GPU as much as possible. These solutions are able to perform both simulation and rendering of hair in real-time, providing interactivity that exceeds that of comparable software designed for VFX usage. It is worth noting that these tools do not provide a method for authoring hair and fur content, but instead designed to use load content generated by 3rd party authoring tools.

Figure 1: *NVIDIA's HairWorks API is used in a handful of modern games, such as The Witcher 3 [CD Projekt RED 2015], to provide realistic and interactive hair rendering.*

2 Implementation

From my research it was clear that there is an apparent lack of standalone software for authoring fur; it makes sense that existing fur software are developed as software plugins, as this allows the tool to make use of the host digital content creation (DCC) package's existing functionality.

However, by developing the system as a simple API one could make it DCC agnostic, which is why I chose to develop my system as a pairing of a simple API with a standalone tool which interfaces with it. By developing a simple API it could allow for a future implementation as a plugin within a DCC, providing the data exchange formats are compatible with the host.

2.1 Resources

I decided to develop the API side of the fur system using C++, primarily as it most commonly used when developing computationally heavy artist tools, but also because it is the language I am most comfortable using. I opted to use OpenGL over other APIs (CUDA,

*e-mail:joewithers96@gmail.com

OpenCL) for offloading of computation onto the GPU, simply because my final artefact is a tool with a graphical interface, and OpenGL is capable of handling both arbitrary computation using compute shaders and rendering of geometry simultaneously.

To handle the user interface I used the Qt framework within C++. Qt is commonly used for artist tools within visual effects as it is cross-platform, and applications can be configured to run within other applications that make use of it, such as Autodesk's Maya.

I wanted to include a node-graph style interface within my application, as these are commonly used within existing artist tools (Autodesk Maya, Unreal Engine), and would encourage modularity within my API. Qt does not natively provide this kind of interface, so I made use of NodeEditor [Pinaev 2017], an existing Qt-based library that provides this functionality.

2.2 Design

From looking at existing fur systems I was able determine that my simplified fur system would need to consist of the following components:

- Geometry Loaders - These are responsible for loading user specified geometry. Example Geometry Loaders could allow for the parsing of Wavefront OBJ files.
- Distributors - These are responsible for the distribution of curves onto user specified geometry. User controllable parameters could include density (curve count), distribution pattern (random, uniform), and curve length. These parameters could potentially be controlled by texture inputs.
- Operators - These are responsible for manipulation of the curves to achieve the desired look. Example operators could provide bending, clumping, or randomisation of input curves. These should behave much like the 'Modifiers' in XGen.
- Renderers - These are responsible for the rendering of curves into the application viewport. Example renderers could provide mesh, curves as lines, or curves as ribbons rendering functionality. User controllable parameters could provide controls for the shading model in use, as well as control of the base and tip widths when rendering curve ribbons for example.

Whilst the list of components specified above provide the required functionality for a standalone tool, they do not provide any functionality for exporting fur for use in a broader pipeline. I chose to omit this functionality to limit the scope of this assignment, but the ability to export hair curves as alembic would be desirable, as alembic is a widely adopted format that allows for the storage of multiple different types of curve geometry.

2.3 Development

In this section I will explain the development of each of the major components, not necessarily in chronological order, but in order regarding to how their data is passed through the system.

2.3.1 Mesh loading

The first problem I tackled was that of loading mesh geometry into the system, in such a way that would allow for later computation on both the GPU and CPU. I opted to implement a Wavefront OBJ loader as they are (deceptively) simple to parse, consisting of a list of vertex positions, vertex normals, vertex texture coordinates, and faces as defined by a list of indices. This data structure formed the basis of my 'Mesh' class. Using the NodeEditor library I then created a node which loads OBJ files, and outputs the 'Mesh' object through it's output data port.

2.3.2 Curve distributors

In order to generate curves for the system to manipulate, it was necessary to create a node that handles the distribution of curves onto the surface of a mesh. This node needs to take 'Mesh' object data as input, and output 'Curves' object data. A 'Curves' object was implemented which consists an array of 'Curve' objects, which in turn consist of an array 5 vertex positions that construct the curve. Distributor nodes would also need to alternate between performing the distribution process on the CPU or on the GPU, which would in turn affect whether the node outputs a 'Curves' object, or an ID for where the curves data is stored on the GPU.

With these parameters in mind I decided to implement an abstract distributor node class, which contains a virtual function for performing the distribution, and handles the callbacks for when input data is changed and the distribution needs to be updated. This makes the system somewhat extensible, allowing for different distribution patterns to be implemented with ease. The node's data type was also extended to be able to store either 'Curves' objects from the CPU, or an ID for an OpenGL buffer object, with an enumerated value to identify which of these is currently in use. This not only allows the distributor to alternate between CPU and GPU computation arbitrarily, but also allows nodes receiving this data to process it accordingly.

There are many different methods one could use to distribute points on an arbitrary mesh, however for replicating creature fur it is desirable to choose a method that distributes primitives at random positions across the surface, yet roughly evenly spaced from neighbouring primitives. A popular method used to achieve this kind of distribution is referred to as 'Dart Throwing', which relies on checking the distance to neighbouring samples after a new sample has been generated, and discarding the sample if the samples are too close. Whilst these kinds of methods produce good results, they can be inefficient as in certain situations many samples may be generated and rejected, before a sample is accepted. An optimisation for this is described in *Dart Throwing on Surfaces* [Cline et al. 2009], which relies on removing areas of the mesh that are already covered by a sample, by means of triangle subdivision.

Unfortunately, parallelising dart throwing methods is not trivial, making them unsuitable for my desired usage of OpenGL compute shaders. Whilst sampling methods suitable for parallel processing do exist, such as those described in *Parallel Poisson Disk Sampling with Spectrum Analysis on Surfaces* [Bowers et al. 2010], I instead opted to implement a naive random sampling method as it was much easier to parallelize.

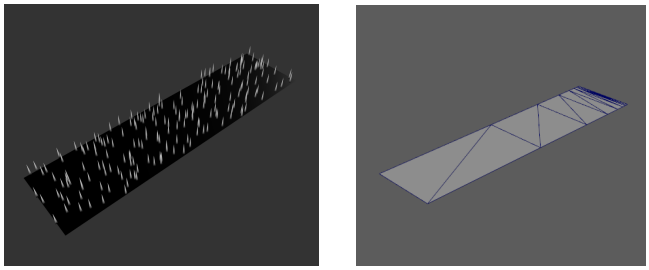

Figure 2: *Example distribution generated by my naive random distributor, note how the distribution stays relatively uniform despite variance in mesh density.*

2.3.3 Compute shaders

In order to get this distribution process running on the GPU, I needed to allocate and store data for both the Mesh and Curves in OpenGL buffers, for later use in a compute shader. The most common way of storing OBJ data in OpenGL buffers, as described in [Wikibooks Community 2011], is to use a `GL_ARRAY_BUFFER` object to store the vertex data and a `GL_ELEMENT_ARRAY_BUFFER` to store the indices of the `GL_ARRAY_BUFFER` that make up each face. However for ease of use in compute shaders I decided instead to store each face in a `GL_SHADER_STORAGE_BUFFER`, where each face consists of three sets of positions, normals, and texture coordinates. Whilst this is less efficient, I found that the overhead in GPU memory usage was negligible, and that this method allowed for much simpler usage in a compute shader. Allocating a buffer for the Curves data was much simpler as it consisted of an array of 'Curve' structs, each consisting of 5 vertices. Listing 1 shows how these buffers were accessed from within the compute shaders; note how all of the `vec3` values have been padded to `vec4`, as the `std430` layout qualifier requires an alignment of 16 bytes.

2.3.4 Curve operators

Once curve primitives have been generated, they need to be passed onto curve operators, which perform most of the work towards creating an appealing visual output. I decided to create an abstract curve operator node, much like I had with the distributor node, which contains a virtual function for performing the curve manipulation. This allows for multiple different curve operators to be integrated into the system with ease, however for the scope of this assignment I chose only to implement curve operators to provide bending and clumping functionality.

2.3.5 Rendering

By default, 'Mesh' and 'Curves' objects present in the node graph are not rendered in the viewport. For example, the user may want to distribute curves from hidden 'scalp' geometry, or use a set of hidden curves as clumping attractors. Therefore it was necessary to implement a pair of rendering nodes that would take a 'Mesh' or 'Curves' object as an input, and provide shading controls for rendering them in the viewport.

Whilst rendering and shading triangulated meshes is relatively straightforward, as described in [Wikibooks Community 2011], there are multiple different approaches one could use to render curves. The technique I chose to implement relies on drawing an empty VAO multiple times, and using `gl_PrimitiveIDIn` from within a geometry shader to determine which curve from the bound `GL_SHADER_STORAGE_BUFFER` should be drawn. Listings 2 and 3 shows how this was achieved. The use geometry shaders also allowed me to alternate between drawing the curves using line primitives, or to draw the curves as forward facing triangle strips.

For shading of the meshes I opted to implement a physically based shading model, as described in [Vries 2016], however some alterations were necessary as my method of rendering curves does not provide usable surface normal information. Instead the normalized direction in which the curve is leading, from one vertex to the next, was substituted as the surface normal. Then, all of the dot product terms in the shading calculations which use the surface normal, were inversed. The result roughly approximates the shading qualities of thin cylinders by assuming that all vectors that are perpendicular to the direction of the curve are valid surface normals. Whilst this technique provides visually pleasing results, it is inaccurate and does not account for properties such as self-shadowing, or light scattering.

2.4 User testing

Near the end of the development I was able to convince friend and fellow student Alin Bolcas to test the system and provide feedback from an artist's point of view; Alin is an experienced modelling and look development artist who uses a variety of different software packages to create his work, so getting his feedback on the usability of my tool proved extremely useful.

His first comments were in regards to how the node graph handles rendering of meshes and curves; he found it quite unintuitive to manually create rendering nodes for each new curves object that needed to be drawn. He suggested that a node for the combination of curves should be added, which would consolidate multiple curve objects into a single output for simplified rendering.

He noted that the lack of a node that applies noise to curves proved problematic during his testing; this kind of node is fundamental to achieving appealing fur visuals. He liked the behaviour of the clumping node, as this functionality is also very common and highly used within fur systems, though visual quality was somewhat compromised as my current implementation does not preserve the initial curve length. Adding a control in the distributor that would allow for the variation of curve length was also suggested as something that would improve the quality of the visual output.

His final comments were in regards to the user interface, suggesting that I make controls similar to their counterparts in existing software wherever possible. Changing the camera controls to match those in Maya was suggested as it would provide a familiar user experience when manipulating the viewport. The ability to create new nodes by pressing the tab key as opposed to right click was also suggested, as this is the standard in node editors, such as those in Maya and Nuke.

3 Results

After I had implemented some of the features Alin requested, I decided that the system was in a satisfactory state for the purposes of this assignment. The following Figures 3 and 4 show examples of the kind of visual output my system is capable of producing, and the accompanying node graph setup used to create them.

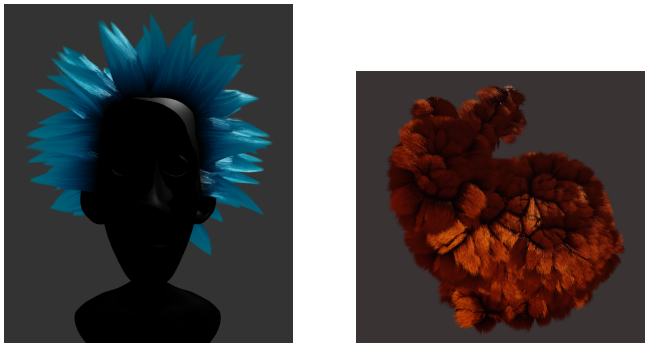

Figure 3: *Examples of visual output that my system is capable of producing.*

3.1 Performance

Figures 5 and 6 show that my system is performant enough to provide real-time feedback, even when dense meshes and high primitive counts are used. Whilst Figure 5 shows performance gains by

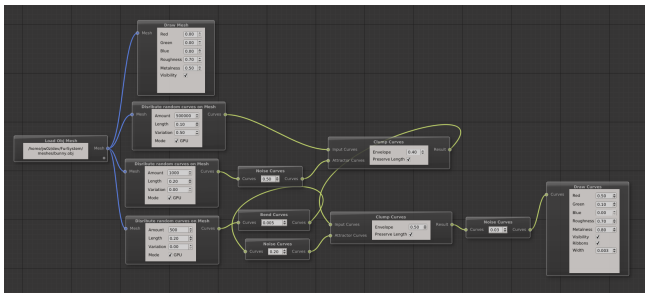

Figure 4: *The accompanying node graph setup used to create the images in Figure 3.*

utilising the GPU over the CPU, note that my CPU implementation is single-threaded, and further performance gains could be made by utilising multithreading.

It is worth noting that my system is quite inefficient in regards to GPU memory consumption; The node graph setup in Figure 4 uses 322 MiB of VRAM. This is a result of storing the state of each `GL_SHADER_STORAGE_BUFFER` in GPU memory, for each of the curve operators used. Whilst this avoids the need for a full graph evaluation each time a parameter is changed, it effectively doubles VRAM usage each time a curve operator is added.

Mesh Triangle Count	Primitive Count	CPU (ms)	GPU (ms)
16 (Triangle Strip)	10,000	2 - 5	1 - 2
-	100,000	18 - 41	8 - 24
-	1,000,000	120 - 150	77 - 110
4968 (Stanford Bunny)	10,000	173 - 211	1 - 2
-	100,000	1718 - 1756	12 - 30
-	1,000,000	17218 - 17259	82 - 112

Figure 5: *Performance metrics comparing time taken to distribute varying numbers of primitives, on meshes of varying densities, using the CPU and GPU. An Intel Core i7-6700 CPU and an NVIDIA GTX1080 GPU were used for testing.*

Mesh Triangle Count	Evaluation Time (ms)
16 (Triangle Strip)	69 - 84
4968 (Stanford Bunny)	81 - 112
14067 (Utah Teapot)	138 - 228

Figure 6: *Performance metrics showing the time taken to perform a full graph evaluation on a complex setup, as seen in Figure 4, for meshes of varying triangle counts. Note that over 500,000 primitives are being processed in this setup.*

4 Conclusion

Overall I am pleased with the visual quality and interactivity of my system; currently it provides the basic functionality required to visualise simple hair and fur content, whether it be for a VFX or real-time usage scenario. However, I believe further work is necessary in order to maximise it's performance and usability within a broader production pipeline.

4.1 Future Work

Unfortunately my system lacks the ability to export curves data, making it virtually unusable within a production pipeline in its current state. A viable file format for this could be *Alembic* [Sony Pictures Imageworks and Industrial Light and Magic 2011], as it natively supports the storage of parametric curves. Integrating alembic support into my system could also allow for animated mesh geometry to be used. Another candidate file format would be RenderMan's RIB format, which would allow for direct rendering of the hair curves in any RenderMan compliant renderer.

Another desirable addition to my system would be viewport manipulators, similar to the 'Groomable Splines' feature in XGen. This could allow me to implement techniques such as those described in Skippy [Krs et al. 2017], which would provide my system with a unique user experience compared to commercially available systems.

Whilst not necessary for usage in a offline pipeline, implementing a physically plausible hair shading model based on the techniques described in [Marschner et al. 2003], would greatly improve the visual accuracy of my viewport renderings.

References

AMD, 2014. Tressfx hair, real-time hair physics system. https://www.amd.com/en/technologies/tressfx. Accessed 18 Mar 2018.

BOWERS, J., WANG, R., WEI, L.-Y., AND MALETZ, D. 2010. Parallel poisson disk sampling with spectrum analysis on surfaces. *ACM Trans. Graph. 29*, 6 (Dec.), 166:1–166:10. Accessed 5 Feb 2018.

CD PROJEKT RED, 2015. The witcher 3: Wild hunt. Accessed 18 Mar 2018.

CLINE, D., JESCHKE, S., WHITE, K., RAZDAN, A., AND WONKA, P. 2009. Dart throwing on surfaces. 1217 – 1226. Accessed 5 Feb 2018.

FAGNOU, D., AND LEANING, J. 2010. Furtility: Dynamic grooming for wolfman. In *ACM SIGGRAPH 2010 Talks*, ACM, New York, NY, USA, SIGGRAPH '10, 51:1–51:1. Accessed 5 Feb 2018.

KRS, V., YUMER, E., CARR, N., BENES, B., AND MĚCH, R. 2017. Skippy: Single view 3d curve interactive modeling. *ACM Trans. Graph. 36*, 4 (July), 128:1–128:12. Accessed 5 Feb 2018.

MARSCHNER, S. R., JENSEN, H. W., CAMMARANO, M., WORLEY, S., AND HANRAHAN, P. 2003. Light scattering from human hair fibers. *ACM Trans. Graph. 22*, 3 (July), 780–791. Accessed 13 Mar 2018.

NVIDIA, 2014. Hairworks api. https://developer.nvidia.com/hairworks. Accessed 18 Mar 2018.

PEREGRINE*LABS, 2012. Yeti procedural fur toolset. https://github.com/paceholder/nodeeditor. Accessed 5 Feb 2018.

PINAEV, D., 2017. Qt5 node editor. https://github.com/paceholder/nodeeditor. Accessed 25 Jan 2018.

SONY PICTURES IMAGEWORKS AND INDUSTRIAL LIGHT AND MAGIC, 2011. Alembic format description. http://www.alembic.io/. Accessed 16 Jan 2017.

THOMPSON, II, T. V., PETTI, E. J., AND TAPPAN, C. 2003. Xgen: Arbitrary primitive generator. In *ACM SIGGRAPH 2003 Sketches &Amp; Applications*, ACM, New York, NY, USA, SIGGRAPH '03, 1–1. Accessed 5 Feb 2018.

VRIES, J. D., 2016. Pbr lighting. https://learnopengl.com/#!PBR/Lighting. Accessed 11 Oct 2017.

WIKIBOOKS COMMUNITY, 2011. Opengl programming/modern opengl tutorial load obj. https://en.wikibooks.org/wiki/OpenGL_Programming/Modern_OpenGL_Tutorial_Load_OBJ, Aug. Accessed 25 Jan 2018.

Listing 1: *Extract from the Distributor compute shader, showing how the Mesh and Curves buffers are accessed.*

```
struct Face
{
   vec4 position[3];
   vec4 normal[3];
   vec4 uv[3];{}
};

layout (std430, binding = 0) buffer facesBuffer
{
   Face faces[];
};

struct Curve
{
   vec4 position[5];
};

layout (std430, binding = 1) buffer curvesBuffer
{
   Curve curves[];
};
```

Listing 2: *Extract from the C++ application code showing how Curves are rendered procedurally.* `emptyVAO` *is an empty Vertex Array Object,* `curvesSSBO` *is a* `GL_SHADER_STORAGE_BUFFER` *containing vertex data for multiple curves, and* `indices` *is the number of curves in the buffer.*

```
glBindVertexArray(emptyVAO);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, ↩
    curvesSSBO);
glDrawArrays(GL_POINTS, 0, indices);
```

Listing 3: *Extract from the curves rendering geometry shader, showing how vertex positions were processed from data stored in a bound* `GL_SHADER_STORAGE_BUFFER`.

```
#version 430 core
layout(points) in;
layout(line_strip, max_vertices = 5) out;

struct Curve
{
   vec4 position[5];
};

layout (std430, binding = 0) buffer curvesBuffer
{
   Curve curves[];
};

uniform mat4 MVP;

void main()
{
   for (int i = 0; i < 5; ++i)
   {
      gl_Position = MVP * vec4(curves[↩
          gl_PrimitiveIDIn].position[i].xyz, ↩
          1.0);
      EmitVertex();
   }
   EndPrimitive();
}
```