Start to Finish Feathers Solution (SFFS)

Masters Thesis

Mark Newport

N.C.C.A Bournemouth University

26 August 2005

Contents

List of Figures

1. Introduction

The aim of this Thesis is to document the technical considerations of the Start to Finish Feathers Solution (SFFS). These considerations included, the research behind the subject matter and previous applications, the design choices, final implementations and the evaluation of the solution so far.

Computer-Generated(CG) feathers has long been an area of trepidation for 3D artists as they try to convey photorealistic feathers and a mixture of solutions have been attempted, with varying success, but it has now been established that if close-up high-detail shots of creatures with feathers are required then a physically accurate approach is the most effective solution. The reasons for this will be made clear in the Research section of this Thesis.

CG feathers are closely related to the concepts of hair and fur, both of which have been tackled to high degrees of success in terms of photorealism and work flow efficiency. Feathers seem to have proved a tougher challenge to researchers and visual effects companies, and few would say that this area has been successfully mastered yet. Research has tended to focus on specific areas of feathers and not as a whole, while few visual effects companies have produced both photorealistic and efficient work flow solutions for feathers, as Mike Milne, Senior 3D CG Supervisor at Framestore CFC, explained in a talk given at the NCCA, Bournemouth.

2. Research

2.a. The Structure of Feathers

To understand how feathers move and appear close-up, it is important to analyse the physical properties of a feather. Obviously feathers can vary greatly in appearance, but they are all made from a similar structure which can be seen in Figure 2.1.

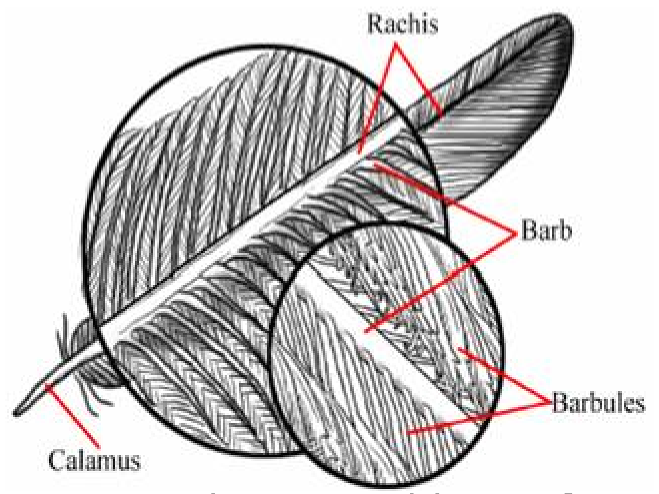


Figure 2.1 Feather Anatomy (Chen et al, 2002)

The rachis or shaft of the feather supports the vanes (i.e. the blades) of the feather. Within the vane of the feather, there are two lateral sets of barbs, interlocking the feather together. On both sides of a barb are many barbules. It is this structure that gives feathers their surface attributes that are recognised as feathers even though they can vary greatly from bird to bird.

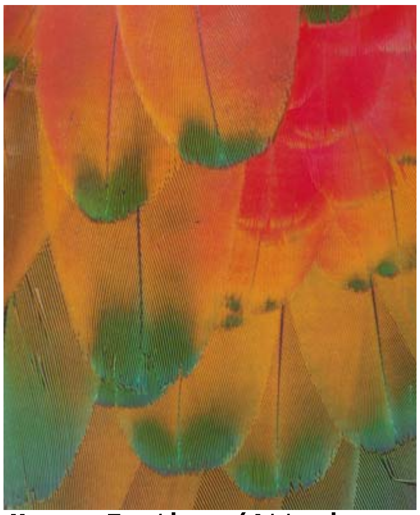


Figure 2.2 Macaw Feather (Attenborough.D, 1980)

Colour can be produced by structure. Some feather filaments have walls so thin they split light like an oil film and change colour with their angle. Some are filled with microscopic bubbles which reflect light. Their colour will disappear if they are doused with liquid and return when dry. Yet others, like the Macaw feather in Figure 2.2 contain pigments and retain their colour even when they are wet and viewed from any angle.

2.b. Previous Work: Computer-Generated Feathers

There is a small selection of research papers that relate to CG feathers. The majority tend to concentrate on realistic feather creation without consideration of applying to an animated geometry and all the problems this can create. Streit.L, and Heidrich.W, wrote a paper called 'A Biologically-Parameterized Feather Model', which suggests a control barb technique to set the structure of the feather and then uses interpolation to generate the rest of the feather detail. This meant they could generate hundreds of barbs on a feather in real-time while only storing the control vertices for a few key barbs.

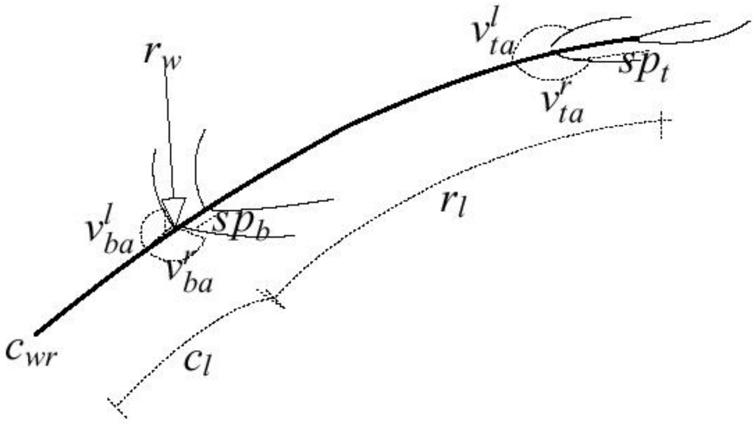
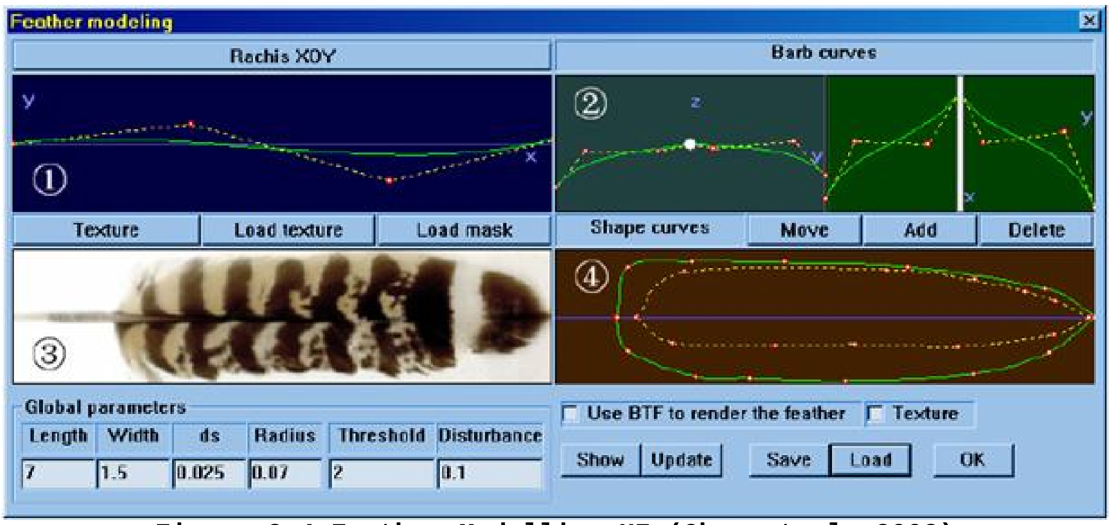


Figure 2.3 Illustration of Control Barb parameters (Streit & Heidrich 2002)

When designing the feather, only key barbs were specified to define the curvature of the vane. Any barb branching from the rachis between key barb locations is then interpolated from the key barbs found immediately above and below the branch location.

Like the majority of research papers on feathers Streit & Heidrich (2002) do not then investigate how these feathers can then be applied to an animated geometry. A paper that does take this into consideration is 'Modelling and Rendering of Realistic Feathers', Chen et al (2002). This paper tries to deal with every aspect of feathering a bird. They employ a parametric L-system modelling tool, which allows the user to generate feathers of different shapes by adjusting a few parameters.



Global parameters

Length	Width	ds	Radius	Threshold	Disturbance
7	1.5	0.025	0.07	2	0.1

Figure 2.4 Feather Modelling UI (Chen et al, 2002)

Chen et al (2002), go on to describe how they position the feathers on the geometry and how they solve the interpenetration between feathers, but this will be discussed in later sub-sections.

Framestore CFC have had several years of experience in creating feathered creatures, probably most famously for the Harry Potter films. Their last attempt was the Hippogriff character (Figure 2.5) in Harry Potter 3, which was half bird, half horse. Mike Milne, Senior 3D CG Supervisor, explained in a talk to the NCCA, Bournemouth, that although they had evolved their proprietary techniques and software, to create extremely realistic images, the inefficiency of the process, meant that Framestore CFC spent so long on the feather shots that they only broke even on the contract for Harry Potter 3. Many of the problems Framestore encountered were the same ones that seemed to keep appearing from the research conducted for this Thesis. The recurring problems are: how to position the feathers on a moving and deforming creature, stopping feathers interpenetrating each other, avoiding feathers flicking out from the body when the geometry creases and how all this can be achieved within user and resource limitations. These issues will be addressed in more detail in later sub-sections.

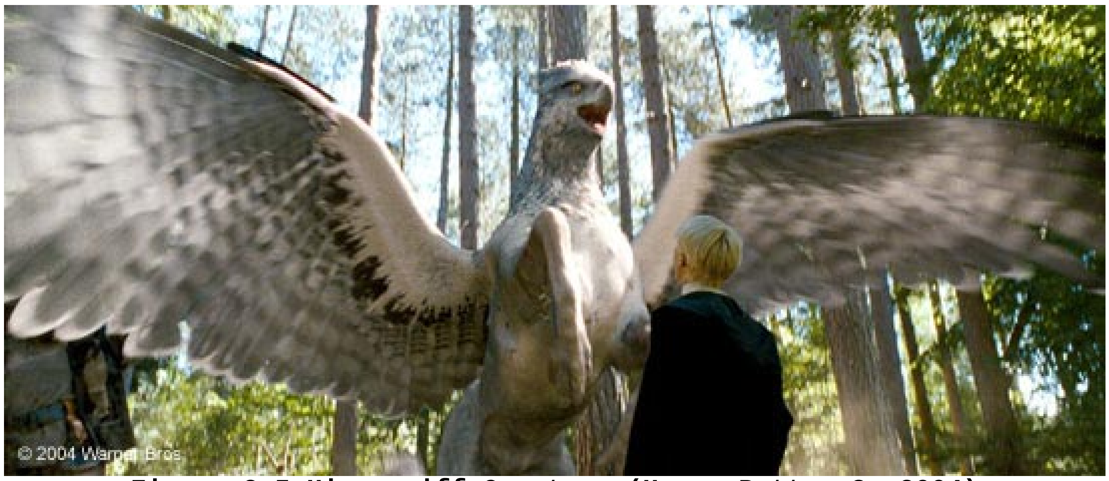


Figure 2.5 Hippogriff Creature (Harry Potter 3, 2004)

In 2000, Sony Imageworks created a feather R&D team, and its first task was to create feathers for the film, Stuart Little 2. The R&D team had eight months to develop an effective feathering process and solve all of the problems, mentioned above. As a dedicated team with plenty of time, they heavily researched the area and came up with improved and brand new techniques which solved some of the problems mentioned and helped to add realism to the characters' feathers. These techniques along with the others already mentioned earlier in this section will now be discussed in the context of the main problems encountered with feathers.

2.c. Specifying the Feather Positions

Unlike hair and fur, feathers do not tend to be spread out evenly over a surface. The areas around the head and neck of a bird tend to be more densely packed with small feathers than the wings where a few larger feathers cover the area. This means that applying a fairly straight forward even density distribution algorithm is no good. Chen et al (2002), employed Turk's algorithm, which is such an algorithm, but they adapted the technique to control vertex density based on the size of feathers.

The route taken by Sony Imageworks, was an adapted particle repulsion technique. Originally particle repulsion had be designed for even distribution over an implicit surface, but they needed to apply the technique to the skin of the bird, which was not described as an implicit surface, but rather as a series of connected NURBS patches.

Alias' Maya 3D Artisan Paint Tool provides a facility called Paint Scripts Tool. This gives the user the ability to paint a surface with an attribute defined in a Maya MEL script. For example, you can paint geometry or particle emitters on your surfaces like you would paint a colour or texture in a painting program. Maya Hair and Fur already uses this technique to apply to the surface geometry, because it allows the user to easily groom the creature quickly by painting the attributes onto the surface, rather than tweaking each individual feather by hand.

2.d. Solving Interpenetration

The human eye finds it very hard to make out an individual hair unless it's looking for one, but with feathers, it can definitely observe individual shapes. Therefore, it is easy for the audience to detect if feathers are interpenetrating each other, and thus destroy the illusion of the animation.

Chen et al (2002), used a recursive collision detection technique on simplified geometry to stop the interpenetration between feathers.

> “To determine the final growing directions, we need to perform collision detection on the feathers based on simplified geometry and to adjust the feather growing accordingly. Because of the large number of feathers and the complex shape of a bird’s body, a collision detection between every pair of feathers is likely to be very expensive. To address this problem we adopt 2 strategies. First, we grow feathers in an orderly fashion according to the initial growing directions. Second we only consider local collisions between neighbouring feathers because collisions rarely happen between feathers far away from each other. We implement these two strategies using a recursive collision detection algorithm.” (Chen et al, 2002)

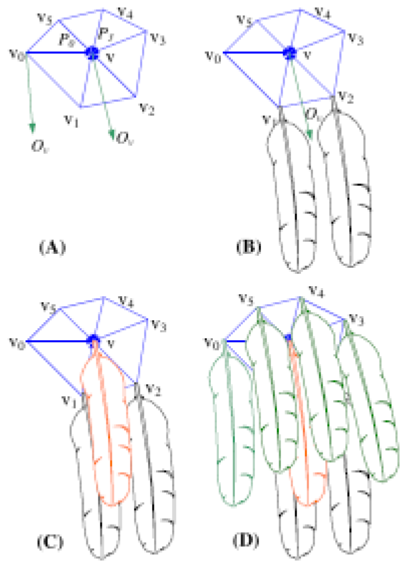


Figure 2.6 Recursive Collision for eliminating interpenetration (Chen et al 2002)

The solution Sony Imageworks originally came up with to solve interpenetration, was called one-dimensional volume deformation or 1DVD. 1DVD remembered, for all of the key curves, how far they were from the surface of the base model.

> “Starting with the bird in open pose, 1DVD remembers all the distance and the offsets beginning with a given curve and going to the next control vortex. Applied to the animation, the key curves are now ‘locked’ so that, as

> the skin moves, the feathers stay aligned with the movement of the skin.” (Armstrong E, et al 2002)

1DVD did create its own set of problems. Most specifically, when the bird turns its head, the skin should stretch diagonally across. They did not want the feathers to rotate when the neck was turned, but with 1DVD, all the key curves turn to align them selves with where they were. The other problem with 1DVD is that it only takes into account the centre of the feather. The curve only describes what happens to the shaft of the feather, not what happens on either side of the feather.

> “Software Developer Jeff Chan came up with 2DVD, or two-dimensional volume deformation. Whereas 1DVD controls the key curves and their offsets from the skin, 2DVD keeps track of every feather over the entire surface of that groomed bird, we can remember all the offsets of the individual CVs of the feathers including extra dimensions of the width of the feather, which allows us to keep them conformed to the skin, so they won’t rotate off.” (Armstrong E, et al, 2002)

With 2DVD when the bird turns its head, the feathers actually slide along maintaining their offset, but to different points on the surface, which means the feathers don’t twist.

2.e. Controlling the Motion

Although not a heavily documented problem, flicking feathers can plague some animations. Flicking feathers occur when the normals of the surface geometry change direction dramatically from frame to frame (Figure 2.7). This normally occurs around creases and joints in the creature’s skin. The feathers root position is normally dictated by the surface normal at that point and if it changes greatly then the feather appears to flick out from the body, destroying the illusion. Therefore some sort of dynamic feather is necessary in order to cope with these changes as opposed to a static feather.

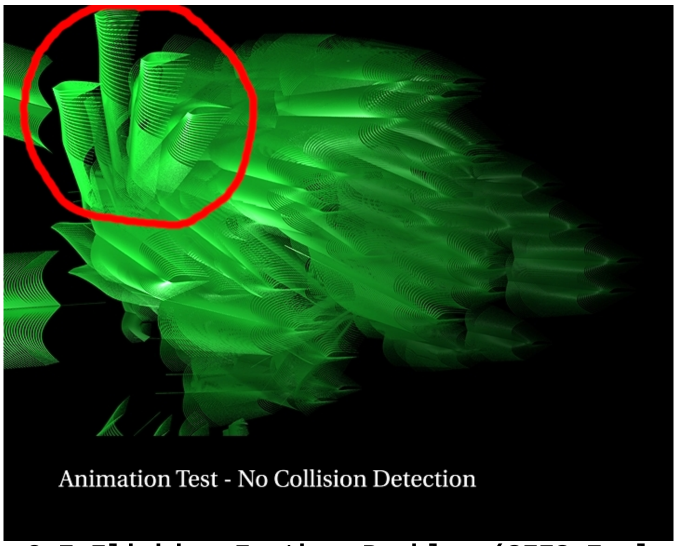


Figure 2.7 Flicking Feather Problem (SFFS Early Test)

Another important consideration is how the feathers react with the surrounding environment. For example, if the feathers are attached to a bird then there is a good chance it will be flying at some stage, so the feathers have to react to wind and other movement from the bird. If the feathers are applied without any dynamics then they will appear to be rigid and stuck on to the creature.

2.f. Work Flow and Resource Considerations

With feathers comes large amounts of geometry, and with large amounts of geometry comes hardware considerations. The 3D artist has to be able to groom the feathers in real-time and interact with the model in the viewport. Therefore it is important to leave as much of the geometry creation until the very last moment. When rendering large amounts of geometry a common problem is to run out of memory, so Pixar introduced a set of render-time primitive generation capabilities. Renderman procedural primitives are user-provided subroutines which manipulate private data structures containing geometric primitives that the renderer knows nothing about. By creating a Procedural Primitive DSO the feather detail can be created at render-time and based on bounding box data passed to the renderer. This means that render should be maximised for speed while dramatically reducing the likelihood of running out of RAM.

When the demand for hair and fur first started to appear in the visual effects market Pixar decide to create a

Curve primitive which was intended to be fast and efficient for use where hundreds of thousands or millions of thin stokes are required. As such, they are flat ribbons that are defined only by a single curve, which is their spine. This makes it a perfect choice of the feather detail as they can be varied in width along the spine.

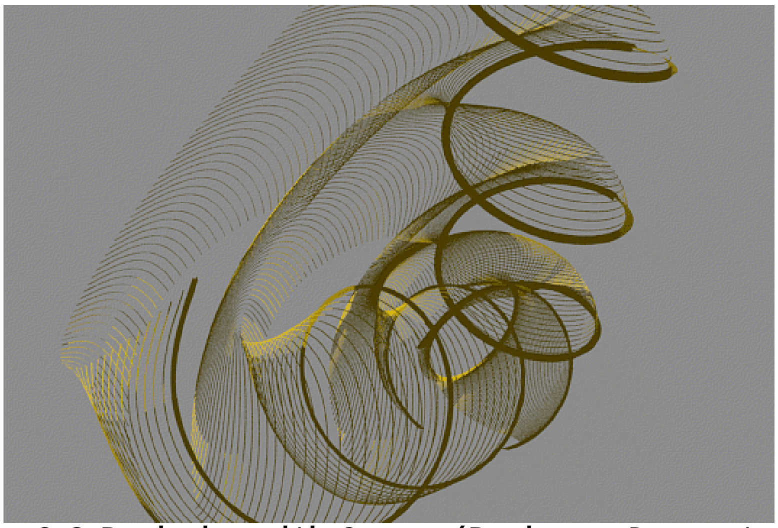


Figure 2.8 Rendering with Curves (Renderman Documentation)

2.g. Deep Shadow Maps

Feathers to some extent, but not as much as hair, gain their appearance from self-shadowing. To ray trace the shadows is still not really a practical solution for large amounts of frames, and traditional shadow maps produce rather harsh results, which look unrealistic. In 2000, Pixar realised a paper called Deep Shadows Maps. The technique they introduced produced fast, high quality shadows for primitives such as hair, fur and smoke. Unlike traditional shadow maps, which store a single depth at each pixel, deep shadow maps store a representation of the fractional visibility through a pixel at all possible depths.

> “Deep Shadow maps have several advantages. First, they are pre-filtered, which allows faster shadow lookups and much smaller memory footprints than regular shadow maps of similar quality. Second, they support shadows from partially transparent surfaces and volumetric objects such as fog. Third, they handle important cases of motion blur at no extra cost.” (Lokovic & Veach, 2000)

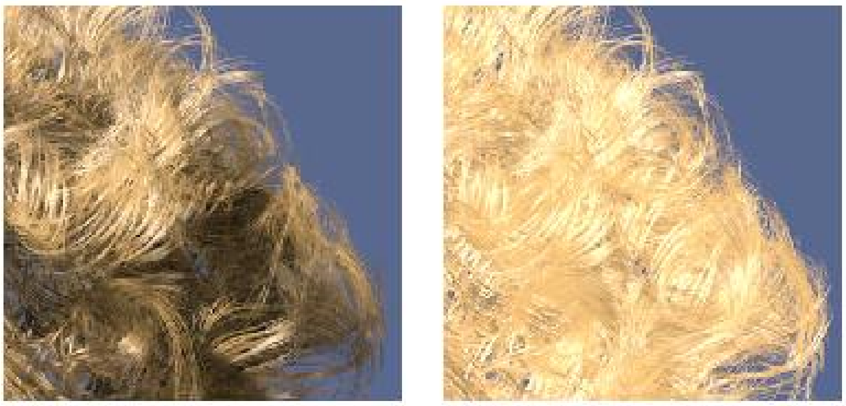


Figure 2.9 Hair Rendered with & without self-shadowing (Lokovic & Veach, 2000)

3. Overview of Solution

3.a. Pipeline Diagram

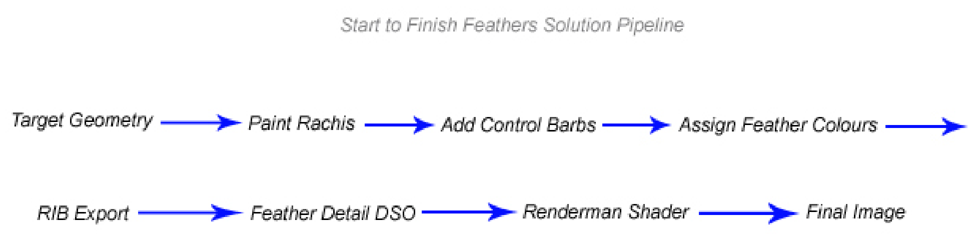


Figure 3.1 SFFS Pipeline Diagram

3.b. Design Choices: Intended Environment

The aim of the SFFS project was to create a photorealistic, efficient suite of tools/techniques that could be used within the visual effects industry in order to create feathered creatures. Like every visual effect, having control over the whole process is crucial to making it work.

The pipeline has been designed to be used with Alias' Maya and Pixar's Photorealistic Renderman (PRman). The reason being that they both provided good API's for software development and are both the primary tools used within the visual effects industry, so they were the natural choice to utilise. The solution was developed and tested on both the Linux Red Hat and Microsoft Windows operating systems, but could easily be adapted for different platforms and Renderman compliant renderers that support dynamic shared libraries.

One of the first considerations at the start of the feathering process, is the type of geometry that the feathers will be applied to. SFFS has been designed to work with both NURBS and Polygon models, as these are the most popular way of modelling at present, but sub-division surfaces has started to become increasingly used recently. The model itself does not necessarily need to be created within Maya, but as long as all the surface properties have been imported correctly then the process should work in exactly the same way.

The generation of the rachis and the barbs at different stages has two purposes. Firstly, it makes the grooming of the feathers with the paint tool interactive in real-time, whereas, tests with the barbs included, proved too slow to be workable. Secondly, once the barbs have been generated it can be quite hard to distinguish the direction of each individual feather, so grooming would then be much harder. Although the barb solving and collisions probably could have been implemented in MEL, the decision was taken to use the C++ API, based on the performance benefits when dealing with large amounts of geometry and testing loops.

Using a painted texture to define the colour of the feathers is not a new concept. It allows an artist to happily spend plenty of time painting the image separately while work on feathering the bird is on going, so therefore reducing the chance of a bottle neck in the pipeline.

As mentioned earlier in this section and based on the research conducted for this thesis, PRman, provided a suitable way of generating the feather detail at render-time and a way of applying a realistic surface property to the feathers via a shader. This is by no means the only way of achieving the final image, openGL being another possible solution, but for the reasons already stated the Renderman Interface was chosen.

4. Painting the Feathers

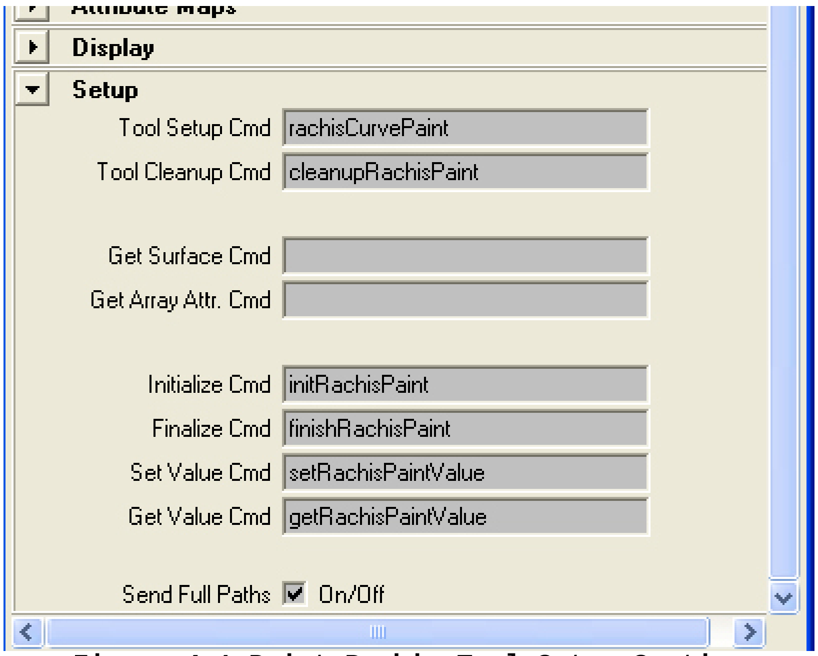
4.a. Maya’s Paint Scripts Tool

The Paint Scripts Tool is based on the concept of a two-dimensional array of numeric values super-imposed on a NURBS or polygonal surface. This array is either defined by the vertex positions of the surface or an arbitrary 2D grid that is evenly spaced in the surface parameter space. This array of numbers can be thought of as a 2D grayscale image, where each pixel corresponds to an array position and the grayscale value corresponds to the number associated with that array position. The scripts associated with the Paint Scripts Tool determine how Maya’s Artisan interprets this array of numbers. Artisan calls the script when it needs to know the number assigned to one of the array positions as described above. Artisan also calls the script when it changes a number assigned to one of the array positions.

The Paint Scripts Tool is defined by a set of MEL procedures. The names of the MEL procedures display in the Setup section of the Tool Settings editor. You can also set up the MEL procedures using the artUserPaintCtx MEL command, which means the paint tool will be instantly ready for the user to use without having to worry about entering all the individual procedures into the Setup section, therefore allowing a non technical artist to work the tool. SFFS achieves this using the script paintRachisTool.mel.

4.b. paintRachisTool

Since the rachis painting requirements are very similar to that of Maya’s hair system, it made sense to use the global hair paint context ($PaintHairAttrToolCtx). This sets the tool up with the necessary features to groom the rachis and the paintRachisTool script then directs the tool to the necessary procedures to paint the rachis onto the desired surface (see Appendix A.1.).



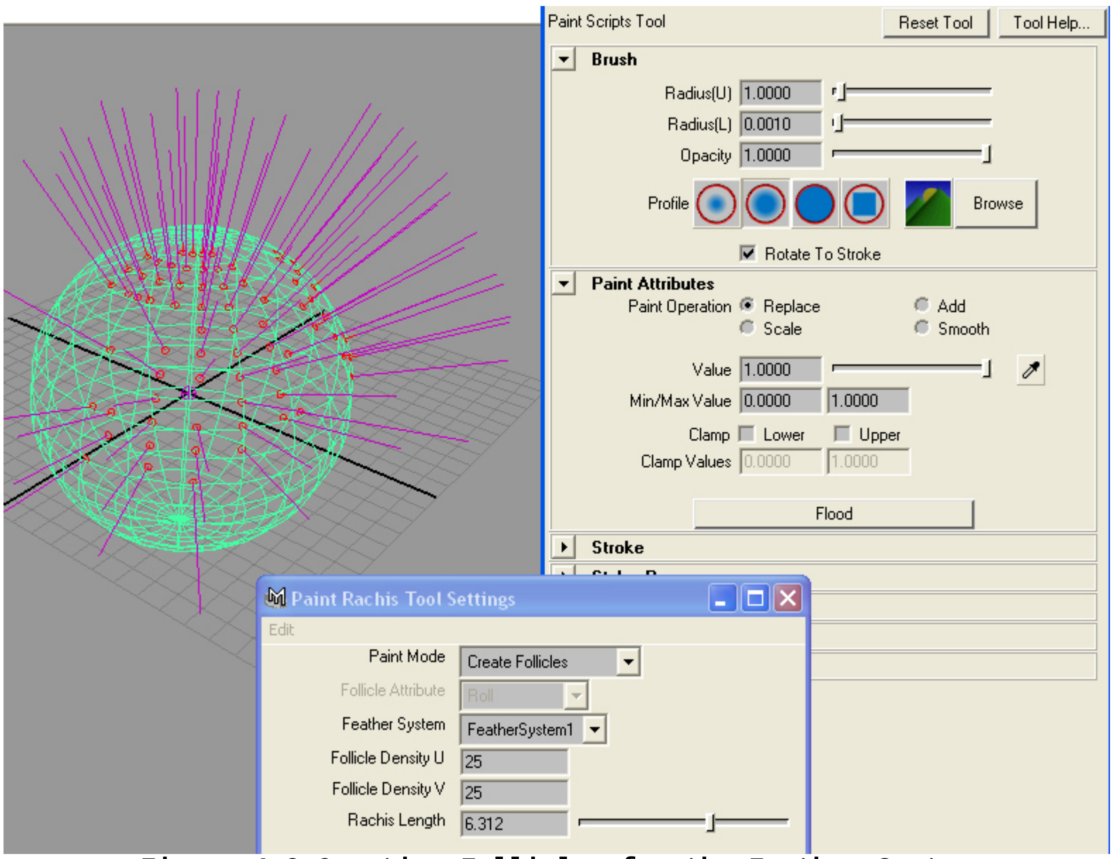
Tool Setup Cmd: rachisCurvePaint
Tool Cleanup Cmd: cleanupRachisPaint
Get Surface Cmd:
Get Array Attr. Cmd:
Initialize Cmd: initRachisPaint
Finalize Cmd: finishRachisPaint
Set Value Cmd: setRachisPaintValue
Get Value Cmd: getRachisPaintValue
Send Full Paths: [✓ On/Off]

Figure 4.1 Paint Rachis Tool Setup Section

4.c. rachisCurvePaint

The rachisCurvePaint script is the heart of the tool, which sets up the User Interface for the tool and contains the procedures to interact with Artisan to allow the painting process. It acts as both the creator of the feather systems and the grooming tool which is used to groom the feathers into the desired style. If there are no feather systems attached to the selected surface then the tool sets up to create a new feather system with the first stoke. The feather system uses Maya’s hair dynamics to control the movement of the rachis. This not only enables the feathers to move with inertia from the motion of the creature, but since it is particle driven it can be effected by external fields such as wind, gravity and turbulence (see Appendix A.2.).

Once a feather system has been created then the user has the choice of creating more follicles for that particular feather system or to start to a new feather system.



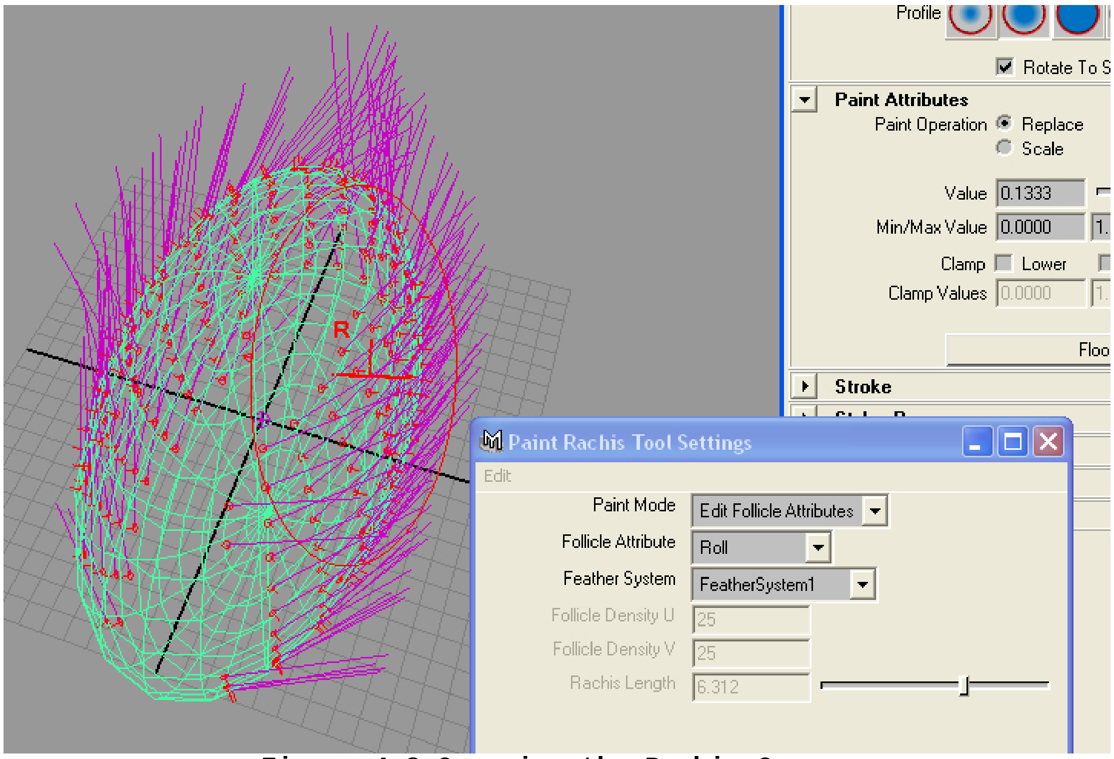
Radius(U) 1.0000
Radius(L) 0.0010
Opacity 1.0000
[✓ Rotate To Stroke]
Paint Operation [✓ Replace] Add Scale Smooth
Value 1.0000
Min/Max Value 0.0000 1.0000
Clamp ☐ Lower ☐ Upper
Clamp Values 0.0000 1.0000

Paint Mode: Create Follicles
Follicle Attribute: Roll
Feather System: FeatherSystem1
Follicle Density U: 25
Follicle Density V: 25
Rachis Length: 6.312

Figure 4.2 Creating Follicles for the Feather System

The user can add and remove any of the follicles at any time up until the generation of the barbs. The follicles are actually generated by passing parameters from the rachisPaintCurve script to the createFeatherCurveNode script, which is covered in the next section.

In order to groom the feathers the paint tool also has to allow for the user to change the attributes of the follicles and rachis. The user has the choice of changing the rotational position of the follicle (roll, inclination, polar), the scale of the rachis and the stiffness of the rachis, which are all controlled by the editFollicle and editFollicleValue procedures. The user just has to adjust the value of the paint attribute in order to affect the desired grooming method.


Profile

☑ Rotate To S

Paint Attributes

Paint Operation ● Replace ○ Scale

Value 0.1333

Min/Max Value 0.0000 1.

Clamp ☐ Lower ☐

Clamp Values 0.0000 1.

Floo

Stroke

Paint Rachis Tool Settings

Edit

Paint Mode: Edit Follicle Attributes

Follicle Attribute: Roll

Feather System: FeatherSystem1

Follicle Density U: 25

Follicle Density V: 25

Rachis Length: 6.312

Figure 4.3 Grooming the Rachis Curves

4.d. createFeatherCurveNode

The createFeatherCurveNode script is actually responsible for the generation of the feather follicles, the rachis', and the set up of the dynamics for each feather. The follicle details are passed to the createFeatherCurveNode procedure, which then attaches the follicle to the target surface based on these parameters, while also setting the dynamic attributes of the follicles. These can obviously be changed by hand by the user, but the set up has to be based on the most suitable attributes for realistic feathers (see Appendix A.3.).

The rachis description is then processed by the drawLineCmd procedure, which sets the control vertices of the rachis curve in local space, which is then converted into world space by the follicle, based on the input geometry (see Appendix A.4.).

The result is a large amount of cubic curves attached to the creatures surface, but in order to work further down the pipeline the curves had to be grouped into a logical order which could then be accessed by the barb curves to create a proper feather structure, which could be exported to a RIB file.

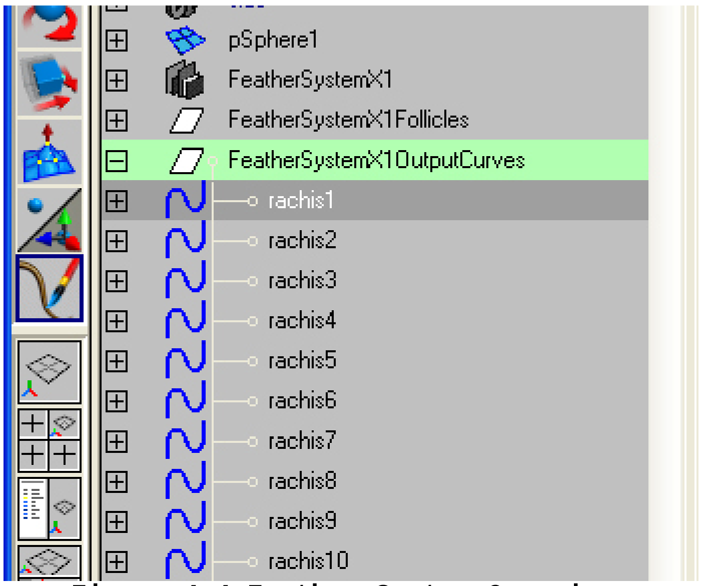

pSphere1
FeatherSystemX1
FeatherSystemX1Follicles
FeatherSystemX1OutputCurves
rachis1
rachis2
rachis3
rachis4
rachis5
rachis6
rachis7
rachis8
rachis9
rachis10

Figure 4.4 Feather System Groupings

5. Generating the Barbs

5.a. controlBarbs Node

The control barbs strategy documented in the Research section of this thesis (section 2.b.), was used in generating the barbs. This meant that there was actually a different node created for each of the eight barbs (four each side). It was important that each of the barbs started at exactly the same parametric position along the rachis curve as the Renderman DSO was dependent on these positions being 0.2, 0.3, 0.8 and 1.0. In order to achieve this, the pointOnCurve node was used to plug into the controlBarbs node and this gave the correct parametric position for the root of the barb on the rachis curve.

It was also necessary to scale the size of the barbs based on the size of the rachis, so the barbs appeared to be proportional to the rachis size. Another utility node called the curveInfo node was used to get the length of the curve even when it started bending.

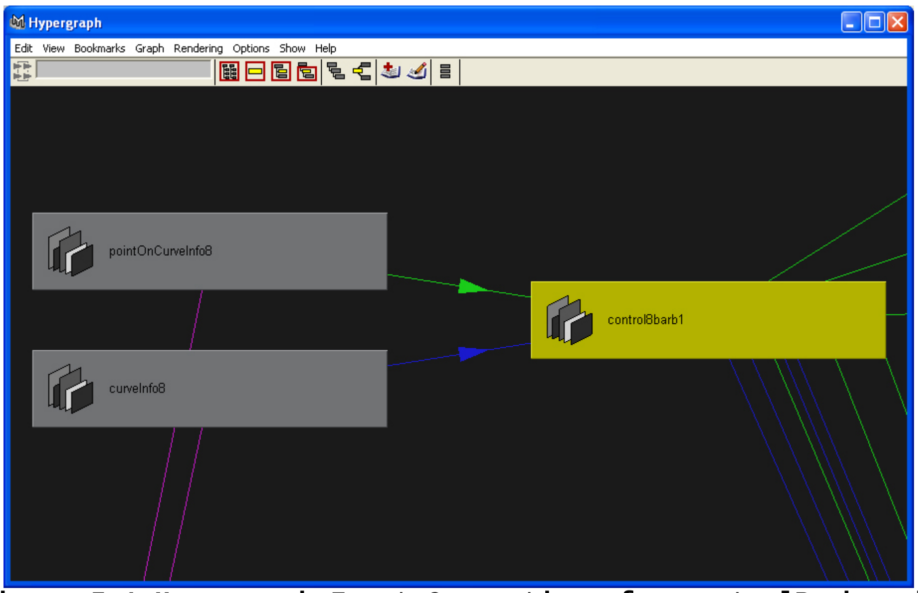


Figure 5.1 Hypergraph Input Connections for controlBarb node

The sole purpose of the controlBarb node is to set the initial positions of the four control vertices that make up the barb curve. An algorithm to achieve this was attempted but without success, so the positions are incremented by hard-coded amounts and scaled by the length in order to achieve a realistic barb shape.

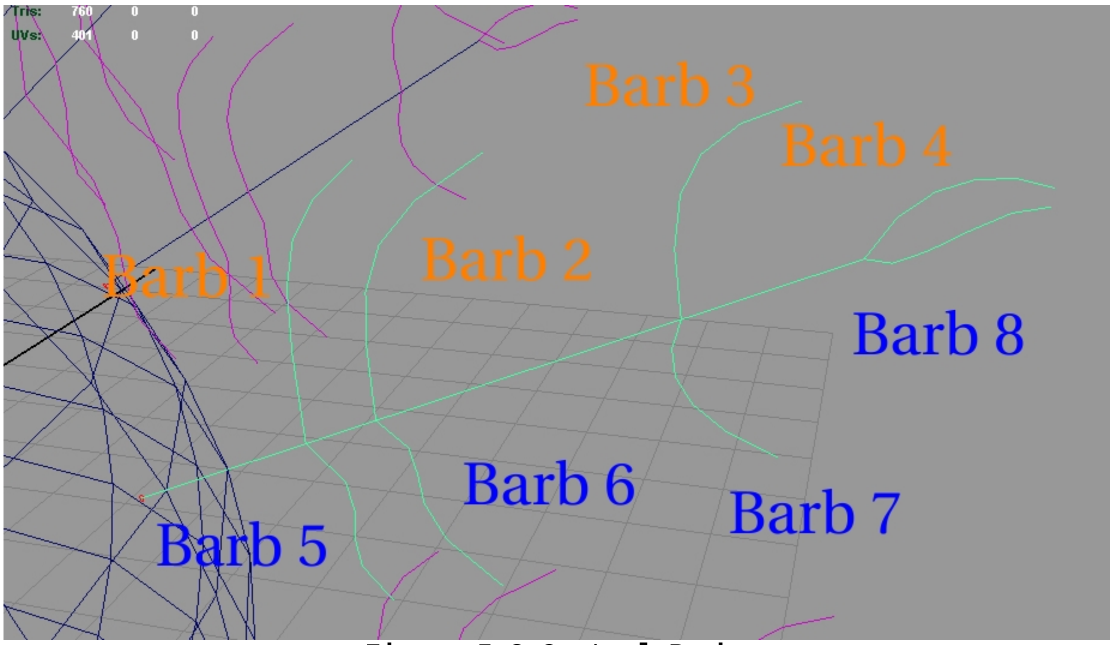


Figure 5.2 Control Barbs

The barb’s control vertices would then be set as attributes within the controlBarb node and passed onto the barbSolver node. The positions are then adjusted to take into account the surface geometry.

5.b. barbSolver Node

In order for the barbs to be positioned more realistically, they needed to be aware of the surface geometry. This had two effects. Firstly, it helped the feathers to take the shape of the surface when compacted against it. Secondly, it provides a more natural growing direction and adds randomisation to the feathers.

To achieve this, the information about the surface needed to be plugged into the barbSolver and the solution came from the closestPointOnMesh (for Poly) and closestPointOnSurface (for NURBS) nodes. With the use of these nodes the control vertex could be sent out and the closest point of the surface geometry was returned along with the surface normal. The closestPointOnSurface node at present does not have a surface normal attribute, so this would need to be rewritten if a NURBS geometry was the intended surface.

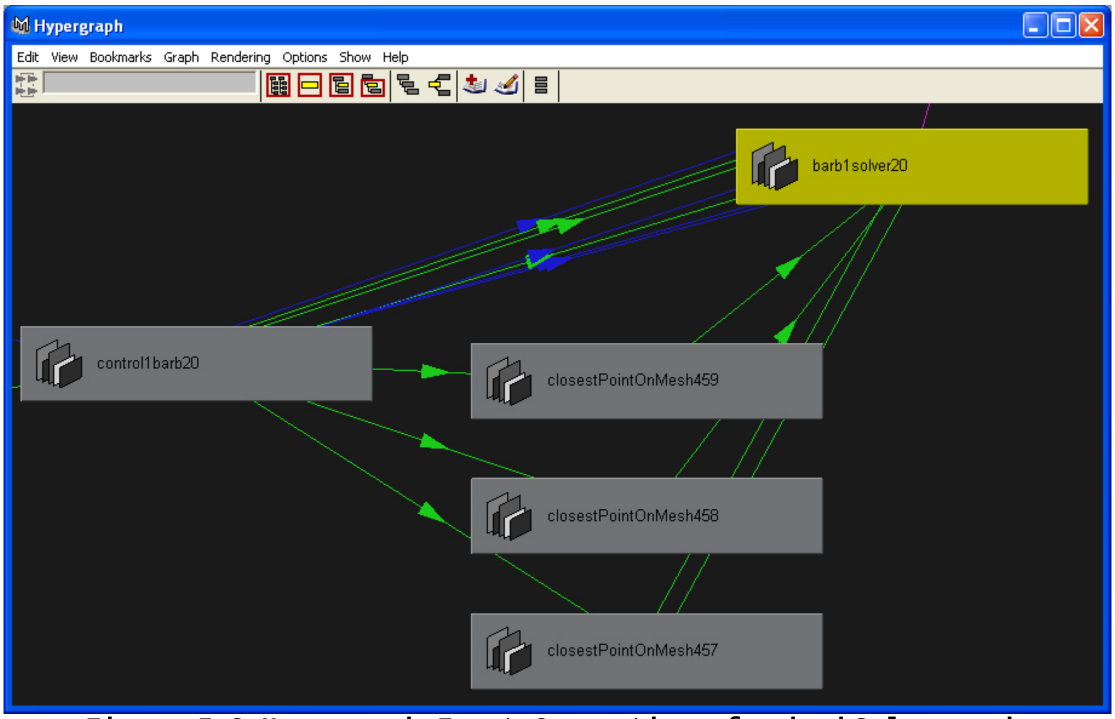


Figure 5.3 Hypergraph Input Connections for barbSolver node

Once the barbSolver has the closest points on the surface to each of the barb control vertices, then it can establish whether each control vertex is inside or outside of the surface geometry. If the control vertex is outside the surface geometry then the position is set based on the predefined hard-coded increments, which are adjusted based on the surface normal. If the barb control vertex is inside the surface geometry then this means there is the likelihood that the barb’s curve will pierce the skin, which would appear rather odd. To combat this, the barbSolver keeps adjusting the x component of the barb control vertex (assuming the creature is pointing along the x-axis), until it is no longer inside the surface geometry (see Appendix A.5.). Figure 5.4 shows how this works in practice. The feathers top row of barbs initially want to grow in a vertically upwards direction, but the shape of the geometry forces them to take on its surface shape.

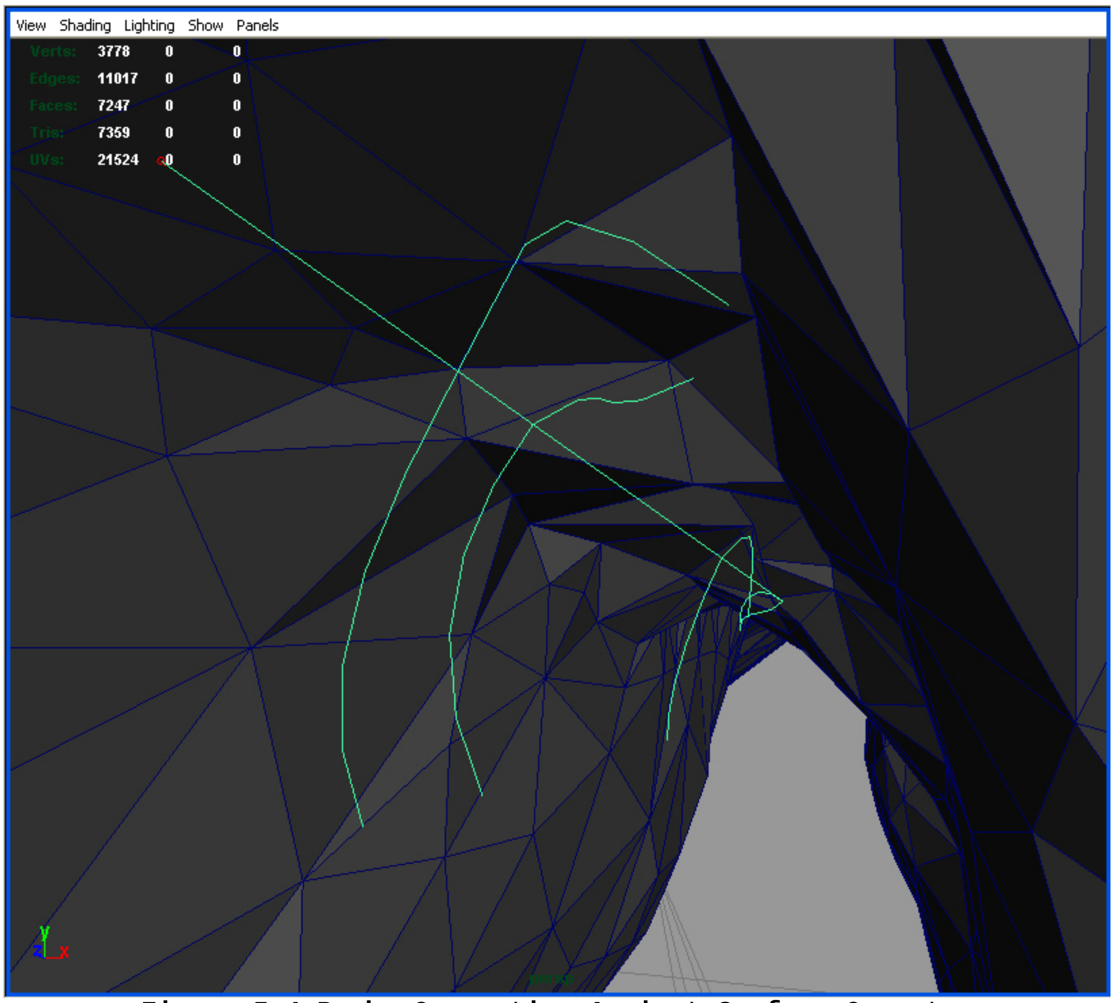


Figure 5.4 Barbs Compacting Against Surface Geometry

Now the barbSolver node has calculated the new positions of all the control vertices for a particular barb, then it needs to pass the curve data to the featherCollision node, in order to solve the interpenetration between feathers. In order to make this process as tidy as possible, rather than sending the featherCollision node hundreds of point attributes, the curve data is sent as MFnNurbsCurveData.

5.c. featherCollision Node

The first consideration the featherCollision node needed to make was what actually defined a feather. All the inputs to the node were curve attributes, but this in itself gave no clue as to what a feather was and what each feather was doing on the surface. In order to achieve the concept of feathers the featherStructure class was created. Each featherStructure instance would have an index and a location, which would be used to test for neighbouring feathers, as unlike the Chen et al (2002) paper, the feathers needed to be able to be created in any order the user wanted.

It was the responsibility of the Feathers class to construct the feathers systems and then solve the interpenetration between the feathers. Each vertices was passed from the input curve data attributes and converted into featherStructures, which could then be indexed and the location calculated from the average of all the points. The location only has to be an estimate, since it was only a way of rejecting feathers that were too far away to possibly interpenetrate the testing feather. It was important to conduct this test, as highlighted by Chen et al (2002), as the testing of every feather against every other feather was a very expensive process.

One of points made by Kaufman.D,(2002), in the paper about Sony Imageworks R&D on feathers was that the 1DVD and 2DVD processes came out of necessity from a restriction of movement from traditional collision detection between feathers. Although not actually specified, the restrictions appeared to be referring to how the feathers could become rigid and pushed from the body when solving the interpenetration by boundary-based collision detection. Although the 2DVD is a very interesting solution to the interpenetration problem it was never a practical method to implement in the time frame for this project, considering it took a team of highly experienced people eight months to fully develop, so a far simpler method was needed.

The method designed to solve the problem, was based on the idea of checking every control vertex (not the root control vertex of each barb, as this could not move without the rachis) of a feather against a control vertex on another feather. If the testing vertex was measured to be very close to another control vertex of a feather then it might be a contender of interpenetration and was moved in the direction back towards the root of the barb. It was a possibility that the control vertex was not actually causing an interpenetration, by just being close, but by moving it back towards the root of the barb made little difference. If this was the case then the adjustment would only be a small amount due to it quickly being outside the distance deemed close. A control vertex that was the wrong side on the other hand, would take more cycles of the loop to adjust as it was forced back towards the root of the barb and this movement would be more noticeable (see Appendix A.6.). On some occasions the control vertex may be impossible to adjust beyond the threshold distance because the two points' root positions might be very similar, but if the interpenetration occurred very close to the rachis then it would be far less likely to be visible to the human eye than an interpenetration occurring towards the tip.

One failure of this method is that it does rely on the groomer of the feathers to make sure no extreme interpenetrations occur like the one shown in Figure 5.5, otherwise the control vertices will be outside the distance threshold and then deemed to be without need of adjustment.

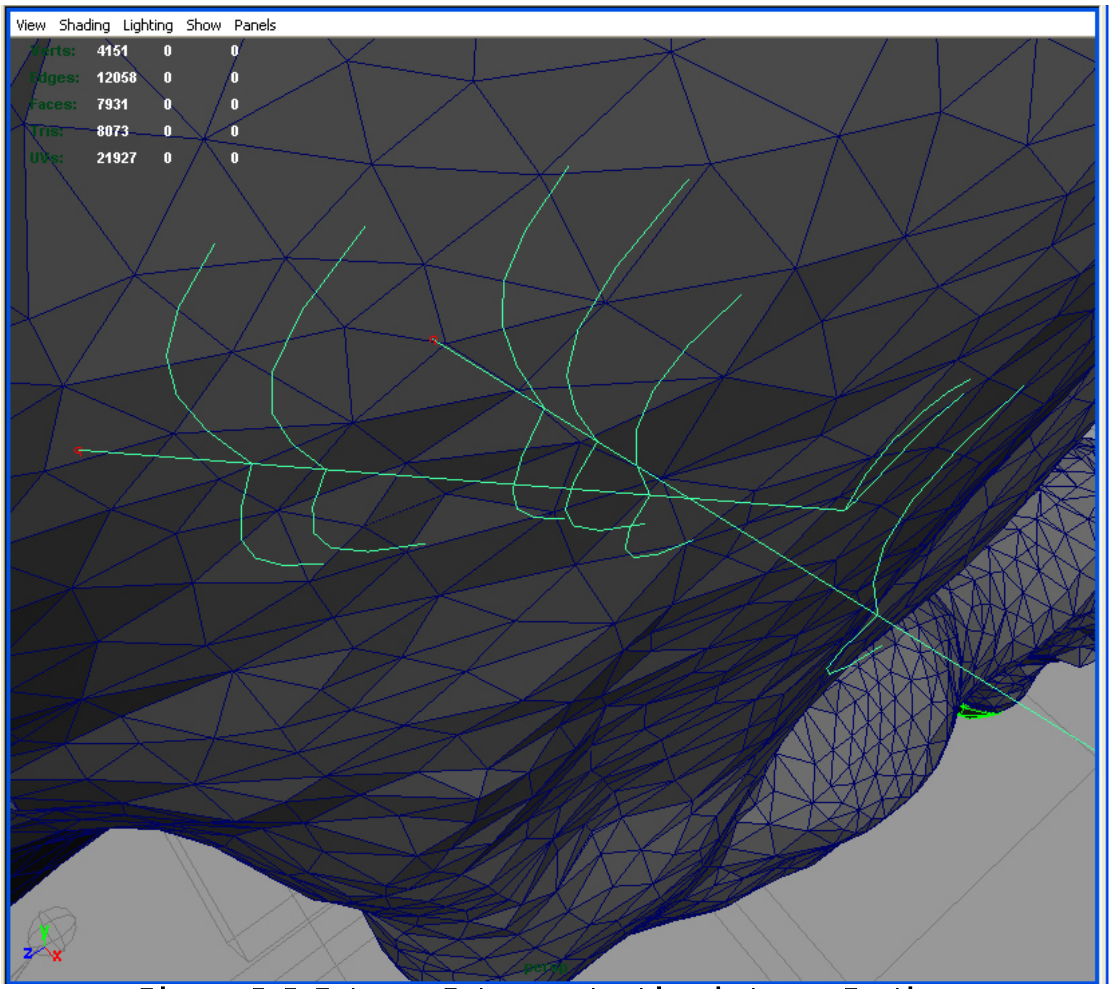


Figure 5.5 Extreme Interpenetration between Feathers

The decision to output curve data to nurbsCurve nodes was based on the fact the the user could then manipulate the points by hand if they so desired. Another option would have been to drawn them in openGL, which would have benefited the performance, but then the direct manipulation of the curves via Maya’s standard tools would not have been possible.

In order to output a varying amount of curves from the feathersCollision node the use of a MArrayDataBuilder was necessary. The MArrayDataBuilder element was added to the array every time a loop for one of the barb curves was executed. Each MArrayDataBuilder element contained another MFnNurbsCurveData which once plugged into a nurbsCurve node created the barb in Maya.

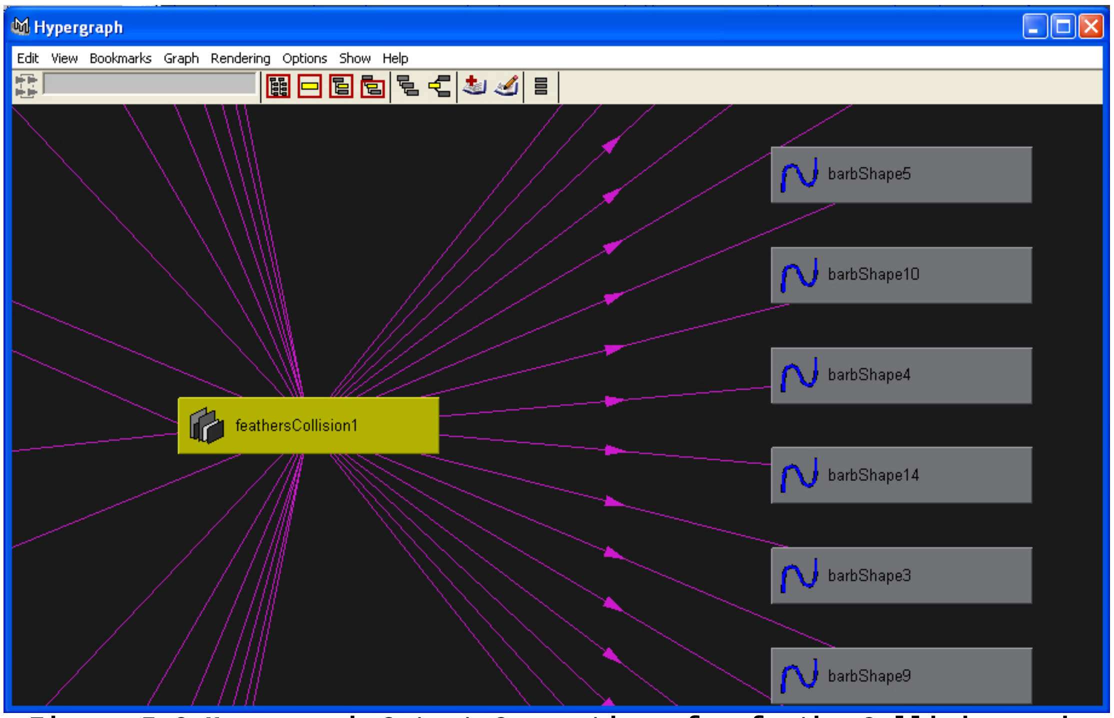


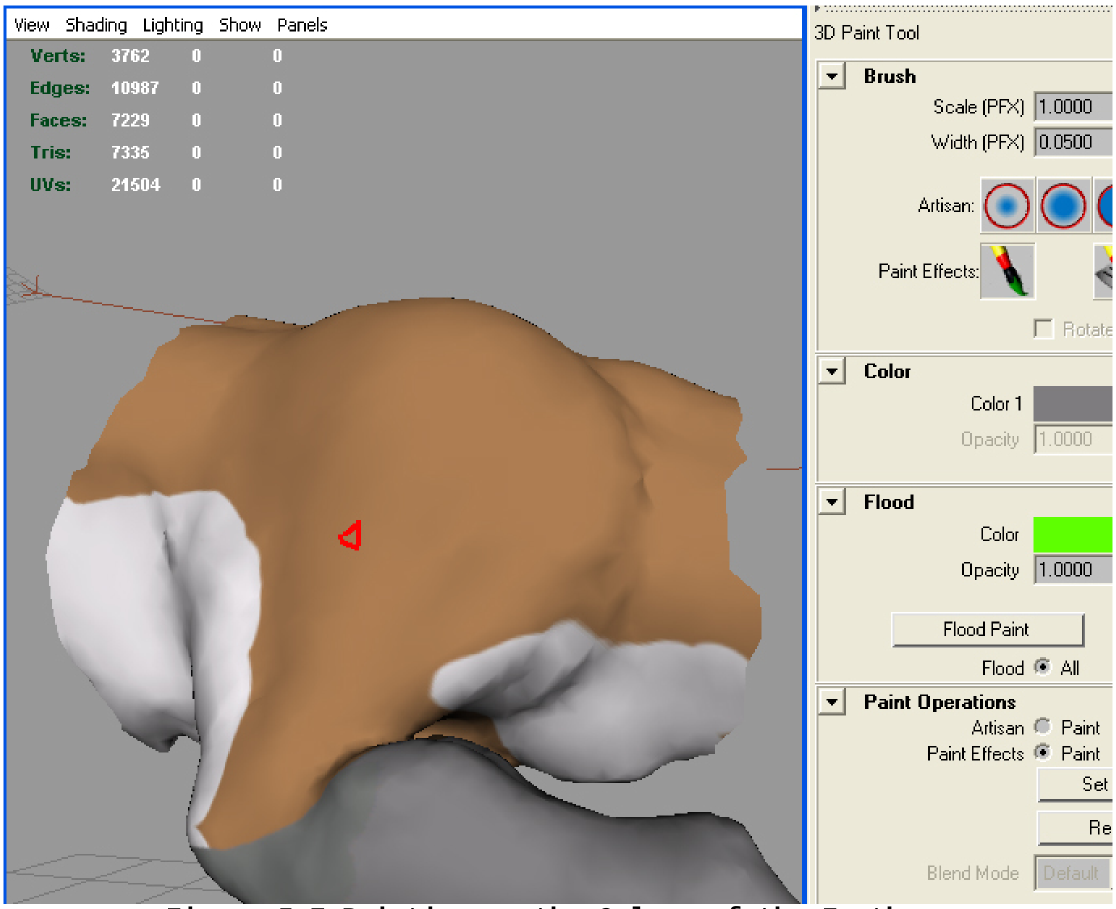
Figure 5.6 Hypergraph Output Connections for featherCollision node

5.d. addBarbs

The reasons for creating the barbs separately from the rachis have already been stated. It is the job of the addBarbs procedure to create all the nodes that go into making up the barb process and make all the necessary hypergraph connections. Once the procedure has worked out how many feathers it is dealing with and what the target surface geometry is, it then loops through each feather creating the nodes for the barb generation.

5.e. assignFeatherColours

To assign colours to the feathers a common technique of painting directly onto the geometry was used. The artist could use Maya’s 3D Paint Tool or any other 3D paint package to paint the desired colour for the feathers in certain regions on the geometry. By using this technique, it gave the artist quick, predictable results, which they would feel very comfortable with. Figure 5.7 shows an example of this painting process.



Verts: 3762 0 0
Edges: 10987 0 0
Faces: 7229 0 0
Tris: 7335 0 0
UVs: 21504 0 0

Scale (PFX) 1.0000
Width (PFX) 0.0500
Color 1
Opacity 1.0000
Color
Opacity 1.0000
Flood ● All
Artisan ○ Paint
Paint Effects ● Paint
Blend Mode Default

Figure 5.7 Painting on the Colour of the Feathers

At present to assignFeatherColours procedure needs the texture image to be a predefined name and resolution, but this can be easily updated to allow the user to input the texture file’s details. The assignFeatherColours procedure loops through each of the feather follicles and based on the u and v parameters of the follicle it works out which pixel in the texture file corresponds to the position of the follicle on the geometry. It then uses the iffPixel procedure to return the colour of that particular pixel and sets the follicle colour to this.

6. Rendering the Feathers

6.a. RIB Export

The decision to use PRman was explained in section 3.b, but the feather curves could not just be exported normally using Pixar’s Maya to Renderman (MTOR) tool, since it would create an incompatible RIB format with what was needed for the input to the procedural primitive DSO.

To solve this problem another MEL procedure was created called writeFeathersRIB, which cycled through each frame of the animation and exported the RIB file in the desired format. The DSO needed the curves in a set order, so it could interpolate the missing barbs correctly, and to make the render more efficient the DSO uses the bounding box information to delay the reading of the RIB data until absolutely necessary, instead of reading all the RIB data into memory at the start.

Another requirement that be exported, was the colour and surface normal of the feather follicle. These were necessary in order to shade the feathers properly, and will be discussed in more detail in section 6.c..

At present the writeFeathersRIB script is only set up for the testing scene with a predefined camera and light position. It would not take much to introduce the calculations for the camera and light positions into the procedure and once this has been done, it should be able to cope with any Maya scene information.

6.b. Feather Detail Procedural Primitive DSO

The Feather Detail DSO was used to generate the majority of the feather structure at render-time. As discussed earlier, this provided an increased efficiency for work flow performance, since a large proportion of the geometry did not exist until the very last stage of the pipeline. It also meant that it reduced the amount of RAM required, a problem highlighted in the Research section. This was achieved by delaying the reading of the RIB data until the bounding box was reached.

The DSO is sent its input in the form of a string via the RIB. The DSO takes the data string from the RIB and converts it to dynamically allocated blind data. In this particular case the string was a list of the control points that made up each control barb. The DSO was reliant on the order of the points provided by the RIB, since that data allocation process was always the same.

The DSO starts at the parametric position 0.2 (position of the first barbs) along the rachis and then makes small increments up of the rachis applying a barb at each position based on an interpolation of the control barbs either side of it. A straight interpolation between each of the four points that made up the barb would not have produced a satisfactory result, since if the rachis was bent in any direction, the barbs would no longer be attached to the rachis. Since all the barbs and the rachis where Bezier curves of degree 3, De Casteljau's Algorithm (see Appendix A.7.) could be used to find out the point on the rachis curve and then use this point as the root position for the barb. The result of this meant that however much the rachis bent, the barb would always stay attached to the rachis.

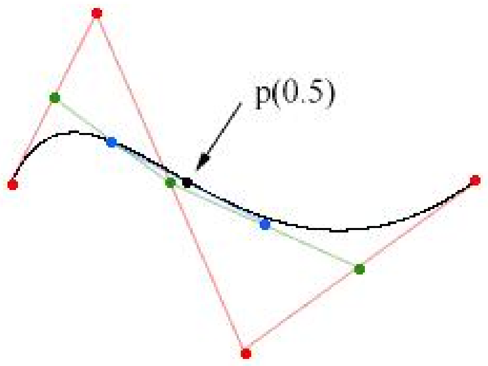


Figure 6.1 De Casteljau's Algorithm

Once the control vertices for the barb have been determined they are sent to the drawCurve function, which then outputs the curve as a Renderman RiCurve primitive that gets thinner towards the tip. The benefits of the RiCurves primitive have been discussed in section 2.f, but the nature of the Curves primitive is that it always faces the camera, which causes a problem when being shaded. The solution to this problem is discussed in the next section.

6.c. Renderman Shader

Since the surface properties of hair and feathers are similar, a search for an existing hair shader was conducted as there seemed little chance of a feather shader based on RiCurves already existing. The RiCurves was an important factor because this primitive consists of many small micro-polygons falling along a specified curve with their normals pointing directly towards the camera (like a 2-dimensional ribbon always oriented towards the camera), making it shade differently to other primitives. One such shader was found in a presentation given by Bredow, R. (Sony Picture Imageworks, 2000) on the Fur in Stuart Little. The shader provided in the notes was actually developed by Hanson, C. and Bruderlin, A.

Some of the parts of the shader were not applicable to feathers, such as clumping, and therefore were removed to increase performance, but the most useful part was how they obtained a shading normal at the current point on the hair, which was not just forward facing. This was achieved by mixing the surface normal vector at the base of the hair with the vector at the current point on the hair. The amount with which each of these vectors contributes to the mix is based on the angle between the tangent vector at the current point on the hair and the surface normal vector at the base of the hair. The smaller this angle, the more the surface normal contributes to the shading normal.

The test renders conducted for this project were intended to be similar to that of a bird of prey, which meant that the shininess of hair would not be appropriate. To solve this problem the specular levels were reduced, by reducing the global variable Ks, but still allowing the specular only to hit in certain parts of the curves, based on the v global variable. After testing and based on pictures of feathers from the research it was decided that the base colour of the feather passed via the RIB should be more prominent towards the root of the barbs, so the root colour was set to pure white, allowing the base colour to totally dominant once added to the calculations. The tip colour was, in contrast to human hair, was set to a dark grey, so the base colour would appear to fade out towards the edges.

The test animation contained 750 feathers of varying sizes and each 1K frame averaged about 40 seconds on a single Intel Pentium 4 2.2GHz processor with 768MB of RAM. This speed can be considered extremely good, as once sent to a multi cpu renderfarm an entire shot could be rendered in a matter a minutes, thus allowing small alterations and continuous updated versions, essential control for a visual effect.

7. Conclusion and Future Work

The success of the Start to Finish Feathers Solution has to be measured by several factors, but most important, is the quality of the final images it produces. These have been very successful and have fulfilled the ambitions of the original aim of this project. The testing geometry was chosen because it provided a fully rigged, deforming skin that had a large array of movement and produced creases around the joints. The SFFS managed to work successfully in these demanding conditions and was capable of providing a photorealistic animation as a result (see Feathers_Solution_Demo.m2v provided on CD and Appendix B).

Another important consideration as stated in section 3, was the efficiency of the solution in a high pressure working environment of a visual effects company. Having the potential of applying the feathers to both NURBS and polygon meshes is a key issue, since it provides no restrictions to the modellers. Having a restriction of geometry type can affect the creative process, while also possibly rendering other propriety animation tools useless. Once the pipeline has been set in place, it does not require a Technical Director to be involved in the process, which means it leaves him/her free to do other jobs. A specialist feather groomer could look after the feathering process from start to finish, hence the name. The majority of the work involves the user painting things onto the surface of the geometry and playing around with a few of the dynamics settings to get optimum realism. This is not a demanding task and could be best achieved by most intermediate users of Maya with very little or no knowledge of Renderman. Once the barbs have been created and the feathers assigned, the job of the groomer is done, until they get the results of the render back. Obviously, with the feather detail generated at render time, the groomer will never know exactly what the result is going to be, but with experience would come a good predictions of results.

The performance of the process can be considered good as the testing was done successfully on below industry standard machines. There was no bottle neck in the pipeline where the user had to sit and wait for a considerable amount of time up until the rendering process.   One area of concern was the use of Maya’s closestPointOnMesh node, which appeared to use large amounts of memory, when the mesh was highly detailed (i.e. large poly count), but this did not actually slow the process down, though it could restrict the number of feathers applied in extreme circumstances.

The testing of the pipeline has shown that the concepts used can provide a successful solution to the problem of feathers, but it is by means it a perfect solution without the need of development.

An area of improvement in barb generation process would be to allow the user paint barb attributes, as at present there is no way of editing the barbs. This would be useful because it would allow the user to specify the look of the feathers rather than the programmer.

One area not explored due to the close-up nature of the testing was a level of detail method built into the procedural primitive DSO. This could work by using less curves to generate the feather detail based on the distance from the camera, thereby reducing render times.

An improvement to the shader would be the ability to create the stylish patterns that some feathers contain, as established in the research. This could be generated procedurally or texture-based, but it would provide a more realistic look for the feathers, rather than just a blend of one colour into another.

For areas of the geometry where there would be high levels of rotational deformation, such as a bird’s turning neck, the two-dimensional volume deformation technique used by Sony Imageworks, mentioned in section 2.d, would be a very useful development to the current solution. It would not be a simple task, but the framework for the current solution would allow for such computations without the need to re-write parts of the existing pipeline.

Further testing on a variety of different creatures and body shapes is still necessary in order to better gauge the overall usefulness of the SFFS, but the results achieved so far meet the requirements of the original aim well and provide a platform for the solution to be further developed in several different ways.

References

Armstrong, E et al. **Stuart Little 2:Let the Feathers Fly**. *Presentation Course Notes from SIGGRAPH '02*. 2002

Apodaca, A. and Gritz, L. **Advanced Renderman**. Morgan Kaufmann, 2000.

Attenborough, D. Life on Earth. Published by Collins, 1980.

Bredow, R. **Fur in Stuart Little**. *Presentation Course Notes*. Sony Pictures Imageworks. http://www.185vfx.com/resources/coursenotes.html, visited 07/05.

Carner, C and Hong, Q. **Elastic Paint: A Particle System for Feature Mapping with Minimum Distortion**. Research Paper from *State University of New York at Stony Brook*, 2003.

Chen, Y et al. **Modelling and Rendering of Realistic Feathers**. In *Computer Graphics (SIGGRAPH '02 Proceedings)*, volume 21, pages 630-636, 2002.

Framestore CFC Harry Potter 3. 3DA Interview with David Lomas, CG Supervisor. http://www.3dyanimacion.com/entrevistas/entrevistas.cfm?link=framecfchp3eng01, visited 07/05.

Gould, D. **Complete Maya Programming**. Morgan Kaufmann Publishers, 2003.

Harry Potter and the Prisoner of Azkaban. Warner Bros Entertainment, 2004.

Harvey, A. **Grooming Furry Surfaces of Arbitary Topology**. In *Computer Graphics (SIGGRAPH '03 Proceedings)*, pages 1-1, 2003.

Kajiya, J. and Kay, T. **Rendering fur with three dimensional textures**. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 271-280, 1989.

Lokovic, T. and Veach, E. **Deep Shadow Maps.** In *Computer Graphics (SIGGRAPH '00 Proceedings)*, pages 385-392, 2000.

Maya API Classes Documentation, Alias Systems Corp, Toronto, Canada.
Parent, R. **Computer Animation Algorithms and Techniques**. Morgan Kaufmann, 2002.

Renderman Documentation (RAT 5.5, PRMAN 11.0), Pixar Animation Studios, Ca, USA.

Streit, L and Heidrich, W. **A Biologically-Parameterized Feather Model.** In EuroGraphics ’02 Proceedings, Volume 21, Number 3, 2002.

Streit, L and Heidrich, W. **Generating Feather Coats Using Bezier Curves.** Research Paper from *Dept. of Computer Science University of British Columbia*, 2002.

Walter, M. and Franco, C. **Modelling the Structure of Feathers.** In *Brazilian Symposium on Computer Graphics and Image Processing Proceedings*. 2001

Zhang, J. **ZJProgramming.com.** http://www.zjprogramming.com/, visited 07/05.

Appendix A: Code Extracts

A.1. Code Extract from paintRachisTool.mel

```
string    $cmd;

$cmd  = "artUserPaintCtx -e";
// Tool Setup Cmd
$cmd += " -tsc \"rachisCurvePaint\"";
// Tool Cleanup Cmd
$cmd += " -tcc \"cleanupRachisPaint\"";
// Initialise Cmd
$cmd += " -ic \"initRachisPaint\"";
// Finalise Cmd
$cmd += " -fc \"finishRachisPaint\"";
// Set Value Cmd
$cmd += " -svc \"setRachisPaintValue\"";
// Get Value Cmd
$cmd += " -gvc \"getRachisPaintValue\"";
// Get Array Attribute Cmd
$cmd += " -gac \"\" -cc \"\" -gsc \"\"";
$cmd +=           $gPaintHairAttrToolCtx;
eval $cmd;
```

A.2. Code Extract from rachisCurvePaint.mel

```
if($gRachisCurveFeatherSys == "")
{
        $gRachisCurveStartIndex[0] = 0;
        string $m =`createNode -n "FeatherSystemX#" transform`;
        $gRachisCurveFeatherSys = `createNode -p $m -n "FeatherSystem#" hairSystem`;
        int $numHsys = size( $gRachisCurveFeatherSystems );
        $gRachisCurveFeatherSystems[$numHsys] = $gRachisCurveFeatherSys;
        connectAttr time1.outTime ($gRachisCurveFeatherSys + ".currentTime");
}
```

A.3. Code Extract from createFeatherCurveNode.mel

```
// Set follicle positional and dynamic attributes
string $foll = `createNode follicle`;

// Set parameteric positions
setAttr ($foll + ".parameterU" ) $u;
setAttr ($foll + ".parameterV" ) $v;
// Set simulation method so feathers are dynamic
setAttr ($foll + ".simulationMethod") 2;
// Point locked at both ends
setAttr ($foll + ".pointLock") 3;
// Get Transform
string $tforms[] = `listTransforms $foll`;
string $follDag = $tforms[0];
int $attachedToSurface = false;
```

```
if( $surface != "" && objExists( $surface ) )
 {
        // Determine type of the target surface and apply world
        // matrix and geometry data types to follicle attributes
        string $nType = `nodeType $surface`;
        connectAttr ($surface + ".worldMatrix[0]") ($foll + ".inputWorldMatrix");
        if( "nurbsSurface" == $nType )
        {
            connectAttr ($surface + ".local") ($foll + ".inputSurface");
        }
        else if( "mesh" == $nType )
        {
            connectAttr ($surface + ".outMesh") ($foll + ".inputMesh");
            // For poly need currentUV set
            string $currentUVSet[] = `polyUVSet -q -currentUVSet $surface`;
            setAttr ($foll + ".mapSetName") -type "string" $currentUVSet[0];
            int $isValidUv = getAttr( $foll + ".validUv" );
            if( !$isValidUv )
           {
                delete $follDag;
                return "";
            }
        }

            connectAttr ($foll + ".outTranslate") ($follDag + ".translate");
            connectAttr ($foll + ".outRotate") ($follDag + ".rotate");
            // Stop accidental editing via channels menu
            setAttr -lock true  ($follDag + ".translate");
            setAttr -lock true  ($follDag + ".rotate");
}
```

A.4. Code Extract from createFeatherCurveNode.mel

```
// Cubic curve
string $cmd = "curve -d 3";
int $i;
float $fac = $curveLength/(float)($numCvs-1);
for( $i = 0; $i < $numCvs; $i++ )
{
     $cmd += (" -p 0 0 " + ((float)$i * $fac));
}
return $cmd;

…

connectAttr ($surface + ".worldMatrix[0]") ($foll + ".inputWorldMatrix");
```

A.5. Code Extract from barb1solver.cpp

```
// Base growth direction on surface normal
if(mnormal[0]>=0.0)
{
            // If second CV is outside closest point on skin
            if(mposition2x>=mclosest2[0])
            {
                // Set the second CV position
                MPoint cv1(mrootPosition1-(1-mnormal[0]),
mrootPosition2+(0.25*mscale+mnormal[1]), mrootPosition3+(0.15*mscale));
                // Apply second CV to vertices array
                vertices.append( cv1 );
        }
        else
        {
                // Make sure the second CV is not inside closest point on skin
                while(mposition2x<mclosest2[0])
                {
                        mposition2x = mposition2x+0.05f;
                }
                // Set the second CV position
                MPoint cv1( mposition2x, mposition2y, mposition2z);
                // Apply second CV to vertices array
                vertices.append( cv1 );
        }
…
```

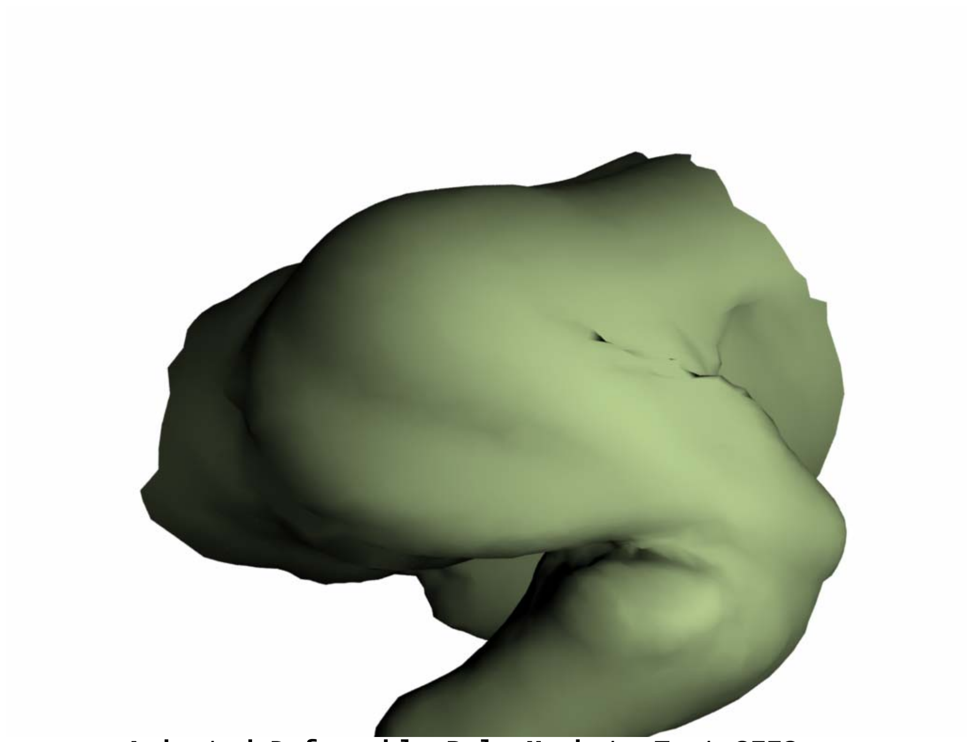
A.6. Code Extract from feathers.cpp

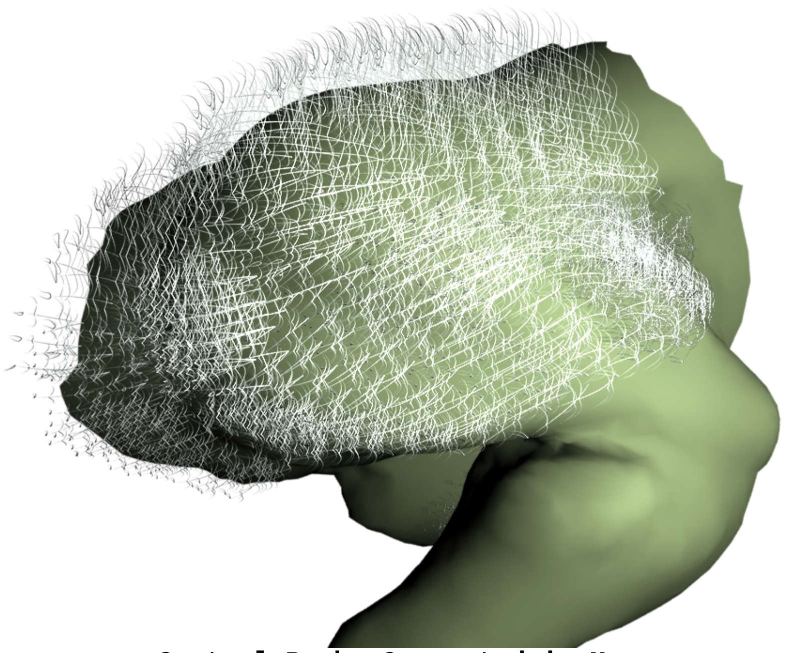
```
// Check distance between cvs
float dist = distanceBetween(feathers[r].barbs[d]->cv[e]->position,
point1);
unsigned int breakloop = 0;
while(dist<0.2 && breakloop < 6)
{
      // Move the cv back towards the root of the curve based on the
vector direction
      Point3D vec = vectorToRoot(feathers[r].barbs[d]->cv[0]-
>position, feathers[r].barbs[d]->cv[e]->position);
      Point3D steps = vec/5;

      feathers[r].barbs[d]->cv[e]->position+=steps;
      // Calculate bew distance between cvs
      dist = distanceBetween(feathers[q].barbs[b]->cv[e]->position,
feathers[r].barbs[d]->cv[e]->position);
      breakloop++;
}
```

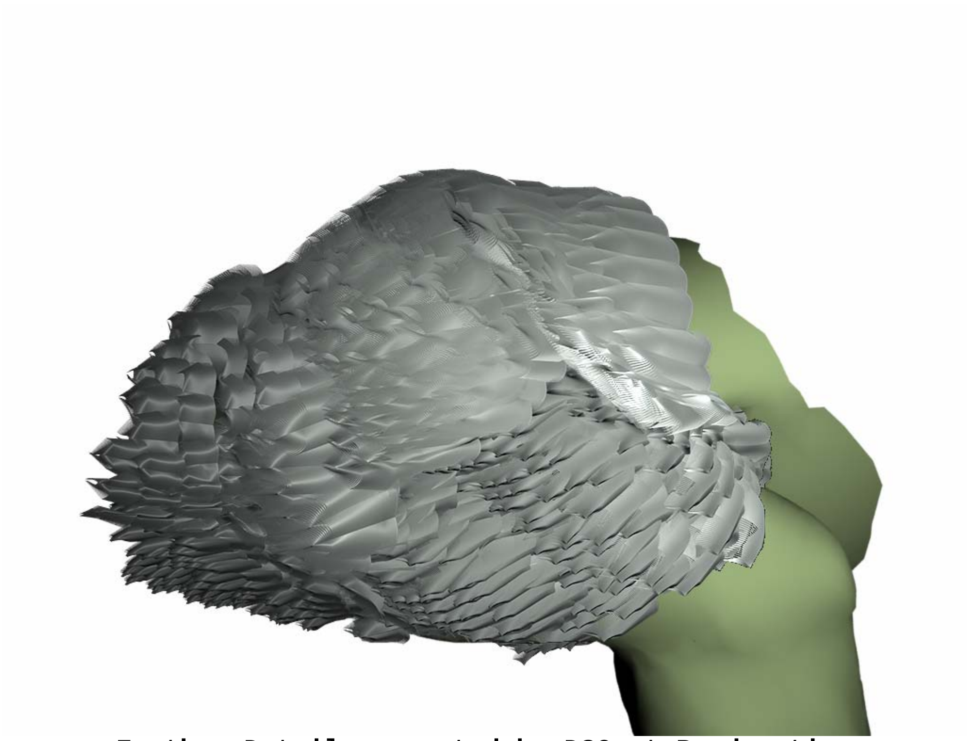
A.7. Code Extract from feather_detail.c

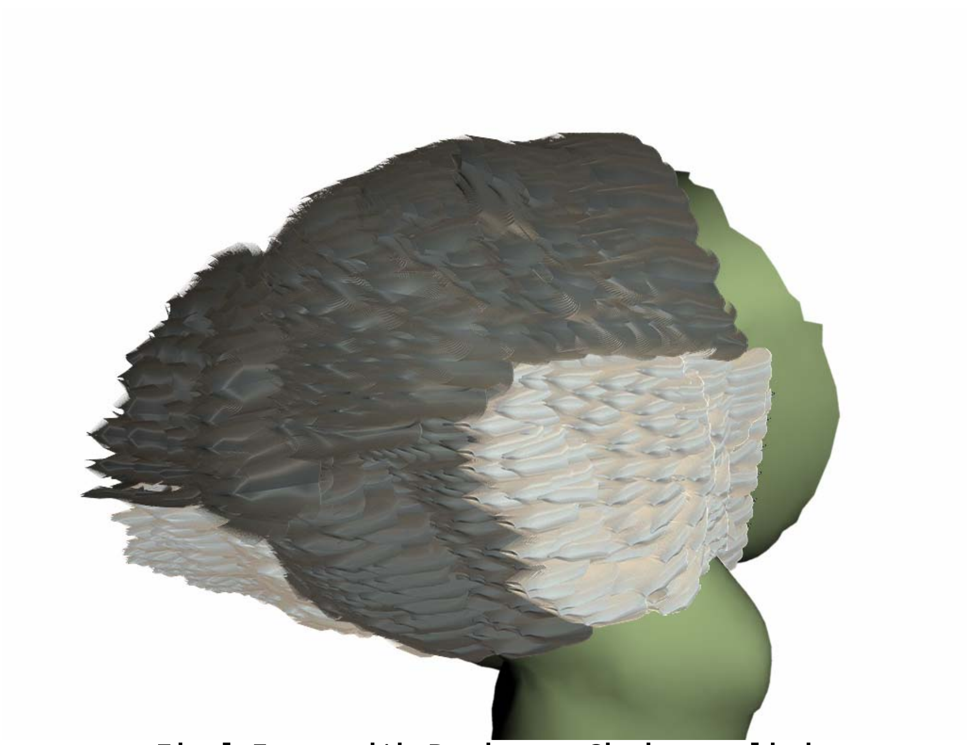
```
// De Casteljau Algorithm
float PointOnBezier(float t, float cv1, float cv2, float cv3, float
cv4)
{
      float ab, bc, cd, abbc, bccd, result;
      ab = lerp(cv1, cv2, t);
      bc = lerp(cv2, cv3, t);
      cd = lerp(cv3, cv4, t);
      abbc = lerp(ab, bc, t);
      bccd = lerp(bc, cd, t);
      result = lerp(abbc, bccd, t);
      return result;
}
```

Appendix B: Test Images


Animated Deformable Poly Mesh to Test SFFS on


Control Barbs Generated in Maya


Feather Detail generated by DSO at Render-time


Final Image with Renderman Shader applied