

Real-Time Rendering of Fur



Gary Sheppard

Supervised by
Steve Maddock

5th May 2004

Module: COM3010 / COM3021

This report is submitted in partial fulfilment of the requirement for the degree of Bachelor of Engineering with Honours in Software Engineering by Gary Sheppard.

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations which are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: Gary Sheppard

Signature:

Date: 05/05/04

Abstract

A vast number of creatures in the animal kingdom exhibit some kind of fur. This paper presents a complete 'shells and fins' method for modelling this fur in real-time for use in interactive systems. A wide range of properties are researched, evaluated and adapted for use in the project.

Original methods are presented for hair type interpolation and fur dynamics. Here, the concept of hair domination is introduced as a method to decrease hair quantities in a fur patch, and the wind vector animation approach is adapted for use on fur.

A review is then made against other methods of fur rendering to develop concept for further work.

Acknowledgements

I would first of all like to thank Steve Maddock, my supervisor, for his guidance throughout the project. Also, for his patience and understanding at the beginning of the year when things all got a little confusing.

I would also like to thank:

Adam Miller for all the things he's done to help out with this project, which are too numerous to mention. Many thanks!

Neil Daniels for his input during weekly meetings.

My parents for their continual support and willing to proof read at all times of day and night.

The people of Gamedev.net and the #OpenGL irc channel for providing such bountiful information regarding OpenGL and shader languages

And finally, many thanks to Kaldi, an Ethiopian goat herder, who discovered Coffee in 850AD. This project would never have been finished without his pioneering work.

Contents

Chapter 1: Introduction.....	1
Chapter 2: Literature Review.....	2
2.1 Approaches to Modelling Fur.....	2
2.1.1 <i>Polygon Strip Approach</i>	2
2.1.2 <i>Texels Approach</i>	3
2.1.3 <i>Shells Approach</i>	3
2.1.4 <i>Summary</i>	4
2.2 Shell and Fin Variants.....	4
2.2.1 <i>Length</i>	4
2.2.2 <i>Colour</i>	5
2.2.3 <i>Lighting</i>	6
2.2.4 <i>Self-Shadowing</i>	7
2.2.5 <i>Summary</i>	9
2.3 Texture Generation.....	9
2.3.1 <i>Particle Systems Method</i>	9
2.3.2 <i>Noise Method</i>	10
2.3.3 <i>Summary</i>	10
2.4 Texture Application.....	10
2.4.1 <i>Inverse Mapping using an Intermediate Surface</i>	10
2.4.2 <i>Lapped Textures</i>	11
2.4.3 <i>ABF + Overlay Grid Smoothing</i>	12
2.4.4 <i>Summary</i>	13
2.5 Dynamics.....	14
2.5.1 <i>Wind Vectors</i>	14
2.5.2 <i>Loosely Connected Particles</i>	15
2.5.3 <i>Summary</i>	15
Chapter 3. Theory Analysis.....	16
3.1 Aims of the Project.....	16
3.2 Key Methods.....	17
3.2.1 <i>Representation</i>	17
3.2.2 <i>Whiskers and Tufts</i>	18
3.2.3 <i>Colour</i>	19
3.2.4 <i>Lighting</i>	19
3.3 The Hair Types Problem.....	20
3.3.1 <i>Hair Dominance</i>	20
3.3.2 <i>Interpolation</i>	21
3.3.3 <i>Texture Generation</i>	22
3.3.4 <i>Texture Application</i>	23
3.4 The Dynamics Problem.....	23
3.4.1 <i>Defining the problem</i>	23
3.4.2 <i>Deriving the wind vectors</i>	24
3.4.3 <i>Perturbing the geometry</i>	25
3.5 Testing Plan.....	26
3.6 Evaluation Plan.....	27
3.7 Tools.....	27
3.7.1 <i>Programming Language and API's</i>	27
3.7.2 <i>Shader Language</i>	28

Chapter 4. FurMak	29
4.1 FurMak.....	29
4.2 Texture Generation.....	30
4.2.1 Data Structures.....	30
4.2.2 Generating Hair Data.....	31
4.2.3 Converting Data into Texture Maps.....	32
4.2.4 Testing.....	33
4.3 Model Conversion & Texture Co-ordinates.....	34
4.3.1 Data Structures.....	34
4.3.2 Vertex Mapping System Structure.....	35
4.3.3 Ensuring correlation.....	36
4.3.4 Testing.....	37
Chapter 5. FurGLe	38
5.1 System Design.....	38
5.2 Fur Rendering.....	39
5.2.1 Updating the Forces.....	39
5.2.2 Constructing the Display Lists.....	40
5.2.3 Rendering the Scene.....	41
5.3 Shaders.....	43
5.3.1 Shell Vertex Shader.....	43
5.3.2 Shell Fragment Shader.....	45
5.3.3 Other Shaders.....	46
5.3.4 Testing.....	47
5.4 User Interfaces.....	47
5.4.1 View Mode.....	48
5.4.2 Edit Mode.....	48
5.4.3 Whiskers Mode.....	49
5.4.4 Testing.....	49
Chapter 6. Results and Evaluation	50
6.1 Fur Rendering.....	50
6.1.1 Fur Features.....	50
6.1.2 Interpolation.....	51
6.1.3 Dynamics.....	52
6.2 Fur Generator.....	53
6.2.1 Results.....	53
6.2.2 Evaluation.....	55
6.3 Overall System.....	56
6.4 Usability.....	57
6.4.1 Comparison to Existing Methods.....	57
6.4.2 Future Potential.....	58
6.5 Future Work.....	59
Chapter 7. Conclusions	60
References	61
Appendix – Shader Code	63

List of Figures

2.1 Polygon Strips.....	2
2.2 Representation of Shells.....	3
2.3 Representation of Fins.....	4
2.4 The need for Offset Maps.....	5
2.5 Visualisation of Diffuse Lighting Vectors.....	7
2.6 Visualisation of Specular Lighting Vectors.....	7
2.7 Example of a Shell Normal Map.....	7
2.8 Shadow Maps.....	8
2.9 Generating Textures using a Particle System.....	9
2.10 Generating Textures using Noise.....	10
2.11 Inverse Mapping using an Intermediate Surface.....	11
2.12 O-Mapping methods.....	11
2.13 Lapped Textures.....	12
2.14 ABF + Overlay Grid example.....	13
2.15 Fur Anomalies from Discontinuous Texture Mapping.....	13
2.16 Wind Vector Calculations.....	14
3.1 Comparison of Results over Varying Shell Quantities.....	17
3.2 Limitation of Fins Viewing Angle.....	18
3.3 Demonstration of silhouette only fins.....	18
3.4 Obtaining the Hair Vector for Lighting Fins.....	20
3.5 Properties of Realistic Hair Interpolation.....	20
3.6 Example of a Dominance Map.....	21
3.7 Interpolation Existence/Dominance Graph.....	22
3.8 Hair Width Reduction Graph.....	22
3.9 Approximating the Bend Factor.....	25
3.10 Comparison of Original and Approximated Hair Bending.....	26
3.11 Illustration of Sub-Fins.....	26
4.1 FurMak Program Flow.....	29
4.2 Fur Texture Construction.....	30
4.3 Data Structure used for Generating Fur Textures.....	30
4.4 The shellify() Method.....	32
4.5 Correct Collision Detection.....	32
4.6 Fin Texture Creation Code.....	33
4.7 The 4 Stages of Model Conversion.....	34
4.8 FurMak Model Data Structure.....	34
4.9 Code for Generating Texture Maps.....	35
4.10 Texture Co-ordinate Correction Code.....	36
4.11 Cube-face Anomalies in Texture Map.....	37
4.12 Corrected Texture Allocation.....	37
5.1 The Structure of FurGLE.....	38
5.2 The Main Program Loop.....	39
5.3 Main Program Loop Code.....	39
5.4 Wind Vector Calculation Code.....	40
5.5 Fin Selection Method.....	41
5.6 Problems Mapping the Fins.....	41
5.7 The Scene Rendering Routine.....	42
5.8 Positional Vertex Shader Code.....	44
5.9 Calculating the Resultant Lighting Value.....	44
5.10 The Shell Fragment Program.....	46

5.11 Interface Classes.....	47
5.12 Interface Screenshots.....	47
5.13 Code for Vertex Selection.....	49
6.1 Illustration of Fins in the Final Product.....	50
6.2 Hair Colour Variance.....	51
6.3 Fur Type Interpolation.....	52
6.4 Dynamics in Action.....	52
6.5 Stanford Bunny.....	56
6.6 Slowest Attainable Viewpoint.....	56
6.7 ATi Chimp Demo.....	58
6.8 Black & White 2 Model.....	58

List of Equations

2.1 Finding the Lighting Normal.....	6
2.2 Diffuse Lighting Equation.....	6
2.3 Finding the Halfway Vector.....	6
2.4 Specular Lighting Equation.....	6
2.5 Banks Self Shadowing Approximation.....	8
2.6 Texture S-Mapping.....	11
2.7 Texture O-Mapping.....	11
2.8 ABF Minimization Equation.....	12
2.9 – 2.11 ABF Constraint Equations.....	12
2.12 Calculating Wind Vectors.....	14
2.13 Calculating the Hair Vector Factor.....	14
2.14 Calculating the Wind Vector Factor.....	14
3.1 Finding Silhouette Edges.....	18
3.2 Alpha Fins Equation.....	18
3.3 Lighting Factor for Shadowing.....	19
3.4 Fur-over-fur Shadowing Equation.....	20
3.5 Pixel Qualifier for Hair Interpolation.....	21
3.6 Hair Thinning Equation.....	22
3.7 Gravity Vector Equation.....	24
3.8 Movement Wind Vector Equation.....	24
3.9 Momentum Force Rule.....	24
3.10 Acceleration Rule.....	24
3.11 Momentum Vector Equation.....	25
3.12 Calculating the Total Force Vector.....	25
3.13 Calculating the Acting Force Vector.....	25
3.14 End Point along Wind Vector.....	25
3.15 End Point along Hair Vector.....	25
4.1 Hair Count Approximation.....	31
4.2 Ray-Plane Intersection.....	35
4.3 Location of Ray-Plane Intersection.....	35
4.4 Customised S-Mapping.....	35
5.1 Object-over-Fur Shadowing.....	44
5.2 Constructing Selection Ray.....	48
5.3 Ray-Sphere Intersection.....	48

List of Tables

4.1 Data Generated for Each Hair.....	31
4.2 Fast Texturing Test Results.....	33
4.3 Complete Texturing Test Results.....	33
4.3 Testing Texture Application on Various Models.....	37
5.1 Input Parameters for Shell Vertex Program.....	43
5.2 Input Parameters for Shell Fragment Program.....	45

Chapter 1: Introduction

The primary aim of interactive 3D worlds is to create an immersive, believable environment for the user to navigate. For this to be achieved, realistic people and animals must be created and made to interact with the world. One major problem that must be resolved to achieve this is that a vast majority of these creatures will exhibit some kind of fur or hair that must be controlled and rendered in real time.

Due to the volumetric nature of fur, it is very rare that an attempt is made to replicate it in real-time systems, although there has been a slight increase in recent years as GPU's have become more powerful. The main purpose of this project is to research, design and build an implementation of real-time fur that has the potential to be used in any interactive system. There are countless different kinds of fur present in the animal kingdom, and it would be impossible to create a single system that could render them all. Having considered the type of furry creatures that are most likely to be included in a real-time system, it is concluded that short fur is to be the focal area. Therefore, the resulting methodologies of this project should be tailored accordingly.

Previous works in the field have provided several techniques for the rendering of fur, and will be analysed to find the best approach. There is also two defining areas that seem to be absent from previous work which are to be considered during the course of this project. These are the concepts of realistic dynamics, and transitions between numerous hair types on the same creature. The latter is an important problem as it produces obvious breaks in the image when producing mixed type animals (for example, a monkey with thick black hair on its head, but only small finer hairs around its face, would produce a rendered image where the fur types will change suddenly around the face). Both these features are considered to be extremely important in producing a realistic representation, and are therefore specified as secondary objectives for this project.

There will be a number of constraints in place upon the system. The most important of these being that the frame rate must remain above 15fps (widely accepted as the absolute limit for a system to be considered interactive [1]) with both the hair type and dynamic options implemented. Ideally a frame rate of 30fps or higher should be achievable on a 1,000 polygon base model. This would present reasonable evidence that the methods were sufficient for use in more mainstream systems such as games and simulators. The project will be tailored towards the Radeon9600 range as this gives the largest scope for GPU based programming. It is also a reasonable target platform as the fourth generation cards will be common place by the time a solution could be implemented in a real-world application.

The paper is organised as follows: Chapter 2 considers the achievements of previous work in the field, considering the visual appearance and efficiency of the techniques described. The most appropriate methods for this project will be indicated. Chapter 3 then analyses any problems with these techniques, and suggests adaptations for use in this project. In cases where no suitable technique could be found, one or more conceptual solutions are proposed. Chapter 4 the design and implementation of FurMak, a pre-processor for the system, is discussed. This is then followed by a discussion around the design and implementation of FurGLE, the main rendering program, in Chapter 5. Chapter 6 then evaluated the achievements of the project in detail, indicating the scope for further work. Finally, Chapter 7 draws a conclusion on the project as a whole.

Chapter 2: Literature Review

Real-time fur has been a research topic for quite some time, resulting in a wide range of research papers that will be of use in this project. This chapter aims to discuss this past work, along with that in similar areas, before considering the suitability of the methods suggested.

Section 2.1 considers the different methods used in the past and present for the modelling and rendering of fur. Section 2.2 then looks into the chosen method in more detail, considering different techniques used for length, combing, colouration and lighting. Section 2.3 discusses two approaches for the offline generation of fur for use in the real time system, followed by an investigation on how to apply these textures in Section 2.4. Finally, Section 2.5 looks at the work previously done in the area of fur dynamics.

2.1 Approaches to Modelling Fur

In the early days of 3D Graphics, fur was rendered using a Brute Force approach. Whilst this produced good results it was extremely expensive, and required hundreds of man-hours to model and animate each hair. Thankfully, the improvement of technology has allowed more efficient methods to be produced, many of which can be optimized for real-time use. The most viable of these are discussed in this section.

2.1.1 Polygon Strip Approach

The most common approach used commercially is that of polygon strips (not to be confused with stripification). With this technique, side-view texture maps of fur are generated and displayed on polygons vertical to the skin surface. These are then layered behind each other giving the impression of dense fur.

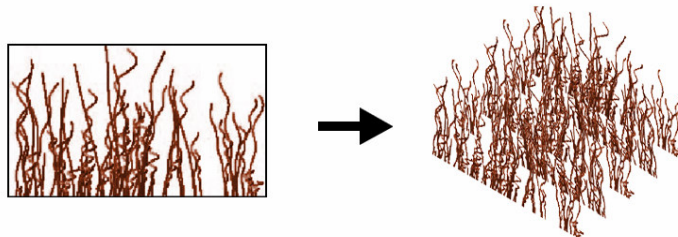


Figure 2.1

Representation of how polygon strip fur is produced from side-view texture maps.

(Fur texture modified from [2])

The problem with this technique is evident from the illustration above. If the displayed patch of fur was viewed from the side or from directly above, then the infinitely thin polygon strips would not be visible, resulting in no fur being rendered.

This can be resolved in two different ways. The first is to use a method known as ‘cross-hatching’, where strips are arranged at a multitude of angles across the skin surface, resulting in fur being visible from any side-view angle. The individual strips can also be broken up into a number of non-vertical segments to produce wavy hair, and allow overhead views to be accounted for. Whilst this can produce good results if the polygon strips are small enough and arranged correctly, it results in extremely complex and intensive geometric models.

Another approach is to incorporate a process known as Billboarding. This process rotates any given polygon strip to always face towards the camera, ensuring a suitable coverage of rendered fur from any side-on viewpoint. One problem with this method is that the polygon strips must be quite small so that the rotation of each strip in respect to the model orientation is not perceptible to the user. The main problem however is that the billboards can only rotate around an axis normal to the face upon which it is placed, making overhead views with this technique impossible.

2.1.2 Texels Approach

(Note: This method was developed before the term 'texel' was coined to represent a texture element. For the rest of this sub-section only [2.1] the term 'texel' will refer to Kajika & Kays original definition.)

One of the best methods to date, by way of visual quality, is the texels model developed by Kajiya & Kay in 1989 [3]. Here the texels concept was introduced; a texel being a 3D-Array storing approximations to the density and lighting properties of points in a volume space (similar in concept to modern day voxels). A ray-marching algorithm then travels through this array from each screen pixel combining the parameterised density and lighting values from each texel-element in order to produce a final screen colour.

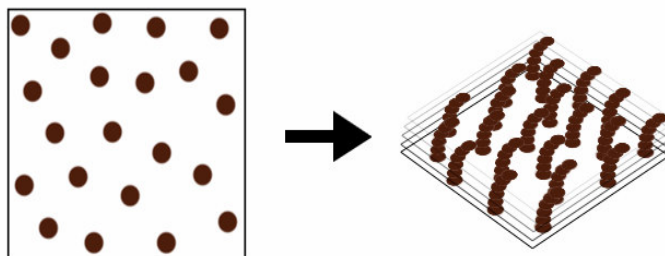
Whilst this method produces very good results, it is extremely slow and requires large amounts of memory to store information for every texel in the virtual world. Therefore it is not suitable for real-time use, but a number of its concepts can be 'borrowed' for implementation in other techniques.

2.1.3 Shells Approach

In 1998 Meyer and Neyret adapted the texels approach for real-time rendering using traditional texture mapping methods supported by graphics card hardware [4, 5]. Lengyel further adapted this two years later to form the shells approach for fur rendering [6].

The basic premise behind this method is an adaptation of the texels approach to utilise traditional texturing methods. Meyer and Neyret realised that by rendering a series of layers, or shells, upon which cross-sections of a fur 'patch' are rendered, the result mimicked a minimized texel volume around the model surface.

Each cross section of the fur volume is stored as an alpha texture map generated 'offline' prior to display. These can then easily be rendered at increasing distances from the base model to represent a 3D volume of fur. Figure 2.2 illustrates this concept using a single texture map for simplification, but in practice a different mask is used for each layer allowing a wide range of curly hair patterns to be produced. The positioning of each shell is obtained by an equal-step displacement along the normal of each vertex.



A major advantage of this property is that combing can be represented simply by changing the normal vector at each vertex.

Figure 2.2 -
Representation of how the shells
method can represent volumetric
texture using 2D texture maps.

The visual quality of this method is highly dependant on the number of shells. Higher shell counts will obviously give a 'fuller' volumetric texture, but will greatly increase the cost of computation. Tests show that good results can be achieved with just 8 shells in most situations, but the extra spacing between these layers can produce breaks in the hairs when viewed at angles over 45° from above.

The main problem with this method is that even if the shell count was increased, it would still break down when viewed from the side due to the infinitely thin shells not being rendered. Lengyel acknowledged this problem, and through collaboration with Microsoft [7] the concept of fins was introduced.

The fins technique extends the shells model by drawing polygon strips between each base polygon edge and the corresponding edge on its uppermost shell (one of which is illustrated in Figure 2.3). For efficiency, these strips are only shown towards the silhouettes of the model where the gaps between shell textures are most evident. Whilst this is still not as good as the full polygon strip model for side-on viewing, it does produce acceptable results.

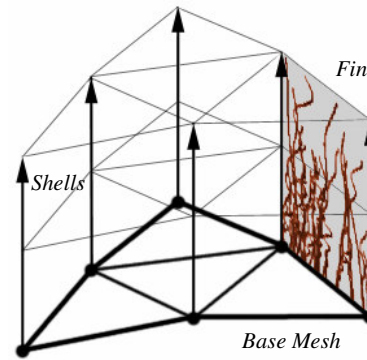


Figure 2.3
The correlation between shells and fins

2.1.4 Summary

After consideration, neither of the polygon strip techniques seem suitable for real-time fur rendering. The crosshatching method would probably produce the best visual result out of the given methods, but the intensity and complexity of the geometry required would result in slow rendering times. This arrangement would also make fur type interpolation and dynamics extremely difficult to compute. This method would be better suited for primarily side-view volumetric rendering such as real-time grass [8].

The bill boarding approach would be more efficient than the crosshatching method, but has no support for top-down viewing. Considering that the majority of visible fur on a creature will be pointing within 45° of the viewing angle this technique can be instantly ruled out. This method would be better suited with longer hair where end-on viewing is very rare. [9]

The shells and fins technique, with its focus towards overhead viewing is chosen as the best choice for this system. It is also theoretically more efficient than a polygon strip model of the same visual quality. The only major restriction with this method is that the fur can only be combed to a maximum of 45° , but in most cases this will not be a problem.

2.2 Shell and Fin Variants

The shells/fins technique has been implemented a number of times since its introduction with slight variations in some of the corresponding techniques. This section considers the various methods used for the length, colouration and lighting aspects of fur rendering.

2.2.1 Length

In their defining paper [7] Lengyel et al. suggest that a value representing the local length of fur is stored at each vertex. The shells are then spread evenly along the vertex normal between the vertex world-space co-ordinate and the point of magnitude equal to the length value along the vertex normal. The number of shells remains constant throughout the model regardless of any local length values, and the fins always stretch between the skin and the uppermost shell, ensuring correct silhouette fur length.

Due to the rapid increase of GPU processing speeds, sufficient polygons can be produced to represent almost any fur variations in real-time with this model. As a result, no other methods have been proposed to extend this technique.

The only area that is not accounted for by this method is with formations such as whiskers, or the tufts on a lynx's ears, where extremely large changes occur over a very small area. Whilst this could be modelled using this method, it would require extremely small polygons and would be extremely expensive for the result obtained. The graphical quality would also be very poor, as can clearly be understood when you consider rendering a long cat whisker using only 8 or 16 small circles.

2.2.2 Colour

Real fur is not all the same colour. Each individual strand varies ever so slightly from those around it, and in some cases can be very different indeed (e.g. the occasional grey hair). This trait needs to be modelled within the system to improve the realism of the final output. There have been two key methods proposed to date in order to achieve this:

The Lengyel Method [7]

This is the simpler, and older of the two models. Each layer of a shell has its own texture map that indicates both the existence and colouration of a hair at any given point. That is, each texel on a shell's map either contains the colour of a single strand of fur at that point, or a full alpha value indicating that no hair is present in that area. Whilst very versatile, allowing hairs to change colour in any way the developer could want, this is very memory intensive, using a full colour texture for every shell in the entire model.

The ATi Method [2]

This method is an adaptation of the previous, devised by Isidoro & Mitchell for ATi. Using this method only a single colour texture map is made for each section of fur. Each shell of the section therefore only has to contain a binary value at each texel representing the presence or lack of a hair. That is, every shell texture is a stencil mask used to 'cut the hair' from a single colour map.

This has two main advantages. Firstly, only a single colour map is required for the base model, with less memory intensive binary maps for the shells. Secondly, because the shell maps contain no locally specific data, the same maps can be used on every polygon on the model that is host to the same type of hair, saving even more memory and reducing the offline processing time required prior to the real-time system.

There was, however, a major problem in that curly hairs can change colour as they progress away from the skin (Figure 2.4a). To resolve this problem, the concept of Offset Maps was introduced (Figure 2.4b). In this map, each texel represents how far each hair has diverged from its root location, storing the x/y distance in texels in the red/green components of the image. Using this map the system can reference the correct element from the colour map for each point (Figure 2.4c). Like the shell maps, the offset maps contain no local specific data and can therefore be reused for every polygon of equal fur type.

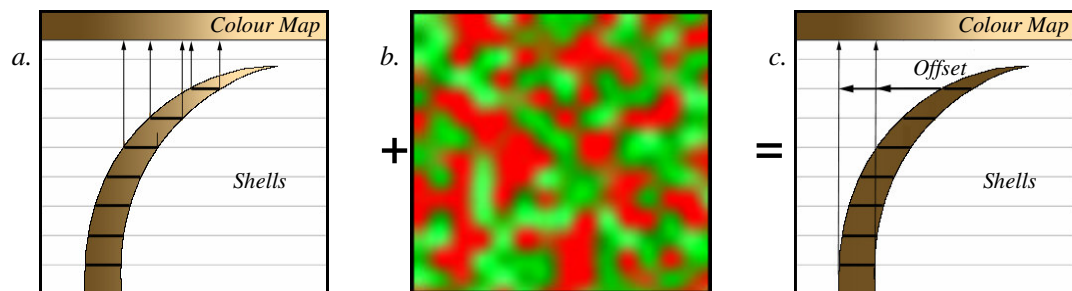


Figure 2.4 - An illustration of the need and implementation of offset maps

With modern day graphics cards this look-up can be done with such efficiency that there is little difference between the results from this method and the direct referencing of the Lengyel approach. This makes the ATi approach the obvious method to adopt for use in the system.

2.2.3 Lighting

Lighting is possibly the most important factor in rendering realistic images. For fur this consists of three key parts; the traditional diffuse and specular components, and the volume texture specific self-shadowing component.

Diffuse

In their defining paper [3] Kajiya & Kay defined a diffuse lighting model based upon the standard Lambertian model, which was later improved by Stalling et al [10]. This method has proved to be accurate enough that no alternative method has been provided, and therefore will be adopted for the project.

It is important to realise that because the hairs are being modelled as a collection of 2d circles rather than true cones, it is not possible to light each side of the hair individually. Even if this was possible it would be totally impractical because each individual hair 'slice' will cover only a very small screen area (rarely more than 4 or 5 pixels), meaning this extra lighting detail would not be noticeable enough to warrant the extra calculations. Therefore we assume that the overall light intensity is equal to the maximum light intensity present on any side of that hair.

To calculate this, we first have to define which side of the hair will host the maximum light intensity. We do this by finding the normal to the hairs direction vector that is closest to the incoming light source. This can be calculated by mapping the light vector to the normal plane using the following equation (This is illustrated in Figure 2.5)

$$N = L - (T \cdot L)T \quad (\text{Eq. 2.1})$$

where L is the vector towards the light source,
and T is the directional vector of the hair.

Having now got the normal to the 'surface' we wish to light (i.e. the side of the hair, rather than the cross-section) the light intensity can be calculated by substituting the previous equation into the Lambertian diffuse model, giving us a final diffuse equation of:

$$D = (L - (T \cdot L)T) \cdot L \quad (\text{Eq. 2.2})$$

Specular

The accompanying method by Kajiya & Kay [3] for specular lighting is also the most frequently used. As with the diffuse model it assumes the maximum intensity value to be true for a given hair section, and utilises the Lambertian lighting model. It is clear, therefore, that we must first calculate the halfway vector (H) between the light source and the viewing angle using the equation:

$$H = (L + V) / (||L + V||) \quad (\text{Eq. 2.3})$$

Where L is the light vector and V is the view vector

By substituting H into Equation 2.1, and using the Lambertian specular lighting model, we can define the final specular equation as being:

$$S = ((H - (T \cdot H)T) \cdot H)^n \quad (\text{Eq. 2.4})$$

Where T is the directional vector of the hair and H is the halfway vector.

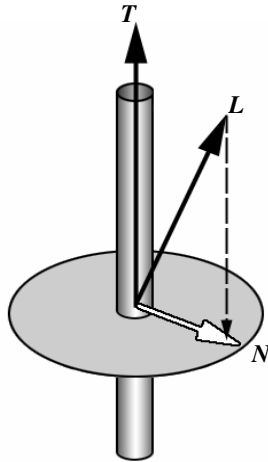


Figure 2.5
Visualisation of diffuse lighting vectors

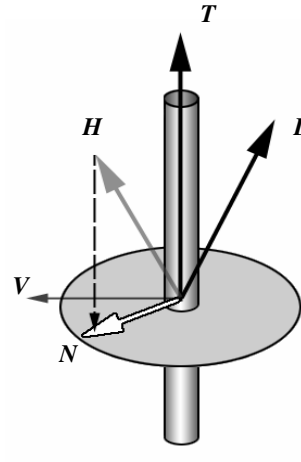


Figure 2.6
Visualisation of specular lighting vectors

The alternative to the Kajira & Kay lighting model described above is the use of pseudo-reflectance maps [11]. This is an offline approach that considers a number of samples on the normal plane (Represented in Figures 2.5 & 2.6 as grey circles) using the Kajira & Kay lighting equations (Eq. 2.2 & 2.4). These samples are then averaged and stored in a texture map for use in real-time. Whilst this produces extremely good results, a new texture map must be generated every time the camera moves. However, as this project is focused towards dynamic furry creatures for use in interactive systems, this method is clearly unsuitable.

Defining normals

The Kajira & Kay methods work perfectly for straight fur where the hair vectors are defined at vertices and interpolated across the polygon. However, our system will be using curly hair, with many individual strands pointing in vastly different directions across a single shell. Therefore a method to define normals for each individual hair is required to allow per-pixel lighting to be implemented.

A solution for this problem is proposed by Isidoro/Mitchell[2] and involves using a normal map for each fur shell. Here the x,y,z values at each point are represented by the r,g,b values of a texture map. Using this method, for any u,v point on a shell mask, the same u,v co-ordinate can be referenced from the normal map to obtain the local hair vector for use in the lighting equations. Like the mask and offset maps, the normal maps do not contain any locally specific data and can be reused with any polygon on the model that is host to the same type of hair. Any singular collection of these three maps for a type of hair will be collectively referred to as a *type set* for the remainder of this paper.

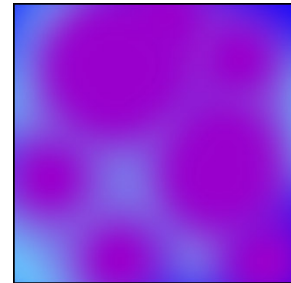


Figure 2.7
An example of a shell normal map.

2.2.4 Self Shadowing

Due to the volumetric nature of the fur, there will almost certainly be some degree of self-shadowing in any given situation. That is, the top of the hairs will block incoming light from reaching the base resulting in a volumetric shadow. Calculating the true shadow for each hair would take far too long even for offline rendering, so an effective 'cheat' must be found. There are two main approaches for solving this problem within the shells/fins methodology.

Banks Self-Shadowing Approximation

This works on the simple basis that the closer to the skin a segment of hair is, the darker it will appear. This gives a range from no shadowing at the top (i.e. full light), to the darkest possible shadow at the base (i.e. a minimum light value). This ‘shadow factor’ can be calculated using:

$$S = (D_{cur} / D_{max}) * (1 - S_{min}) + S_{min} \quad (\text{Eq. 2.5})$$

Where D_{cur} is the current shell, D_{max} is the number of shells and S_{min} is the minimum ‘shadow factor’.

The existing light value can then be multiplied by this shadow factor to give the ‘true’ light value. This method is extremely fast, but lacks in quality with no consideration of fur density or the angle of incoming light, both of which will have a large affect on the resulting shadow area. These conditions can be simulated by changing the S_{min} value for each polygon, but it would be much more efficient if all these considerations were combined in a single formula. The application of shadow is also unrealistic, as shadowing is rarely of linear increase along the full length of a hair.

Another major drawback is that it only considers hair-over-hair shadows, and not the object from which they are protruding. Because of the manner our lighting normals are calculated, hairs that point directly towards and directly away from a light will calculate the same lighting intensity. That is to say, the hairs on the opposite side of an object than a light source will be lit as if the light is pointing directly at them. It should be the responsibility of the self-shadowing algorithm to rectify this problem and ensure that object-over-hair shadowing is implemented correctly.

Shadow Maps

An alternative to the Banks approach is a process called Shadow Mapping[12]. This method generates approximation shadows for each individual hair by using multi-texturing. Starting from the top-most layer with an empty shadow map, for each shell layer the shell map is combined with the current shadow map to produce a shadow map for the next shell down. This shadow map is then displaced along the light vector to the correct shadow location for the next layer. The shadow map is a binary map that either darkens (shadows) a pixel, or leaves it at its current value. This is then repeated down the fur shells until the skin is reached and a shadow for the entire fur patch is cast.

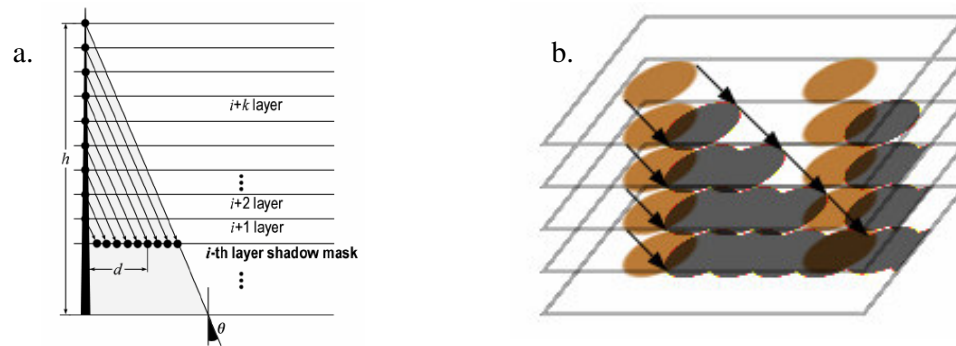


Figure 2.8 – a) The shadow maps approach to self-shadowing (Used with permission from [12])
b) An example of implementation on a 2 hair shell set.

This method produces extremely good results for vertical shells, and by calculating the shadow ‘shift’ at each vertex of a polygon it could easily be adapted for combed hair. Its major downfall is that to be applied to a multi-polygon model it must use lapped textures to pass the current shadow map to its neighbouring polygons (as the offset will almost always push it over a polygon edge). Without this lapping of shadow maps, each polygon would have one or more edges with no shadowing producing a ‘broken’ image. This lapped texture solution is viable with flat surfaces, but if the surface were curved some complex buffering and transforming would be required to correctly lap the textures. Like the Banks approximation, this technique fails to support object-over-hair shadowing.

2.2.5 Summary

The Lengyel method for representing hair lengths at vertices is well suited for use in real-time fur systems. The only limitation is that large variations over small areas, such as whiskers or tufts, are not supported. A target of this project is therefore to provide a solution for small details such as these to meet the project requirement for rendering a large range of fur types.

The ATi approach to colour definition is a very suitable model, and is adopted as the chosen method for this project. The only minor limitation is that variations over hair length, such as coloured roots, are not supported in this method. However, this is only a minor issue as natural roots are rarely visible on animals and would be too small to warrant the extra cost for rendering in most cases.

The Kajika/Kay lighting models for ambient and specular lighting are also used due to their simplicity and well-documented visual quality. However, the shadowing method is a more difficult area. Neither as of the proposed methods are suited for full model implementation; a problem which is to be addressed within this project. The Shadow Maps approach would be of much higher quality, but would require a complex solution that would take too much time to warrant inclusion. Therefore the simpler task of adapting the Banks Approximation is taken, allowing more important areas to be focussed upon.

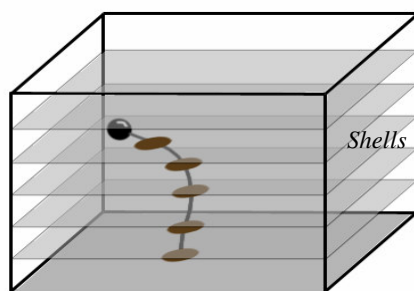
2.3 Texture Generation

To draw all the mask, normal, and offset textures for each shell on every fur type of a model would be a long and tedious task. Therefore a texture generator should be included to calculate these maps in an offline process given information about the fur such as thickness, density and curliness.

The most complete fur generation tool to date is called FurGen and was created by ATi for use in their card-specific fur system. Unfortunately details on how this operates have not been published, and the product is not commercially available. Therefore, this section considers two existing methods for texture generation, and discusses the potential for adaptation to produce the type sets for this project.

2.3.1 Particle System model

This is the method used by Lengyel [6, 7]. A 3D bounding box is defined with the width and depth of the required map resolution. The height is then determined as a multiple of the number of shell textures required. A number of horizontal planes (called shell boundaries) equal to the required quantity of shells are then evenly spaced up this bounding volume. A simulation system then generates fur, a hair at a time, by sending a particle through this bounding box from a randomised



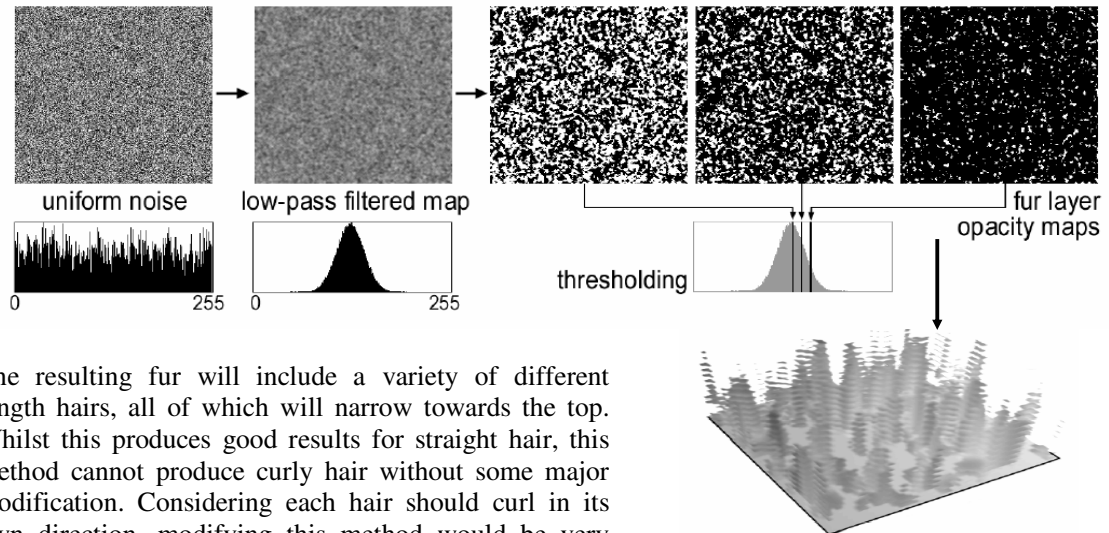
Bounding box

base location. Every time it hits a shell boundary, a circular mark is made on the corresponding shell maps to indicate the presence of a hair at that location. The density, curliness and length of the fur can be controlled by varying the quantity, path randomisation, and lifespan of the particles respectively.

Figure 2.9 – Generating textures using a particle system.

2.3.2 Noise model

Another method of generating the fur textures is the Noise Model, devised by Papaioanou [12]. Here, uniform noise is generated to produce a random greyscale texture of the same dimensions as the required maps. This is then passed through a low-pass filter and a varying threshold to produce the shell textures as shown in Figure 2.10.



The resulting fur will include a variety of different length hairs, all of which will narrow towards the top. Whilst this produces good results for straight hair, this method cannot produce curly hair without some major modification. Considering each hair should curl in its own direction, modifying this method would be very difficult, as there is no way of knowing where each individual hair is for producing the offset maps.

Figure 2.10 – Generating textures from noise (Used with permission from [12])

2.3.3 Summary

The Noise model, whilst the faster of the two methods, is unsuitable for this project due to its inability to produce curved hairs. The Particle System on the other hand already supports curly hair, and will be extended to create the offset and normal maps required to render the hair correctly. Some problems may develop however due to the inclusion of these extra maps; for example, if two hairs happen to cross a shell at the same location it would create problems with the normal maps, as the first hair's direction vector will be overwritten with the latter, resulting in incorrect lighting at run-time. These issues are considered in more detail later in this paper.

2.4 Texture Application

Another factor that will have a large affect on the overall quality of the system is the way in which the type set textures are applied to the model. Ideally the pattern should repeat continuously at equally regular intervals across the entire model. However, unless the model is a planar surface, this is not possible. This section looks at existing methods to specifying texture co-ordinates with this problem in mind.

2.4.1 Inverse Mapping using an Intermediate Surface

The simplest and most common approach to texturing an object is via an intermediate surface. This is a two stage mapping process, both of which have a number of variants dependant on the application required.

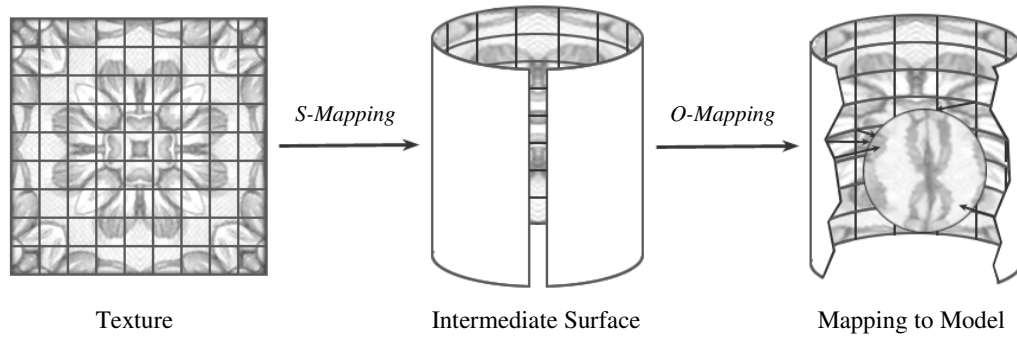


Figure 2.11 – An example of inverse mapping using a cylindrical intermediate surface and intermediate surface O-mapping

The first stage, known as the S-Mapping, consists of mapping the 2D texture to a simple 3D shape to convert the texture co-ordinated into a 3D domain. This intermediate shape can be any shape that can be represented mathematically, such as, but not limited to, a plane, sphere, cylinder and cube. The S-Mapping is a function of type:

$$T(u, v) \rightarrow T'(x_i, y_i, z_i) \quad (\text{Eq. 2.6})$$

The second stage, known as the O-Mapping, then maps the 3D texture representation onto the object surface. For pre-texturing a model there are three possible methods for this stage; object normal, object centred and intermediate surface normal. These methods are illustrated graphically in Figure 2.12. A fourth method, called reflected ray, is also available, but this is only suitable for run-time allocation or pre-processing of a static scene. The O-Mapping is a function of type:

$$T'(x_i, y_i, z_i) \rightarrow O(x, y, z) \quad (\text{Eq. 2.7})$$

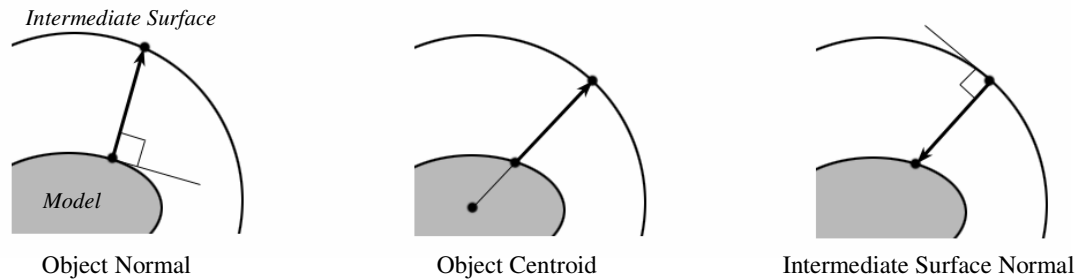


Figure 2.12 – The three 'static' O-Mapping methods

2.4.2 Lapped Textures

In their paper, Lengyel et al. adopted the lapped textures method for specifying their texture co-ordinates. This method, developed by Praun, Finkelstein & Hoppe [13], continually applies segments of a given texture to a model, aligned along a user-defined 'tangent field', until the model is completely covered. The name derives from the property that many of these texture patches will be overlapped by new ones as the full texture allocation is constructed.

An example of this is shown in Figure 2.13. The image on the left represents the tangent field on the model as specified by the user. The insert is the texture patch being applied, and the image to the right is the textured model.

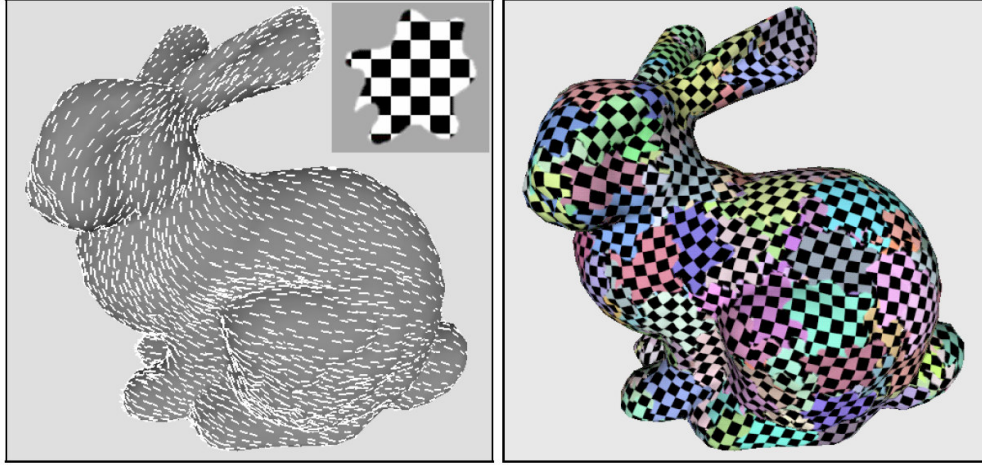


Figure 2.13 – Lapped textures
(Used with permission from [13])

This method evidently results in a well scaled texture application, but with a large amount of discontinuity where the separate texture patches connect. This property was acceptable with Lengyel’s implementation as it was focused towards dense, straight fur, resulting in minimal disruption where the discontinuities exist. A solution to the discontinuity, provided by Praun et al, is to blend the texture patches together, but this requires a unique texture to be generated for every patch on the model. Unfortunately, this would remove the non-localisation of our type set textures, resulting in unacceptably large quantities in texture data.

2.4.3 ABF + Overlay Grid Smoothing

Angle Based Flattening (ABF) and Overlay Grid Smoothing are techniques developed by Sheffer and Sturler [14, 15]. This works on the theorem that for any smooth surface, there exists a mapping of that surface to a plane, which is conformal [16]. Based on this theory, this method approximates a mapping from a faceted surface (i.e. a polygon mesh) to a plane (Angle Based Flattening). Once both the mesh and texture are in a 2d domain, the texture can be easily mapped onto the 2d model, then transferred back to the 3d mesh to produce a continuous texturing.

This stage is carried out by solving the ‘constrained minimization problem’, which when given the optimal angles within each polygon (Φ), flattens the mesh with minimal difference possible between the generated (α) and optimal angles. This is achieved by solving equation 2.8 with a Lagrange multiplier via Newton recursion, under the constraints indicated by equations 2.9, 2.10 and 2.11. These constraints ensure that the flattened mesh maintains correct connectivity by ensuring each face is valid, edges have the same length against either adjoining polygon, and that all the edges around each vertex is valid respectively.

$$\text{minimal value of } \sum_{i=1}^P \sum_{j=1}^3 (\alpha_i^j - \Phi_i^j)^2 (\Phi_i^j)^{-2} \quad (\text{Eq. 2.8})$$

$$\alpha_i^1 + \alpha_i^2 + \alpha_i^3 - \pi = 0 \quad (\text{Eq. 2.9})$$

$$\sum_i \alpha_i^{j(k)} - 2\pi = 0 \quad (\text{Eq. 2.10})$$

$$(\prod_i \sin(\alpha_i^{j(k)+1}) / \prod_i \sin(\alpha_i^{j(k)-1})) - 1 = 0 \quad (\text{Eq. 2.11})$$

Where α_i^j represents angle j in polygon i of α ,
 Φ_i^j represents angle j in polygon i of Φ ,
and $\alpha_i^{j(k)}$ is the angle at node k in polygon i of α

Once the mesh is flattened, a bounding grid representing the texture co-ordinate set is created. This grid is then warped, in a process called smoothing, by varying the length of each grid edge relative to change in local edge lengths in the flattened mesh. This results in a concentration of texture co-ordinates in areas of the original mesh that have undergone the greatest reduction in size.

When the overlay grid is complete, texturing the object is achieved simply by mapping the texture co-ordinate from the grid onto the flattened mesh. The co-ordinate is then copied to the corresponding node in the original surface to give a regular and continuously textured surface.

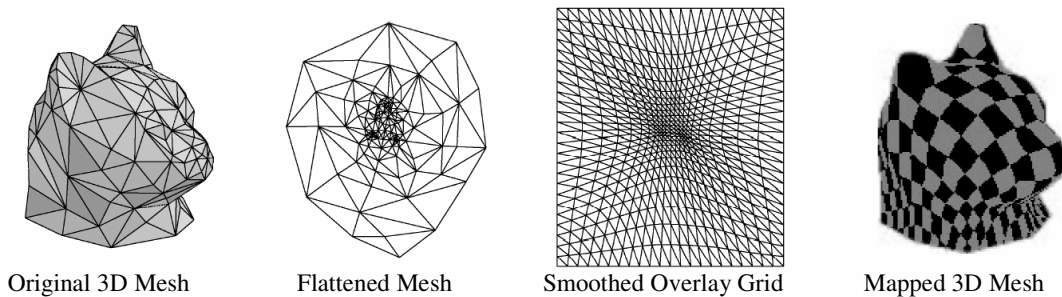


Figure 2.14 – ABF + Overlay Grid Smoothing example
(Used with permission from [15])

This method produces very good results as illustrated in Figure 2.14, and is a fully automated process, unlike lapped texturing. There is only one major problem with this method in that only faceted surfaces are supported, not solid meshes. This is because external edges must be present for the mesh to be flattened (for example, it would be mathematically impossible to flatten a sphere unless you made a hole somewhere in its surface).

2.4.4 Summary

Due to the intention of modelling sparse and curly fur, the continuity of the texturing is an extremely important factor. This is to reduce the amount of half-hairs on discontinuous boundaries, as shown in Figure 2.15. Therefore the lapped textures method can be dismissed as it is the method that presents the highest probability of this occurring.

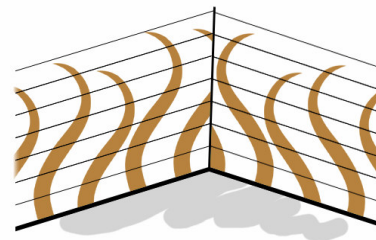


Figure 2.15 – Fur anomalies from discontinuous texture mapping

Evidently, the quality of ABF/OGS texturing is by far the superior option, but the inability to texture full ‘solids’ must be considered. One possibility is to ‘flatten’ the mesh onto a sphere, rather than a plane. However, this would then present the problem of how to texture a sphere without anomalies, which is mathematically impossible. A viable solution would be to simply split the model in two halves, like a chocolate Easter egg, and texture either side individually. These could then be merged back together, but would have a discontinuous equator where the textures do not meet correctly.

Considering this, the results from ABF and mapping via an intermediate surface would be similar in overall quality. Therefore the obvious choice is to use the intermediate surface approach as the mathematics involved are much less complex, and therefore more likely to lead to a robust implementation.

2.5 Dynamics

Dynamics on short fur is an unusual area. All the papers available are concerned with grass (the simplest case) or long hair dynamics (the most complex case). In the few cases where short fur dynamics is present [17] no details are available as to how it has been implemented. Therefore, a method must be developed by adapting the techniques used in grass and long hair dynamics.

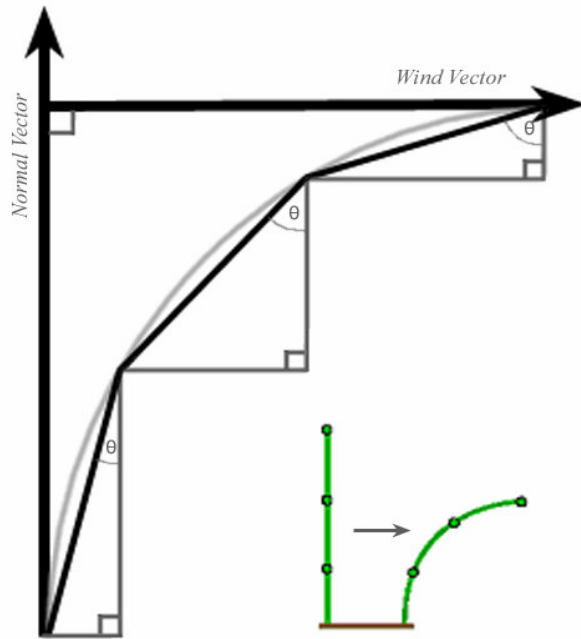
2.5.1 Wind Vectors

This method was originally implemented for use in modelling grass [18], but could be easily adapted to simulate fur dynamics. A wind vector, representing the resulting force of a global wind on a hair, is calculated for each vertex on the model by mapping the wind vector to the vertex 'true-normal' plane. (The same principle as deriving the lighting normal, see Figure 2.5)

$$W_v = W - (W \cdot N) N \quad (\text{Eq. 2.12})$$

where W is the global wind vector and N is the vertex normal.

By ensuring each wind vector is normal to the vertex normal, the offset for each shell vertex can be calculated using simple trigonometry. When normalised, the wind vector and hair vector form a 2D quad into which a curve approximation can be drawn. (Shown in Figure 2.16)



By sampling this curve, values can be obtained representing the distances the corresponding vertex for each shell should be moved along the hair and wind vectors. This results in a steady bending of hair along its full length in reaction to a defined force vector. Equation 2.13 defines the hair vector factor, and Equation 2.14 defines the wind vector factor for these relocation values.

As can be seen in the diagram, calculating the relocation values using this method preserves the hair length regardless of the direction and intensity of the force acting upon it.

Figure 2.16 - Deriving new vertex positions using the hair normal, and a wind vector.

$$W_f = \sum S \cdot \cos(I \cdot \pi \cdot i / 2 \cdot N) \quad (\text{Eq. 2.13})$$

$$W_f = \sum S \cdot \sin(I \cdot \pi \cdot i / 2 \cdot N) \quad (\text{Eq. 2.14})$$

where S is the distance between shells, I is the wind intensity, i is the current shell and N is the number of shell

The main problem with this method is that it only considers external wind, which is perfect for grass, but fur is susceptible to a much larger range of forces such as motion, momentum and gravity. There is also the problem that each polygon of fur will be animated as a whole with this method which, although efficient, may produce visibly unrealistic results when subjected to strong force

2.5.2 Loosely Connected Particles

This is a very recent method developed at the University of Tokyo [19] for long hair dynamics. A number of invisible particles are positioned randomly within the hair volume, and connected with 'dampened springs'. That is, each particle is set a rule that encourages it to remain within a certain range of its neighbouring particles.

Each particle is then animated independently in relation to the forces acting upon it, relying on the springs to maintain the correlation between hair segments. If large forces are inflicted upon particles towards the top of the hierarchy, the springs will pull all lower particles into place, resulting in realistic swaying of hair. Also, in the event that the forces acting on the hair is less than the spring strength, the hair shall be forced back to its natural position.

In the original work, the hair was then rendered by drawing billboards at each of the particles to build up a dense hair volume, but this could be used in the shells model if the particles were placed upon shell vertices.

Whilst this method gives good, fully volumetric dynamics, the mathematics involved are quite intensive and would be quite expensive considering the number of particles required for an entire animal model.

2.5.3 Summary

Whilst the more evolved of the two methods, the loosely connected particles system was developed for use with billboards allowing the particles to be placed almost anywhere upon the model. If this method was adapted for the shells model, the particles could only be placed upon the vertices, removing the benefits of the 'loose' structure. Also, due to the length of the fur, each vertex would host no more than 2 particles, effectively reducing the accuracy of any output to that of the wind vectors model.

Considering that both methods could, with a little work, produce equal quality results, it is sensible to go with the simplest method; in this case, the wind vectors method. It is important to note, however, that further work will need to be conducted to incorporate the full range of forces necessary to model dynamics on a moving furry animal.

Chapter 3: Analysis

Now that a good understanding of the project area has been developed, consideration of the project details can be made, along with further development of techniques specific to this implementation. All methods in this chapter are original work unless defined otherwise.

Section 3.1 reviews the project aims in more detail, which will form the basis of the evaluation later in the project. Section 3.2 recaps the most suitable methods from previous work, and suggests alterations to make them more focussed towards this particular project. Sections 3.3 and 3.4 then consider the key problems of hair types and dynamics. This is followed by plans for testing and evaluation in Sections 3.5 and 3.6, before Section 3.7 considers the tools that will be used in the project.

3.1 Aims of the Project

From initial ideas and research done, the following 4 categories of project aims have been defined.

General Fur Objectives

The primary aim of this project is to research, experiment, and present a method for rendering real-time fur. Suitable consideration should be made for colouration and lighting to produce as realistic a representation as possible. The ability to comb and vary the length of hair should also be present, further increasing the realism by allowing natural fur patterns to be represented. The presence of whiskers and tufts should also be considered.

Hair Type Specific Objectives

The secondary focus of the project is the generation, combination and display of numerous fur types. The chosen methods must be flexible enough to model any combination of the following properties.

- Natural or Synthetic fur
- Dense or Fine fur
- Thick or Thin hairs
- Curly or Straight hairs

An interpolation technique should also be implemented to generate realistic transitions between fur types on a single model. This project will not, however, deal with complex hair arrangements such as partings, swirls and cowlicks as this would require a major restructuring of the shells method.

Dynamic Objectives

The tertiary aim of the project is the implementation of realistic dynamics upon a furry model. The affect of gravity, motion and momentum should all be considered and rendered in a believable fashion. As this is the tertiary focus of the project, only the most common situations for furry creatures will be considered, such as gradual and sudden movement. Considerations of hair control (e.g. twitching whiskers), collisions and water interaction, for instance, would require a much more extensive physics engine and will therefore not be considered in this project.

Efficiency Objectives

Finally, the system should be efficient enough for interactive use. An absolute minimum of 15fps must be obtained for the project to be deemed a success, with an ideal target value of around 30fps rendering a 1000 polygon model on a machine with the following specification:

- P4 2.6ghz (w/ HT)
- 512MB RAM
- ATi Radeon 9600XT
- Resolution: 1024x768x32

3.2 Key Methods

First the general rendering methods will be considered, in relation to the General Fur objectives. Some consideration will also be given to the display of fur types in this section.

3.2.1 Representation

The key method behind the system will be the shells and fins approach as defined in the Literature Review. We shall now consider the finer details of this method for use in the system.

Shells

A test program was developed early on in the project to ensure the suitability of this method and test some theoretic aspects. It was found that using 16 shells produced sufficient quality images with relatively short fur, and will therefore be the default shell quantity. However, this should be determinable in case longer hair is required. Somewhat surprisingly, it was also found that increasing the shell count in excess of 32 shells results in a lower quality image, as the hairs start to look too solid and less fur-like.

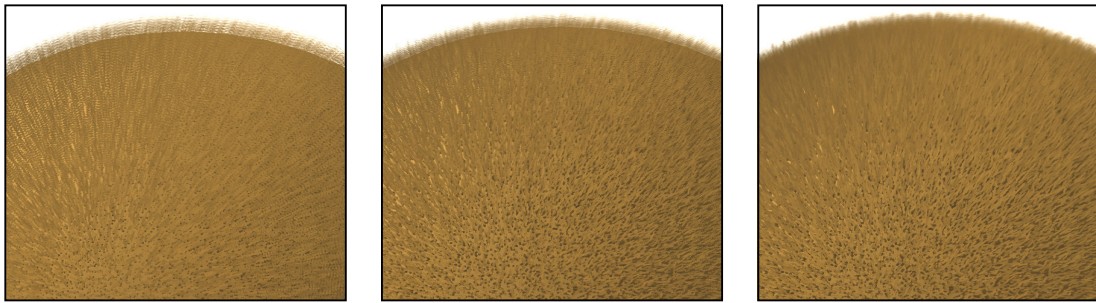


Figure 3.1 – Comparison of results using 8, 16 and 32 shells (from left to right)

Initially, it was hoped that 3-dimensional textures could be used to represent the fur volume. This would allow the number of shells to vary dependant on length and distance from camera (LOD), as the shell texture could be referenced at any height. In the case of distinct 2D textures, the number of shells must remain constant to the number of textures generated, as information between these cross-sections cannot be generated on the fly. Unfortunately, it became obvious early on in the development of the project that referencing 3D textures takes far too long when linear (min/mag) blending is enabled. Therefore this implementation utilises 16 different 2D textures to represent the fur volume, and the number of shells must remain constant across each model regardless of the fur length.

Early programming also found that the combing limitations are not as prominent as suggested in Lengyel’s paper when working with dense fur. This is due to other hairs being visible through the gaps between shells keeping the image full, if not a little fuzzy. Therefore the combing limitation will not be enforced as it would add unnecessary computing time. Ensuring the hair normals are in a suitable range will be the responsibility of the modeller, not the program, but extra care must be taken when working with sparse fur as the changes of gaps being filled by hairs further back in the pattern will be less lenient in these cases.

The shells will be rendered from the bottom up as specified in Lengyel’s paper. An attempt was made to render them from top down to avoid overwriting, by utilising the ability to directly manipulate the Z-buffer. However, this produced horrible aliasing effects where ‘half-pixels’ were not drawn. This made the resulting image flicker unbearably whenever the model was moved. The top-down approach also resulted in only the tiniest of performance increases because graphics cards cannot, at this point in time, discard a fragment until the process of calculating it has fully completed. Thus, pixels at lower levels would still have their lighting, colour, etc. calculated before being discarded, even if it was known it was going to be discarded early on in the calculations.

Fins

In their paper [7], Lengyel et al. placed fins between every shell and altered the alpha values to reduce the visibility of fins that point towards the camera. This is because such fins clearly stand out when viewed from near above (see figure 3.2) Early tests suggested this was unnecessary and that fins were only necessary on silhouettes edges (figure 3.3).

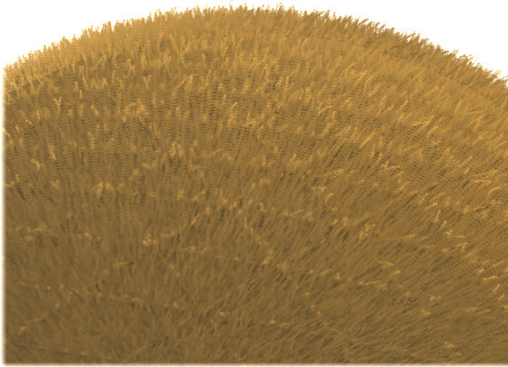


Figure 3.2 – Limitation of fins viewing angle

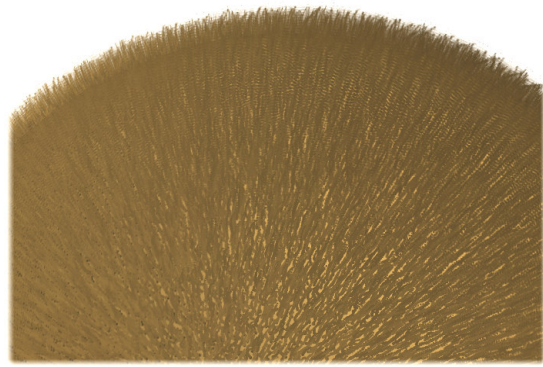


Figure 3.3 – Demonstration that fins only at silhouettes is suitable.

The benefit of doing this is that much fewer polygons have to be drawn, making the solution faster. Unfortunately, shortly into the development process it was realised that using only silhouette fins produced unwanted effects during animation; as the model was moved around, the fins could be seen changing state, creating popping affects around the silhouettes.

To solve this, the alpha-fins method was reintroduced to fade the fins in gradually as they approach the silhouette. The fin culling was still performed to reduce the number of polygons, but with a greater leniency. The equation used to do this is an adaptation of the process used in cel shading, as is shown in equation 3.1

$$\text{If } (N_1 \cdot V) * (N_2 \cdot V) < 0.3 \text{ then edge is a silhouette edge (Eq. 3.1)}$$

where N_1 and N_2 are face normals, and V is the view vector.

The alpha values assigned to the fins will be calculated using the following equation so that they are fully opaque when tangential to the camera, and fully transparent at the angle at which they are disabled.

$$\text{Alpha} = 3(N \cdot V) \text{ (Eq 3.2)}$$

Where N is the normal to the fin polygon, and V is the view vector

As the fins are only a support method to the shells, and will generally be further back in the image, they shall be rendered first. Also, because they require blending (due to the inclusion of alpha) they will be sorted prior to rendering from furthest to nearest.

3.2.2 Whiskers and Tufts

A review of the length abilities of the shells/fins method identified the problem of whiskers and tufts. To solve this, the concept of ‘Rogue Hairs’ is introduced. A rogue hair is a collection of 8 points in space connected by straight lines to generate a hair. The thickness of this line will be proportional to that of the fur type present at that vertex. Small tufts will be rendered by placing several rogue hairs at the same vertex, each with varying lengths.

Only the direction and length of the rogue hairs will be provided. The positioning of the hair points will then be calculated using the same method as present on the shells. Colour will be a defined value stored with the vertex and lighting will be done using the Kajira/Kay method (Equations 2.2 and 2.4)

Although these rogue hairs will be more expensive to render per-hair than those covered in a shell structure, the numbers will be considerably less, and the visual impact will be strong enough to warrant the extra cost.

3.2.3 Colour

Early tests have shown that the offset system produces good results over curly hair. The only problem with it is that each hair takes its colour from a single texture map which may contain very little variance in colour over some areas. To increase slight variations in hair colour, and thus enhancing the visual quality, the concept of hair variance is introduced. This is another map stored with the type set containing slight variations in colour, local to the fur texture.

In addition to hair colouration, the inclusion of fine fur in this project means the skin will be visible in many cases and will therefore require a texture map itself. This will be stored and rendered on the base model. To increase realism, the lowest shell mask can be used to darken the points on the skin texture where hairs will be. This will give the impression of pores in the rendered image with only minimum processing cost.

This introduces a problem, in that the colour map for the fur will be under a different texture space than the offset texture. This is due to the texture set being repeated many times across an object, whereas the colour mask must be a single, non-repeated texture. To resolve this problem, a 4-element conversion vector can be pre-calculated for each polygon, and used to map between the two texture spaces.

The first 2 elements of this vector would be the u and v displacement along the colour map for a single step along the u axis of the fur type maps. Likewise the second two elements would be the offset for each increment along the v axis. Whilst this would work, the extra data being sent to, and manipulated by, the GPU would greatly increase the render time. As it is very rare to have animals with sudden changes in fur colour it was decided this extra cost did not warrant the barely noticeable correction in fur colouration, so offset mapping is absent from our implementation.

3.2.4 Lighting

The lighting of fur in the project will be performed using the Kajira/Kay method (Equations 2.2 and 2.4) in conjunction with normal maps to present realistic per pixel lighting. There is, however, the problem of correct self-shadowing as raised in the review stage. The shadow mapping method would produce better results for this if its problems were solved, but it would be both inefficient and exhaustive to develop in respect to the time allowed for this project.

Therefore, the proposed solution is an extension to the Banks approximation method, taking into consideration the angle of light and the density of the fur. That is, instead of shading from pure darkness to full brightness over the full length of the hair, this occurs between the root and a point along the hair where it has been calculated full light will be received.

This point is calculated in a 2-step process. Firstly the lighting factor is calculated using:

$$F = 1 - N \cdot L \quad (\text{Eq. 3.3})$$

Where N is the hair normal and L is the light direction

This tells us how much light is reaching a hair as a full entity, with a value of 0 for no shadowing and 1 for a fully shadowed hair. This then needs to be converted in to a shadow factor dependant on the density, and current displacement along the hair. An assumption made is that for fully dense hair, the fur-over-fur shadow will reach 75% of the distance along the fur when tangential to the lighting direction. This will then decrease linearly for less dense fur to a degree of no fur-over-fur shading for a 0 density type. This gives us a final fur-over-fur shadowing equation of:

$$S = ((4h/d) - 3F) / F \quad (\text{Eq. 3.4})$$

Where h is the displacement along the hair in range $[0(\text{root}) - 1(\text{tip})]$
And d is the density of the fur of range $[0-1]$

The secondary objective of the shadowing model is to ensure that hairs facing away from the light source receive no lighting. This can be done by rendering full shadow along any hair segment where the dot product between the light and the hair direction is less than zero (i.e. facing away from the light)



The fins will also contain a normal map for lighting and will use its vertical edges rather than the polygon normal to define the hair direction. This hair direction can then be interpolated across the face of the fin, and the lighting performed using the same equations as for the shells.

Figure 3.4 – Obtaining the Hair vector for lighting fins

3.3 The Hair Types Problem

Due to the volumetric aspect of the shell layers, the type sets cannot be linearly interpolated as would be possible with singular 2D texture maps. Taking this approach with volumetric hair would result in hairs becoming increasingly transparent over distance rather than reducing in numbers and length, which is obviously wrong. Therefore, the constraint that a shell mask texel cannot be a halfway value (i.e. it must exist or not) is added. This section describes the approach taken to enable transitioning of fur types under this constraint.

3.3.1 Hair Dominance

Fur interpolation in the real world consists of two key features. Firstly, the quantity of each hair type reduces in number as it progresses into the ‘territory’ of the opposing type. The second phenomenon is that the length of the fur generally gets shorter as its numbers decrease. These concepts are illustrated in Figure 3.5.

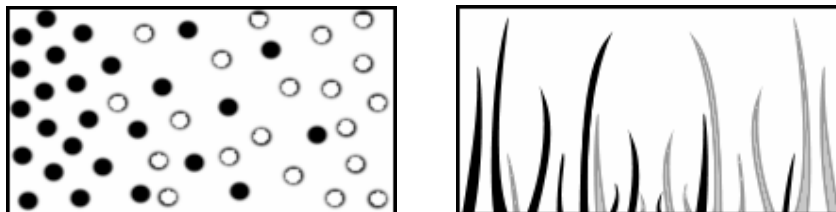
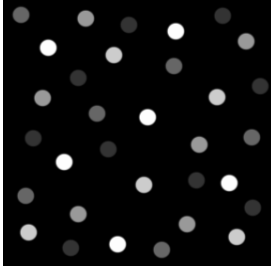


Figure 3.5 – Properties of realistic hair interpolation

In order to simulate these transitions, the concept of Hair Dominance is introduced. Here, each hair is given a value that determines how far into the opposing fur ‘Territory’ it can continue to exist. By determining how far a hair is towards its overlap limit (the point where the most dominant hairs cease to exist), its length can also be shortened accordingly without the need for extra data.



The hair dominance values will be represented by using a greyscale, rather than monochrome, image for the base shell mask. This will be referred to as the ‘dominance map’, containing element intensities between 0 (pure black) for no hair and 1 (pure white) for the most dominant hair. Storing this as a single base map ensures that a hair maintains the same dominance value along its length.

Figure 3.6 – Example of a dominance map (top), and its ‘filled’ counterpart (bottom).

3.3.2 Interpolation

The interpolation itself will be a multi-pass texturing procedure, rendering a single fur type at a time. For each fur type an existence value (0 for existence and 1 otherwise) will be defined at each vertex and interpolated across the polygon.

Then, for each pixel, the corresponding dominance map element is compared against the interpolated existence value. If the value obtained from the dominance map is greater than, or equal to, the existence value, then the pixel is deemed to exist, and shall be rendered. It should be clear to see that as the existence value increases (that is, the section being rendered gets closer to an opposing fur type) then the less dominant hairs will not get rendered, resulting in the thinning of hair.

Whilst this correctly reduces the quantity of hairs away from the type area, it does not reduce the size. To do this, the shells must also be clipped based on their level and existence value, giving us the following formula for determining whether to draw a given pixel:

$$\text{If } (S_c/S_t) \leq 2 - ((2 - 2e) / d) \text{ then render. (Eq. 3.5)}$$

*Where d is the dominance value from the dominance map,
 e is the existence value interpolated from the vertices,
 S_c is the current shell layer and S_t is the total number of shells.*

Note that the right side of this equation is multiplied by 2 from the simplest interpolation case. This is to ensure that the most dominant hairs do not begin reducing in length and density until half way through the transition. This is to reduce the illusion of hair thinning during the transition due to neither hair type being at full length.

This test is performed on every pixel of every shell to produce a fall-off as demonstrated on the next page in figure 3.7. This will then be completed for each fur type in turn to produce a full transitional render.

Another problem to handle is the consideration that the separate passes for each hair type may generate different values for a single visible shell pixel. This problem is unavoidable with this multi-pass method. An initial thought was to take the average of the generated pixels as an approximation of sub-pixel geometry, but this proved un-necessary and costly. Therefore the algorithm will simply overwrite any existing pixel with a newly generated one, provided it passes a less-than-or-equal-to Z-buffer depth test (GL_LEQUAL).

A final point to note is that transitions to bald areas can be defined with this method by simply failing to define a fur type for the adjoining polygons.

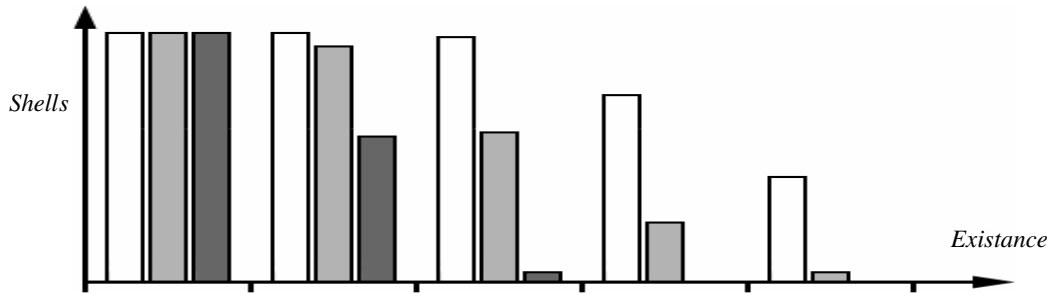


Figure 3.7 – An example of the affect of existence and dominance maps over a fading fur type.

3.3.3 Texture Generation

An adaptation of the particles approach by Lengyel et al [7] will be used to generate textures for the project. The technique for modelling the fur with particles will remain unchanged, but the conversion to texture maps needs reviewing so as to produce the required dominance and normal maps.

The dominance values will be created at the same time as the hairs by a random number generator. All values for dominance will be between 1 and 0.5 to ensure that no hairs fade out too quickly. The normal maps will be generated by comparing the hair position on the current shell to that below it and obtaining the x/y shift. The z shift will be a defined constant as the spacing between the shells will always be equal. The generated vector must then be normalised before being stored in the texture.

The particle paths will be defined by 4 sine curves, two for each variable axis, and a length value. Several other techniques have been tried, such as random directions weighted towards a central point, but the multiple sine waves produced the best results. Upper caps of 7 radians are set for the phase of these sine waves, with the magnitude being limited to 4 times the radius of the hair. In the case of a particle leaving the bounding box, it will be displaced by the box dimensions to re-enter correctly on the opposing side, ensuring correct tessellation of the texture.

The final consideration for the particle properties is the thinning of the cross-sections towards the tip of the hairs. The particle size reduces as it progresses away from the skin using a derivative of the inverse function, obtained through experimentation. This function, and the resulting radius graph, is shown below.

$$R = 1 - (1 / (5.4 - 4h)) \quad (\text{Eq. 3.6})$$

Where h is the distance up the hair [0-1] between the base and the tip of that hair (not to the top of the bounding volume)

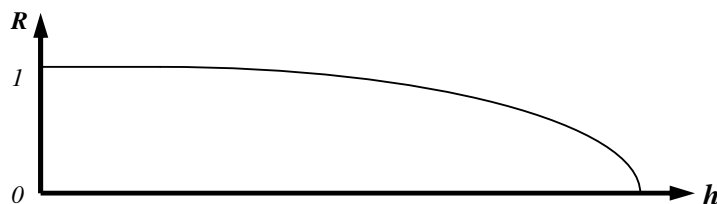


Figure 3.8 - The hair width reduction graph

To avoid generating fur patches where hairs pass through each other (as is very likely with a randomised method such as this), each particle's path will not be added to the type set maps until it has successfully completed its lifecycle without causing collisions with an existing hair. If a collision is detected a new randomisation seed will be generated and the particle rerun. If no path is found within 20 tries, the system should abandon the hair and continue to the next one to avoid the

system from hanging when no remaining paths are possible. When a successful path has been completed the randomisation seed will be restored to that at which the particle began and it will be run 'live', writing its position, dominance and direction to the type sets every time a shell plane is crossed.

The fin textures will be generated by sampling a small strip across this bounding box. Background reading suggests this should be approximated a quarter of the z range taken from the centre, but personal work found that only a tenth of the range was required. Any successful particle that starts in this range will have its full path mapped on to the 2D fin texture. The normal will also be written to the fin normal map with the same values as the shell map. This is because, even though the texture will be arranged differently to the shells, the normal will still point outwards from the model, not the polygon, so the normal map values taken from the shells will remain correct.

3.3.4 Texture Application

As stated earlier in the colouration subsection (3.2.3), the model will host 2 different texture domains; one for the repeating fur textures, and another for the colour maps. It was also specified that the fur texture should repeat continuously where possible, but with the highest priority being given to regular sized repetitions.

The best approach for this was considered to be intermediate surface mapping. Having considered the options available in this technique, and through a little experimentation, it was found that a combination of Cube S-Mapping, and Intermediate Surface O-Mapping produced the best results.

Each side of the cube map will be given texture co-ordinated from (0,0) to (x,x) where x is a multiple factor defined before the texture co-ordinates are allocated. This is to allow the fur to be scaled accurately to the model dependant on the scale of the model.

The colour maps will be applied to the model as indicated in the incoming 3D model. This means that an artist can map a colour texture to the model using and method they see fit, as they would in the case of a normal model, and the system could apply the fur automatically with minimal user intervention.

3.4 The Dynamics Problem

It has already been identified that hair dynamics will be modelled using the Wind Vectors model [18]. Several downfalls have also been identified with this method that must be considered in detail.

3.4.1 Defining the problem

The wind vectors approach produces good effects given a single vector of force. However, for use with fur a number of additional forces must be considered. These are:

- Gravity – A constant force along the vector [0,-1,0].
- Wind – The force of any weather affects on the fur.
- Movement – The force of air resistance as the furry object moves through world space.
- Momentum – The force of fur momentum as the model moves.
- Hair Resistance – The force of the hair resisting any movement.

These individual forces must be calculated and combined to produce a single 'Bend' force that will be used to perturb the fur shells.

To make the technique more flexible, the method detailed below is entirely self-contained so that the same results will be generated regardless of the underlying animation method for the base model. That is to say, the method represents a medium-level animation approach.

3.4.2 Deriving the wind vectors

To calculate the wind vector for each vertex we must first calculate the individual forces that are acting on this point. Each force will be represented as a 3-dimensional vector with the same direction and magnitude of the force. The physics equations in this section are taken from [20].

Gravity vector

This is the downwards force on each hair, the magnitude of which is calculated using the Newtonian weight equation (Eq. 3.3), resulting in an overall force vector of $[0, -W, 0]$. This must be calculated for every vertex as the fur mass will not be constant across the whole model in most cases.

$$W = mg \quad (\text{Eq. 3.7})$$

where m is the mass of the hair (defined at each vertex),
and g is a global constant representing the force of gravity.

External wind vector

This environmental condition can take any direction or magnitude and will be controlled by an environment routine in the system. This is assumed to originate from an infinitely distant point, resulting in an equal force upon every vertex on the model. Therefore, to improve efficiency this should only be calculated once every time the environmental condition changes.

Movement wind vector

This is defined by the direction and speed of the vertex movement in world space. Using the distance over time equation for defining speed we can determine the movement of the vertex. This is then inverted to get the opposing wind resistance vector.

$$M_v = -x(V_b - V_a) / t \quad (\text{Eq. 3.8})$$

Where V_b is the current vertex position in world space,
 V_a is the world-space position of the same vector the previous time it was rendered,
 t is the time elapsed since the previous render, and x is a positive constant.

This must be calculated for every vertex on the model, as each individual vertex will be moving in a different direction in most cases.

Momentum vector

This is the most expensive force to calculate, but has the greatest affect on the realism of the fur motion. According to Newtons second law of motion, the momentum vector is calculated by multiplying the object mass by its acceleration:

$$F = ma \quad (\text{Eq. 3.9})$$

Where m is the mass of an object and a is the acceleration of that object.

The mass of the hair will be generated with the type set textures and be stored at each vertex, but we do not yet have its acceleration. This can be calculated using the average acceleration rule, defined as:

$$A = \Delta v / \Delta t \quad (\text{Eq. 3.10})$$

Where Δv is the difference in velocity and Δt is the difference in time

The unknown is now the difference in velocity, which we can easily calculate by subtracting the current Movement vector from that generated in the previous frame. Combining these theories then gives us the final momentum-vector equation of:

$$L_v = m (M_v - M_{v-1}) / t \quad (\text{Eq. 3.11})$$

Where m is the mass, M_v and M_{v-1} are the current and previous movement vectors, and t is the time elapsed since the previous render.

Combining the forces

Once the individual forces have been calculated, they must be combined to produce a single resultant force for each vertex. This is done using the Parallelogram rule, which states that the resultant force between two vectors is equal to the sum of the vectors. Knowing this, the following simple equation can be used to combine the forces, giving us the overall force vector acting upon the hair.

$$F_v = G_v + a (E_v) + b (M_v) + c (L_v) \quad (\text{Eq. 3.12})$$

Where a , b and c are constants, and G_v , E_v , M_v and L_v are the Gravitational, Environmental, Motion and Linear Momentum vectors respectively.

The final stage in deriving the final wind vector for perturbing the shells is to introduce the hair resistance factor. This is basically a delay factor so that the acting forces take effect over time rather than being instantaneous. The simplest method to do this is to simply combine the current acting force with the final force value from the previous frame as follows

$$FF_v = a (F_v) + b (F_{v-1}) \quad (\text{Eq. 3.13})$$

Where a and b are constants in range $[0-1]$ such that $a + b = 1$

3.4.3 Perturbing the Geometry

Unfortunately, the graphics card I am developing this system for (Radeon 9600XT) does not support recursion in calculating the vertex position unless the number of iterations can be identified prior to the calculation occurring (in which case the recursion can be in-lined). This results in the chosen method for perturbing the hair using a summation of trigonometric equations unfeasible with the hardware available.

A solution to this problem is to approximate the perfect curve with a directly referable algebraic equation. The first step is to approximate the end point of the fur in relation to a given force. By plotting a graph of the results given with the summation method defined in the literature review, the following equations were developed for this purpose:

$$W_x = 0.861986f - 0.176676f^2 \quad (\text{Eq. 3.14})$$

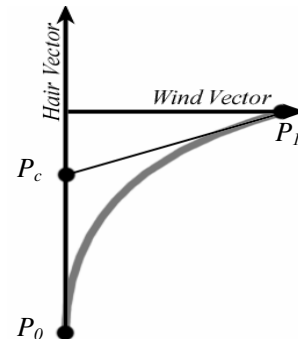
$$H_x = 1 - 0.04743f - 0.36726f^2 \quad (\text{Eq. 3.15})$$

Where f is the force magnitude

Now that an end point is obtained, and the start point is known inherently, it is possible to approximate any point along the hair using a Quadratic blending function (Eq 3.16). Experimentation with this method has found that the ideal position for P_c is half way along the hair vector, as illustrated in figure 3.9.

The comparison between this method, and the traditional method is illustrated in figure 3.10 below. There is some minor difference in the spacing, but overall the result is sufficiently accurate for use.

Figure 3.9- Approximating the bend factor



$$Q(u) = P_0(1-u)^2 + P_1u^2 + P_c2u(1-u) \quad (\text{Eq. 3.16})$$

Where u is the distance $[0-1]$ along the curve



Figure 3.10 – Comparison between the approximated(left) and original(right) bend functions..

The only remaining problem is that the wind vectors technique was developed using a shells only method. In order to fully implement this technique into the project, considerations on how to render the bending of fins must be made. The obvious solution to this is to split each fin into horizontal segments with one sub-fin between each shell layer (illustrated). Fortunately, all the vertices required to do this will have already been calculated in rendering the shells. By buffering these vertex positions on the graphics card it should be possible to render all the sub fins required at minimal extra cost.

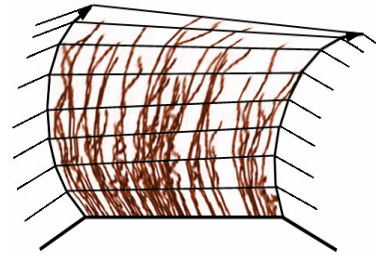


Figure 3.11 – Illustration of sub-fins

3.5 Testing Plan

The development of the system is to be carried out in three key stages, each a superset of the last. In chronological order, these are:

- *The Basic Model stage* – Implementing the base system upon which the more advanced techniques will be built upon.
- *The Hair Type stage* – Concerned with mapping several hair types onto a single model
- *The Dynamics stage* – The final step is to incorporate the dynamics system.

As each stage builds upon the code of the previous one, testing will be an ongoing procedure with thorough testing at the end of each stage to ensure stability. When all three stages have been completed, a final stage of testing will be performed to ensure all aspects of the system work together correctly, prior to a demonstration being coded for the end of project Poster Session.

Any auxiliary programs will be thoroughly tested independently of the fur renderer. Due to the nature of the project, it is difficult to define any functional testing in the traditional sense, as most output is graphical rather than numerical. The testing will consist mainly of testing the robustness of methods on a variety of models under a time-critical viewpoint.

3.6 Evaluation Plan

To determine the overall success at the end of the project, a thorough evaluation is to be conducted. This will consist of the following four sections.

Aims Achievement Evaluation

Firstly, a personal evaluation of the project's ability to achieve each of the objectives set out in the General Fur objectives at the beginning of this chapter. The dynamics will also be considered in this section, evaluating its accuracy and its affect on the realism of the system.

Type Set Evaluation

In this section, the ability to generate different fur types will be tested. This will consist of obtaining a number of close-up images of various fur types and trying to emulate them within the system. This will highlight both the quality of fur representation, and indicate any fur types which cannot be simulated.

Overall Evaluation

This will be an overall review of the method. A full model will be developed in an attempt to accurately represent a furry creature.

Technique Review & Comparison

Finally, once the technique has been evaluated as a singular entity, a comparison with existing real-time systems currently on the market will be considered. These systems will be

- The ATi demo for the shells method, which will compare the base technique with the extensions that have been imposed during this project.
- Black & White 2 for a comparison with an alternative method. This game implements fur via the polygon strips approach described in the Literature Review section of this paper.

3.7 Tools

This section considers the tools used to develop the system.

3.7.1 Programming Language and API's

The programming language used in the project is C++ due to the speed of computation available compared to other OO-Languages (such as Java). It is also the standard language for the field of 3D Graphics and Video Games providing a good basis for later expansion.

To make use of 3D graphics card features, a 3D API was adopted. The choices for this were SGI's OpenGL or Microsoft's DirectX. For this project, OpenGL was adopted as it has slightly more support for vendor-specific card extensions, and its functional structure makes it somewhat easier to use. Unfortunately, the base functionality of OpenGL is less advanced than that of DirectX, but with the large range of extensions available in the OpenGL community the differences between the two are minimal.

The project makes use of the GLFW and GLee expansions for OpenGL. The first provides a framework to simplify interfacing with the operating system. This makes functionality of windowing routines and I/O considerably easier, and has multi-platform support, allowing the project to be easily ported at a later date if necessary. The second, GLee, is an extension wrapper that allows all OpenGL expansions to be included with just a few lines of code.

3.7.2 Shader Language

As this project is heavily focussed on real-time rendering, the use of a shader language is deemed necessary. This allows the graphics card functions to be reprogrammed, allowing more of the rendering calculations to be performed on the graphics card to free up the CPU for more application specific tasks.

All third and fourth generation graphic cards (GeForce3 onwards) have two programmable pipeline stages. The first is concerned with the position, colour and lighting properties of vertices, and is called the Vertex Shader. The second, the Fragment Shader, is responsible for the final output colour and Z-buffer depth of each fragment (a fragment being a potential on-screen pixel). Many of the techniques presented in this chapter could therefore be calculated on the graphics card itself, freeing up the CPU to concentrate on other operations.

Traditionally, these shaders would be programmed in low-level assembly code. Whilst very versatile, this would be a long and complicated process. In 2002, NVIDIA released Cg, a high-level language similar to C for shader programming. This is the approach taken within this project.

As well as being simpler to code, Cg's high-level stance has many advantages, such as support for arrays, structures, looping, conditional branching and function calls. All common data structures and mathematical functions are also included, with 3D specific functions, such as calculating the reflection vector for specular lighting, possible with one hugely efficient instruction (`reflect`). [21] Cg's structure also allows for multi-API and multi-platform support by its range of parsers, compilers, and the CgFX file structure. Because this program has been developed primarily for fourth generation ATi cards, the CgFX structure has not been used. However, inclusion of this as a later extension could easily be considered.

Chapter 4: FurMak

The final system developed during this project consists of two programs. The first, FurMak, is a pre-processor that generates fur textures, and converts .obj files into a format used for displaying the fur. The second program, FurGLE, is a combined render/editor for the furry objects. The following two chapters consider the design and implementation of these two programs, and discuss the findings made in the development process.

In this chapter, we concentrate on the pre-processor. Section 4.1 discusses the layout of the program. Sections 4.2 and 4.3 then discuss the generation and output of data regarding texture maps and co-ordinates respectively.

4.1 FurMak

FurMak itself has two main purposes. Firstly, it must generate the type set textures for both the fins and shells of the model. As earlier specified, multiple fur types should be present on any given model. A decision has been made through careful consideration of animals that a total of 4 fur types will suffice, one of which must be baldness. This can be quantified by considering a rabbit or hare. The three custom fur types would be a fairly dense coat for the body, an equally dense, but thinner and curlier hair for the bushy tail, and a less prominent hair type for the inner ears. A bald option must be provided because every single animal has areas without any fur, even if it's only the eyeballs.

As indicated in the Analysis stage, each type set will contain:

- 16 Dominance/Variance maps for the Shells
- 16 Normal maps for the Shells
- 1 Dominance/Variance map for the Fins
- 1 Normal map for the Fins
- A colour map for altering the fur colour

The second purpose of FurMak is to convert a given 3D model into a format that contains information regarding the fur volume as well as the basic mesh. As part of this process, a second set of texture co-ordinates must be generated, concerned with the application of the fur set maps. Although direction and length values cannot be generated automatically, default values should be created to ensure the resulting file format can be used to save the models once these values have been set.

The overall program flow is as indicated in Figure 4.1

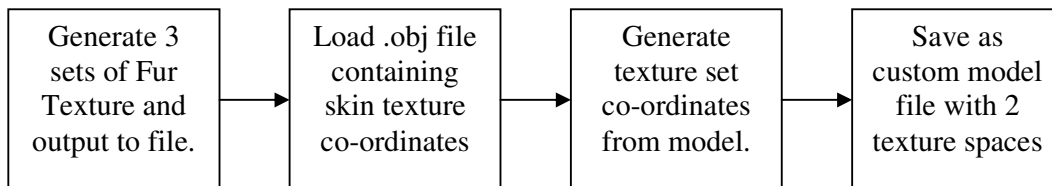


Figure 4.1 – FurMak program flow

4.2 Texture Generation

For this implementation, a texture size of 256x256 for the shells, and 256x64 for fins has been adopted. This is considerably larger than you would expect to see in a practical implementation of this method, but for an experimental program we have the available memory to be a little more elaborate.

As earlier specified, the fur will be constructed a hair at a time, before the data is converted into data maps. This is done as follows:

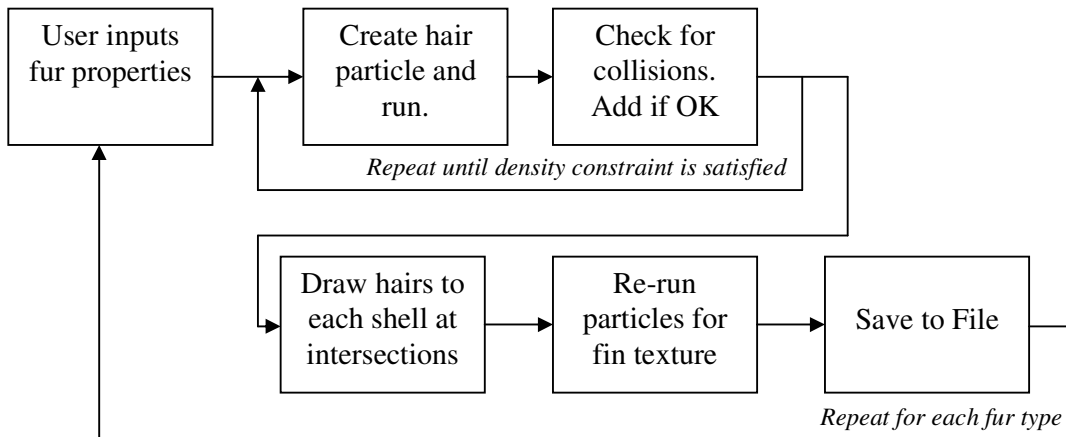
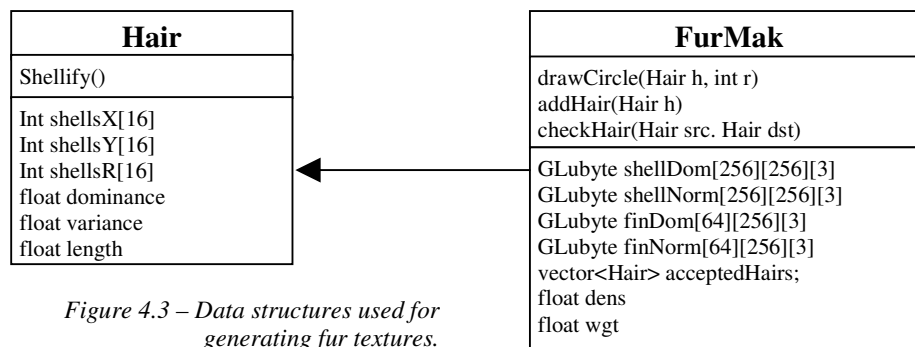


Figure 4.2 – Fur texture construction

4.2.1 Data Structures

The internal representation for the fur generation is as shown below (Figure 4.3) Note that not all variables and methods are listed here, only those that are necessary to understand how the program works.



It was decided that the output would be of a custom type collecting all the fur type textures into a single file, as saving each individual texture as separate image files would become highly confusing when managing file resources. The .fts (FurGLE Type Set) file-type was generated to serve this purpose and is organised as shown on the following page...

- Innermost Dominance map, stored as an array of chars read in row-major order
 - [variance][dominance][alpha]

...Repeated for each shell layer...
- Outermost Dominance map
- Fin Dominance map, stored as an array of chars read in row-major order
 - [variance][dominance][alpha]
- Single float value representing the overall fur density (for use in the shadowing algorithm)
- Single float value representing the weight of the fur type (for use in the dynamics algorithm)

As this structure is constant for any successful generation, each component in the file does not need to be separated, so the data is simply serialized into the file.

Normal maps were also to be included in this file, each occurring after the corresponding dominance map. However, difficulties were encountered both in the transferring and rendering of the normal maps, and they were excluded as the project neared its end. Most of the code remains and is well commented for the possibility of extension.

4.2.2 Generating Hair Data

For each fur type to be created, the user supplies three property inputs – Density, Thickness and Curliness. The thickness and density values are first used to calculate the number of hairs to be rendered (thickness is required as a patch of dense thick hair will contain less hairs than a patch of dense fine hair). This is achieved by using the following equation.

$$\text{hairCount} = d * mh * (1 - (t * 0.7)) \quad (\text{Eq. 4.1})$$

where d is the fur density (0-1), t is the fur thickness (0-1) and mh is the maximum possible number of hairs for any given input (defaulted at 1500)

Hairs are then repeatedly generated up to this quantity, each taking its defining values as indicated in the following table. Note that the two magnitude pairs are calculated differently. The purpose of this is to increase the random curvature of fur, producing more realistic results.

Length:	random number in range [0.0 - 0.5]
Breadth:	supplied thickness value [0-1] * maximum radius (6 texels)
Dominance:	increases with each hair from 0.2 for first, to 1.0 for the final hair generated
Variance:	random number in range [0.0 – 0.8]
Position:	random x/y co-ordinates in range [0.0 – 1.0]
Phase1:	random x/y values in range [4 – 7.141 radians]
Phase2:	random x/y values in range [2 – 5.141 radians]
Magnitude 1:	(+/-) random x/y values in range[0.0 – 1.0] * supplied curliness value * radius
Magnitude 2:	(+/-) supplied curliness value * radius – random x/y values in range[0.0 – 1.0]

Table 4.1 – Data generated for each hair

Once all the input data for the hair is specified, the shellify() method is called to factorise this raw data into an arrays containing the x /y co-ordinates and radius for that hair at each shell. The code for this is shown in figure 4.4. The only problem to note here is that to maintain tessellation of the texture, any hair protruding from the edge of the bounding volume must re-enter from the opposite side. This is easily done by checking the co-ordinate against the bounding box and offsetting the location accordingly.

```

void Hair::shellify() {
    for(int l=0; l < LAYERS; l++) {
        double sl = (double)l / (double)LAYERS;
        shellsX[l] = x + (int)((xm * sin(xp * sl)) + (xm2 * sin(xp2 * sl)));
        shellsY[l] = y + (int)((ym * sin(yp * sl)) + (ym2 * sin(yp2 * sl)));
        float factor = ((float)l / length) * (float)l;
        float rad;
        if (factor > LAYERS) {
            rad = 0.0f;
        } else {
            rad = 1.0f - (1 / (5.4f - (4 * (factor / (float)LAYERS))));
        }
        shellsR[l] = rad * r;
    }
}

```

Figure 4.4 – The shellify() method

When a hair has been ‘shellified’, it is checked against each of the accepted hairs in turn for any collisions. This is simply performed by ensuring that the Euclidian distance between any two hairs is greater than the sum of the hair radii. Again, the tessellation of the texture can cause a problem here, allowing hairs to be accepted incorrectly as illustrated in Figure 4.5

To resolve this, the bounding area is split into 4 quadrants (indicated in the diagram). If the two comparing hairs are on opposite sides of a separating line, the position of the hair is temporarily offset by the box dimension in the according direction. E.g. in the case of Figure 4.5, H1 and H2 are on opposing sides of the Y separation, so H2 will be positioned at (H2.x – box dimension, H2.y) for the duration of the collision test.

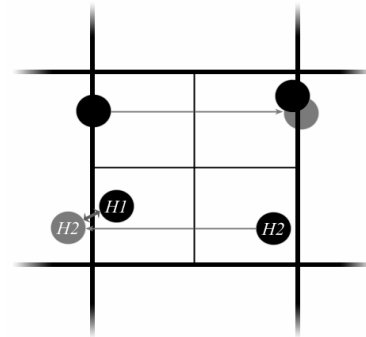


Figure 4.5 – Correct collision detection

The check is made for both the original position and the offset position. If a hair encounters no collisions it is added to the `acceptedHairs` vector in the `FurMak` class. In the event of a failure the hair is regenerated up to a maximum of 100 times before it is thrown away, allowing the program to continue.

4.2.3 Converting Hair Data into Texture Maps

Once all the hairs have been specified, and checked, the data is converted into byte arrays for OpenGL to use as a texture format. We shall consider how this is done for the two different texture categories.

Shells

This is done by ‘drawing’ filling circles for each hair on each shell in turn. The `shellify()` method has already defined the positions and radius of each circle, and the dominance and alpha values are stored in the hair structures, so this process is manifestly evident.

Once variance and dominance were implemented it became clear that there was a problem with mip-mapping the shell textures. This was due to the large amounts of empty data surrounding each hair resulting in lower levels of the texture containing considerably smaller values than intended. To resolve this problem, each texel takes the variance and dominance values of the closest hair via a naïve 1-NN search algorithm. Alpha values remain unaltered to preserve the fur pattern.

Fins

For the fins only the hairs that start within the central 6% of the bounding box are used. To draw these hairs, the `shellify` procedure is performed to a higher frequency, referencing the location and

radius of each hair over 64 planes in the bounding box. These are then used for constructing each line of the fin texture. The code for this is illustrated below.

```

for(unsigned int currentHair = 0; currentHair < acceptedHairs.size(); currentHair++) {
    if(acceptedHairs[currentHair].shellsX[0] > 120 &&
        acceptedHairs[currentHair].shellsX[0] < 135) {
        for (int cyrp = 0; cyrp < FUR_HEIGHT; cyrp++) {
            double sl = (double)cyrp/(double)FUR_HEIGHT;
            int cxrp = acceptedHairs[currentHair].shellsY[0]
                + (int)((acceptedHairs[currentHair].ym
                    * sin(acceptedHairs[currentHair].yp * sl))
                    + (acceptedHairs[currentHair].ym2
                    * sin(acceptedHairs[currentHair].yp2 * sl)));
            float factor = (1.0f / acceptedHairs[currentHair].length) * (float)cyrp;
            float rad;
            if (factor > FUR_HEIGHT) {
                rad = 0.0f;
            } else {
                rad = (1.0f - (1 / (5.4f - (4 * (factor / (float)FUR_HEIGHT))))
                    * acceptedHairs[currentHair].shellsR[0]);
            }
            for(int xdisp = (int)-rad; xdisp < (int)rad; xdisp++) {
                int newx = cxrp + xdisp;
                if (newx > DIMENSION) newx -= DIMENSION;
                if (newx < 0) newx += DIMENSION;
                finDomvar[cyrp][newx][0] = acceptedHairs[currentHair].shellsY[0]
                    - (cxrp + xdisp);
                finDomvar[cyrp][newx][1] = (int)(acceptedHairs[currentHair].dominance
                    * 255.0f);
                finDomvar[cyrp][newx][2] = 255;
            }
        }
    }
}

```

Figure 4.6 – Fin texture creation

4.2.4 Testing

The texture generator was tested with a range of inputs to ensure correct operation. The results are shown below. Note that due to the naïve approach taken by the ‘filling’ algorithm, the full texturing method takes considerably longer. The time taken is extensive, but the results are accurate and time is not a critical issue for an offline process. However, if the generator was to be further extended this would seriously need optimising.

The only fault found with the results is that thickness values below 0.3 result in an absence of fur as the radius calculated for each hair is zero. However, the fur generated is correct in all other cases, so this is not a major problem provided any developer using the system is aware of this limitation.

Density	Thickness	Curliness	Time taken	Results
0	0	0	0.659 seconds	No texture generated
0.3	0.8	0.5	0.806 seconds	Suitable texture generated
0.8	0.2	0.2	1.987 seconds	Suitable texture generated
1	1	1	13 seconds	Suitable texture generated

Table 4.2 – Fast texturing test results

Density	Thickness	Curliness	Time taken	Results
0	0	0	0.67 seconds	No texture generated
0.3	0.8	0.5	1 min, 7 sec	Suitable texture generated
0.8	0.2	0.2	6 min, 18 sec	Suitable texture generated
1	1	1	1 min, 31 sec	Suitable texture generated

Table 4.3 – Complete texturing test results

4.3 Model Conversion & Texture Co-Ordinates

The second function of FurMak is the generation of a secondary texture set for the application of the fur textures. This is done in the following four stages. Loading and saving objects is considered generic, so only the two central stages are discussed here.

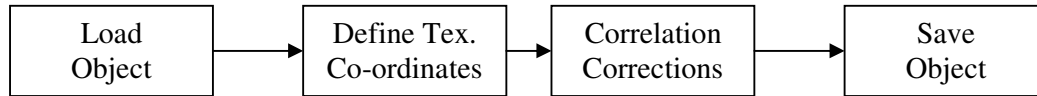


Figure 4.7 – The four stages of model conversion

4.3.1 Data Structures

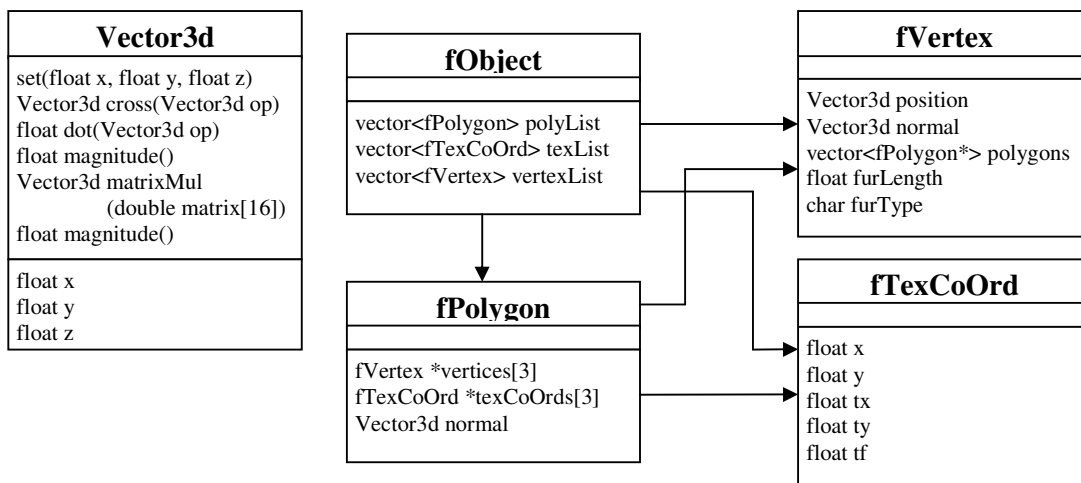


Figure 4.8 – FurMak model data structure

The data structure for the model is as shown above. The `fTexCoOrd` has two co-ordinates stored with it. `x/y` are the colour co-ordinates, and `tx/ty` are the fur co-ordinates. `tf` is a marker used to identify which face of the cube map that particular co-ordinate was obtained from. This is used for ensuring correlation across the texture domain.

The incoming object is a traditional `.obj` file with vertex, face and texture co-ordinate entries. The normals are computed in the file loader so are not required in the `.obj` file itself. The object represented must be a triangular mesh, and all referencing in the file must be direct, not relative, for the file to be accepted. After conversion a new file type is needed to store the extra derivative. Therefore the `.fo` (FurGLE Obj) file format is introduced with the following format.

Vertices are listed as follows. All values are floats except `fur_type`, which is an integer.

```
v x_position y_position z_position x_normal y_normal z_normal length fur_type
```

This is then followed by the list of texture co-ordinates in the following format. Note that both texture spaces are stored together to make loading and saving much easier.

```
t colour_x colour_y fur_x fur_y
```

Finally, the polygons themselves are stored exactly as with `.obj` files, except with the preface ‘p’ - This is to help instantly distinguish between `.obj` and `.fo` format when files are being manually inspected. The vertex and texture references are again direct, not relative.

```
p v1/t1 v2/t2 v3/t3
```

4.3.2 Vertex mapping System Structure

The only user input to the vertex mapper is a pre-formatted .obj file, and a floating-point number which represents the scale factor, defining how often the fur texture repeats along the model. This recursion factor is dependant on the maximum dimensions of the model, which are obtained by iterating through all the vertices and updating the `minDim` and `maxDim` global variables.

As described earlier, cube mapping is used to apply the texture co-ordinates. The first stage in achieving this is to find the cube face against which a given vertex will be referenced. This is attained by using a ray-plane intersection test with each of the cube faces; the ray being cast from the vertex position along the vertex normal.

Ray-plane intersection is detected with equation 4.2 [22]. If the ray does intersect with the plane, a number, representing the distance from where the ray starts (in this case the vertex) and the plane, is returned. The cube surface most relative to a given vertex will be that which produces the smallest result from this calculation.

$$t = -(P_0 \cdot N + d) / (V \cdot N) \quad (\text{Eq. 4.2})$$

$$P = P_0 + tV \quad (\text{Eq. 4.3})$$

Where P_0 is the origin of the ray, N is the normal to the plane, d is the distance to the plane, and V is the direction vector of the ray

Once the cube face is selected, the texture co-ordinate can be obtained by finding the position on the face at which the intersection occurs. This is easily done by multiplying the result from the previous solution by the vertex normal and adding this to the vertex position (Eq 4.3). This cube co-ordinate is then converted to the model texture space by Eq 4.4. The code for the vertex mapping is shown below in figure 4.9

$$T = ((T_0 - \text{minDim}) / (\text{maxDim} - \text{minDim})) * \text{sfact} \quad (\text{Eq. 4.4})$$

Where T_0 is the cube map location derived from Eq 4.?, and sfact is the scalar factor defined by the user.

```
vector<fPolygon>::iterator poly = furryObj->polyList.begin();
while(poly != furryObj->polyList.end()) {
    for(int vert = 0; vert <3; vert++) {
        bool found = false;
        Vector3d bestMatch(10000.0f, 10000.0f, 10000.0f);
        int bestFace;

        if(poly->vertices[vert]->normal.x > 0.0f) {
            Vector3d ip = checkCubeIntersect(poly->vertices[vert]->position,
                poly->vertices[vert]->normal, Vector3d(-1.0f, 0.0f, 0.0f));
            if(ip.magnitude() < bestMatch.magnitude()) {
                bestMatch = ip; bestFace = 0;
            }
        } else if(poly->vertices[vert]->normal.x < 0.0f) {
            Vector3d ip = checkCubeIntersect(poly->vertices[vert]->position,
                poly->vertices[vert]->normal, Vector3d(1.0f, 0.0f, 0.0f));
            if(ip.magnitude() < bestMatch.magnitude()) {
                bestMatch = ip; bestFace = 1;
            }
        }
        // Process repeated for y and z axis planes
        float xpos;
        float ypos;
        int SurfNormTex(bestFace, &(*poly), vert, xpos, ypos);
        poly->texCoOrds[vert]->tx = ((xpos - minDim) / (maxDim - minDim)) * sfact;
        poly->texCoOrds[vert]->ty = ((ypos - minDim) / (maxDim - minDim)) * sfact;
        poly->texCoOrds[vert]->tf = bestFace;
    }
}
```

Figure 4.9 – Code for generating texture maps

Note: `checkCubeIntersect` performs equations 4.2 and 4.3 to return the intersection point. `intSurfNormTex` performs Eq. 4.4 returning the resulting co-ordinate in `xpos/ypos` reference parameters.

4.3.3 Ensuring correlation

The only short coming of this method is the anomalies which occur when the referenced texture coordinates on a single polygon come from different faces of the bounding box. This results in incorrect texturing as shown in Figure 4.11.

The traditional method when performing cube mapping is to identify where on the polygon the cube face changes occur, and split the model polygon into several sub-polygons so that each only references a single face of the cube map. In this instance, this process is unnecessary as the cube map textures do not match up (as they would on an environment map for example) so we can get away with simply mapping the existing polygon to a single cube map face without worrying about the joins.

To select which face the polygon maps to, the dot product between the polygon normal and each of the cube face normals is calculated. The largest result from the 6 faces indicates the face to be used for mapping that polygon. The mapping then occurs as in the previous subsection. The improved mapping is shown in Figure 4.12 for comparison.

There is however one other problem that is caused here as more `fTexCoOrd`'s are being added to the system. If these were simply added to the vector there is a high probability the memory location of the vector will change, destroying the pointer references that the model relies on. To work round this, a secondary `fTexCoOrd` array is included to store these new entries, to which the polygons point instead of the vector. As this will be a static array, there is no danger of the addresses changing during a single execution. The correction code is shown below:

```

if ((poly->texCoOrds[0]->tf == poly->texCoOrds[1]->tf) &&
    (poly->texCoOrds[1]->tf == poly->texCoOrds[2]->tf)) {
    //All ok
} else {
    Vector3d bestMatch(10000.0f, 10000.0f, 10000.0f);
    int bestFace;
    // Best face searching removed for conciseness

    float xpos, ypos;

    fTexCoOrd newTexCoOrd = fTexCoOrd();
    newTexCoOrd.tf = bestFace;

    for(int vert = 0; vert < 3; vert++) {
        if(poly->texCoOrds[vert]->tf != bestFace) {
            intSurfNormTex(bestFace, &(*poly), vert, xpos, ypos);
            newTexCoOrd.x = poly->texCoOrds[vert]->x;
            newTexCoOrd.y = poly->texCoOrds[vert]->y;
            newTexCoOrd.tx = ((xpos - minDim) / (maxDim - minDim)) * sfct;
            newTexCoOrd.ty = ((ypos - minDim) / (maxDim - minDim)) * sfact;
            newTexCoOrds[newTexCounter] = newTexCoOrd;
            poly->texCoOrds[vert] = &newTexCoOrds[newTexCounter];
            newTexCounter++;
        }
    }
}

```

Figure 4.10 – Texture co-ordinate correction code

When saving the `.fo` file, these are then written after the vector, with the first element of the secondary array being the `vector.size()+1`'th texture entry. The references in the polygon data is also saved in this manner. Therefore, once the file is saved, there is no indication a second array was used, and the resulting file can be loaded into a single vector next time it is opened.

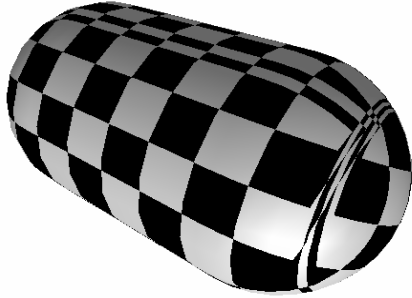


Figure 4.11 – Textured capsule showing cube-face anomalies

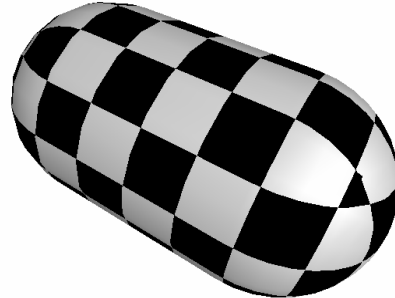


Figure 4.12 – Corrected texture allocation

4.3.4 Testing

To ensure correctness and stability of the method, the texture allocation was tested using a simple 2-dimensional checkerboard pattern on a variety of different models. An example of this is the textured ‘capsule’ as shown above in figure 4.12. The table below shows the accuracy, and computation time of four other models.

<i>Model Name</i>	<i>Polygons</i>	<i>Time Taken</i>	<i>Texture Application</i>
Sphere.obj	180	0.686ms	Perfect
Torus.obj	432	1.652ms	Perfect
Knot.obj	2,000	7.63ms	Some minor misallocations, but barely noticeable.
Rabbit.obj	1,506	5.827ms	Again, some extremely minor misallocations

Table 4.4 – Testing texture application on various models

Chapter 5: FurGLE

Once the necessary files have been created by FurMak, it is FurGLE's role to display the furry model, and allow the user to vary details of the fur, such as type, length and direction. This chapter looks at how this is done. Firstly, Section 5.1 considers the underlying structure of the program, giving an insight into the relationship between the different components. Section 5.2 then considers the main program loop, and how the scene is rendered. The GPU-side functionality is then described in Section 5.3. Finally, Section 5.4 describes the approach taken for user interaction with the program.

5.1 System Design

The general structure of the program is as shown in Figure 5.1. This is a simple overview of the link between the different components that should provide enough information for the system to be understood.

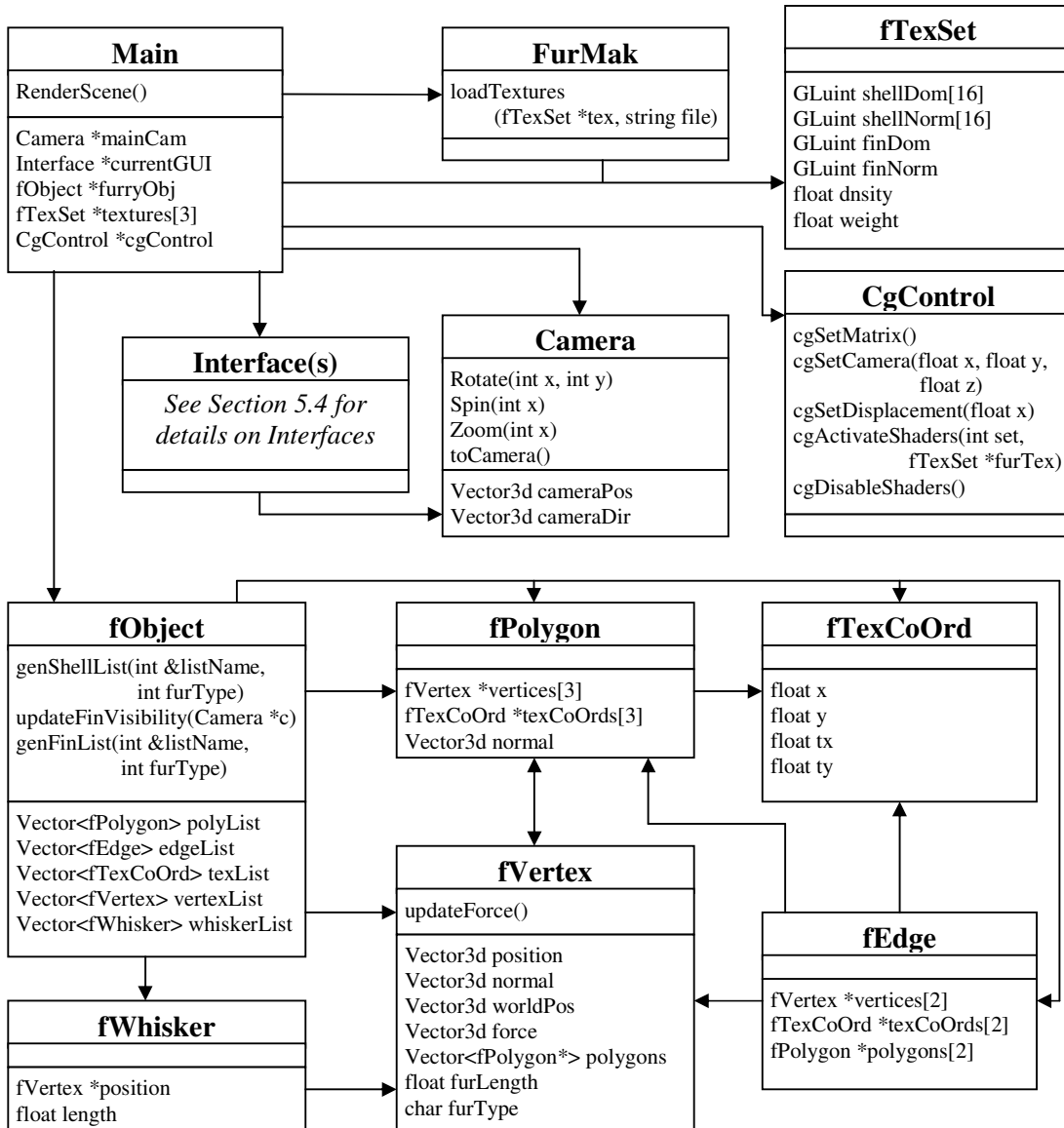


Figure 5.1 – The structure of FurGLE

5.2 Fur Rendering

When FurGLe is started, the model file and all necessary textures are loaded, a window is created, OpenGL and Cg contexts are created, and the Control interface is initiated. These are all trivial processes and are not described in too much detail in this chapter. Once these processes are complete, control is passed to the main loop. The stages of the main loop are shown in Figure 5.2 and the code is shown in Figure 5.3.

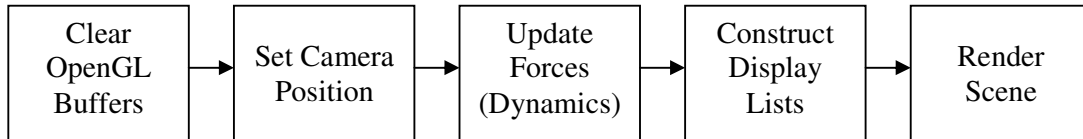


Figure 5.2 – The main program loop

Setting the camera position is simply done by calling the `toCamera()` method of the relevant camera object. Note that the updated camera position must also be sent to the Cg controller so the shaders can be informed of the state change. The details of the remaining 3 stages are described in the following 3 subsections.

```

while (glfwGetWindowParam(GLFW_OPENED)) {
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    mainCam->toCamera();
    cgControl->cgSetCamera(mainCam->cameraPos.x, mainCam->cameraPos.y,
                          mainCam->cameraPos.z);

    vector<fVertex>::iterator v = furryObj->vertexList.begin();
    v->trace = true;
    while (v != furryObj->vertexList.end()) {
        if (v->furType < 4) v->updateForce();
        v++;
    }

    furryObj->updateFinVisibility(mainCam);
    for (int typeRec=0; typeRec < 3; typeRec++) {
        if (renderFins) furryObj->genFinList(finList[typeRec], typeRec+1);
        if (renderFur) furryObj->genShellList(shellList[typeRec], typeRec+1);
    }

    GLfloat light_position[] = {10.0f, 10.0f, 10.0f, 1.0f};
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    renderScene();

    currentGUI->displayGUI();
    glfwSwapBuffers();
}
  
```

Figure 5.3 – Main Program Loop code

Note that fur type 4 is bald and is therefore not rendered. This also means it does not require a wind vector to be calculated for any of its vertices.

5.2.1 Updating the forces

The wind vector for each vertex is calculated as described in the Analysis (3.4.2). The only complication involved is in the conversion between spaces. The vertex positions are in local space, which need to be converted to view space for the movement vector to be obtained. Once this is done, the resulting movement must be converted back to local space to represent the movement

relative to the vertex. Similarly, the gravity vector, which must be defined in view space, must also be converted to local space. The code for generating the wind vector for a given vertex is shown in Figure 5.4 below.

```

void fVertex::updateForce() {
    extern fTexSet *textures[3];
    GLdouble tmvMat[16];
    glGetDoublev(GL_TRANSPOSE_MODELVIEW_MATRIX, tmvMat);
    GLdouble mvMat[16];
    glGetDoublev(GL_MODELVIEW_MATRIX, mvMat);

    Vector3d gravity = Vector3d(0.0f,
        -(0.45f + (textures[furType-1]->weight/4.0f)), 0.0f).matrixMul(tmvMat);
    extern Vector3d globalWind;
    Vector3d wind = globalWind.matrixMul(tmvMat);

    Vector3d currentPos = Vector3d(position.x, position.y, position.z).matrixMul(mvMat);
    Vector3d movementTemp = ((worldPos - currentPos) * 1.5f).matrixMul(tmvMat);
    Vector3d movement = movementTemp - (normal * (movementTemp.dot(normal)));
    movement.normalise();
    movement = movement * movementTemp.magnitude();
    Vector3d momentum = (movement - force);

    worldPos = currentPos;
    Vector3d fullActiveForce = (gravity + wind + movement + momentum);
    force = (force * 0.85f) + (fullActiveForce * 0.15f);
}

```

Figure 5.4 – Wind Vector calculation code.

5.2.2 Constructing the Display Lists

Early on in the project, it was found that it was possible to create a single display list when the program starts, and continually draw it with different Cg parameters to render the fur. Unfortunately, when dynamics was introduced this single list approach no longer worked. This is because the dynamics routine changed the data being sent with the vertices every frame (namely the wind vectors) and display lists cannot be changed. Before dynamics were included, the various shells could be attained by changing the GPU program states, which does not affect the raw data, only the way in which the results are calculated. Now we must recreate the lists every time the vertex data changes,

It was intended that this rendering approach would be replaced with a considerably more efficient vertex arrays method once all the separate features were implemented. This did not happen due to time constraints, but the display list approach is still faster than immediate mode as each generated list will be run at least 16 times (once for each shell) in its lifetime.

Each fur type will have its own display list to allow for multi-pass rendering over hair transitions. The shell list for each fur type is constructed by cycling through the polygons in the model and adding those that contain at least one vertex of the current hair type. When a polygon is added, data is also included to identify the existence at that point (i.e. 1 if fur present, 0 otherwise). This process is repeated for each fur type. The texture co-ordinates from the objects `fTexCoOrd` list are used for the vertices.

The fin display lists are slightly more complicated, and require selection to be performed based on the equations introduced in Section 3.2.1. The successful fins are then sorted into reverse order so the fins towards the far clip plane are rendered first. This is required because the introduction of the alpha fins method means that some fins may be semi-transparent and therefore inherently require everything behind them, including other fins, to be rendered first.


```

void fObject::updateFinVisibility(Camera *camera) {
    silhouetteEdges.clear();
    for(unsigned int x=0; x < edgeList.size(); x++) {
        Vector3d eyeVector = ((edgeList[x].vertices[0]->position +
            (edgeList[x].vertices[0]->normal * edgeList[x].vertices[0]->furLength)) +
            (edgeList[x].vertices[1]->position + (edgeList[x].vertices[1]->normal *
            edgeList[x].vertices[1]->furLength))) / 2.0f) - camera->cameraPos;

        edgePair newEdge;
        newEdge.edge = &edgeList[x];
        newEdge.distance = eyeVector.magnitude();
        eyeVector.normalise();

        float sil = ((edgeList[x].polygons[0]->normal).dot(eyeVector) *
            (edgeList[x].polygons[1]->normal).dot(eyeVector));
        if (sil < 0.3f) silhouetteEdges.push_back(newEdge);
    }
    sort( silhouetteEdges.begin( ), silhouetteEdges.end( ), PairComp );
}

```

Figure 5.5 – Fin Selection method

For each fur type, this sorted list is then iterated to create the display list containing existence values as with the shells list creation. Each accepted fin is drawn to the list as two triangles, rather than a quad as originally intended. This is due to the introduction of combing making it highly likely that any given fin is not planar.

The texture co-ordinates for the fins are again more difficult to attain than with the shells method, as there is no pre-define texture space for edges of the model. The reason for this is that it's mathematically impossible, as illustrated in figure 5.6. There is no possible way for the texture to be drawn from $A \rightarrow C$ that will match the position attained via $A \rightarrow B \rightarrow C$ over the edges of a model.

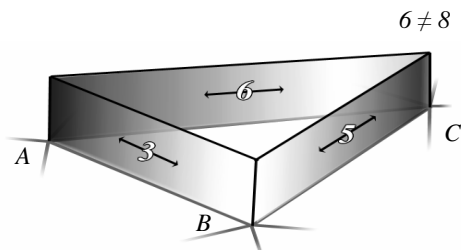


Figure 5.6 – Problems mapping the fins

The y co-ordinate for the texture fins is simply the displacement value of the current shell, from 0 at the base to 1 at the tip. The x value is more difficult to define. The approach taken here is to allocate the starting co-ordinate a value taken to be Euclidean distance of vertex texture co-ordinates from the origin. This is to ensure all fins do not start with the same point in the texture, which would result in a noticeably repetitive rendering. The x co-ordinate for the other end of the fin is defined as the starting x co-ordinate plus to length of the edge at skin level.

5.2.3 Rendering the Scene

Once the display lists have been created, the scene can be rendered. First, the base skin is rendered with a specified TGA texture. This stage is done using plain OpenGL, so lighting and texturing needs to be enabled. Face culling is also enabled to prevent back facing polygons from being drawn. This making the rendering faster as it reduces the time spent calculating areas of the scene that are not visible.

Next the fur is rendered onto the model. When rendering the fur, texturing and lighting will be disabled. This is because the shader programs will be overwriting the default OpenGL implementation for these processes. As the fur render will contain transparency, blending must be activated to allow incoming pixels to be 'mixed' with those in the buffer, based on their alpha values. If this was not activated, the outermost shell would overwrite everything, regardless of whether there were any hairs present or not.

Finally, the z-buffer must be turned off for the fur. This is required as the interpolation method draws two polygons in exactly the same location, using different textures, to achieve hair transitions. It was defined earlier that this would be handled by setting the depth test to

GL_LEQUAL, but rounding errors still produced z-buffer artefacts. Therefore the depth test is disabled entirely while the fur is being rendered.

The shells are then drawn from inner-most to outer-most by increasing the displacement value that the shader uses to set the offset position. At each ‘layer’, the fins are drawn first, followed by the shells. The reason for this is that the shells are the primary feature in the rendered image and the fins are only a secondary assistance feature which should support, not overwrite, the basic shell model. A final point to note here is that culling must be disabled when rendering the fins. This is because some back-facing fins must be rendered as they fade out. If they were to simply vanish as soon as they’d passed the silhouette, it would produce noticeable popping affects as they disappeared.

```

void renderScene() {
    cgControl->cgSetMatrix();

    glEnable(GL_LIGHTING);
    glEnable(GL_CULL_FACE);
    GLfloat col2[] = {1.0f,1.0f,1.0f};
    glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, col2);
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, skinTex);
    if(renderSkin||editMode) glCallList(skinList);
    glDisable(GL_TEXTURE_2D);
    glDisable(GL_LIGHTING);

    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glDepthMask(GL_FALSE);

    float col = 0.7f;
    GLfloat col3[] = {col,col/2,0.0f,1.0f};
    glColor4fv(col3);
    for(int type=0; type < 3; type++) {
        for(float shell=0.0f; shell < 1.0f; ) {
            shell += 1.0f / SHELL_QUANTITY;
            cgControl->cgSetDisplacement(shell);

            //RENDER FINS
            if(renderFins) {
                glDisable(GL_CULL_FACE);
                cgControl->cgActivateShaders(CG_FINS, textures[type]);
                glCallList(finList[type]);
                glEnable(GL_CULL_FACE);
            }

            //RENDER FUR
            if(renderFur) {
                cgControl->cgActivateShaders(CG_SHELLS, textures[type]);
                glCallList(shellList[type]);
            }
        }
    }
    cgControl->cgDisableShaders();
    glDisable(GL_BLEND);
    glDepthMask(GL_TRUE);
}

```

Figure 5.7 – The scene rendering routine

5.3 Shaders

The most important factor in this system is the shader programs. As can be seen from the previous section, the OpenGL code is concerned primarily with setting up the card status for the shaders to do all the detailed work. In this system the vertex programs will be responsible for calculating the correct offset position of each vertex, along with the lighting of the model, taking in to consideration the shadowing factor. The fragment program then identifies the existence of fur at each pixel and allocates it a colour (including the application of lighting and variance to the standard hair colour).

The shaders are not pre-compiled in this project. Instead, they are compiled by FurGLe when the program starts. The reason for this is so the shaders can be compiled specifically for the card present on the machine in question, resulting in a more efficient solution. The loading and operating of the shaders is controlled by the `CGControl` class.

The full listings of the Cg Shader Programs are included in Appendix C.

5.3.1 Shell Vertex Shader

We shall first consider the shell vertex shader. The input parameters to this program are:

Float4 position : POSITION	Local space location of the incoming vertex. Passed to shader via <code>glVertex*</code> .
Float3 normal : NORMAL	Vertex normal (hair direction). Passed to shader via <code>glNormal*</code> .
Float3 texCoord : TEXCOORD0	x/y elements represent texture co-ordinate for fur set textures. Passed to shader via <code>glMultiTex*</code> .
Float2 colCoord : TEXCOORD1	Co-ordinates of colour map. Passed to shader via <code>glMultiTex*</code> .
Float3 tangent : TEXCOORD2	Tangent for the current vertex. This was included for per-pixel lighting, but is unused in the final version. Passed to shader via <code>glMultiTex*</code> .
Float4 wind : TEXCOORD3	The wind vector for the vertex. x/y/z elements represent direction. w element indicates the length of the hair. Passed to shader via <code>glMultiTex*</code> .
Uniform 4x4 modelViewProj	The model-view-projection matrix
Uniform float3 eyePosition	The location of the camera in local space
Uniform float3 lightColor	The colour of the light in rgb format
Uniform float3 lightPosition	The location of the light source in local space
Uniform float displacement	The distance from the skin of the current shell, in range [0-1]
Uniform float density	The overall density factor of the fur type currently being rendered, in range [0-1]

Table 5.1 – Input parameters for shell vertex program

The uniform properties are set just once between the renderings of a fur type at a certain displacement. Any dynamic data, which varies every vertex, is packed into multi-texture co-ordinates. This is because integrating variable transfers into existing OpenGL calls is considerably faster than making direct calls to the shader itself. The allocation of the uniform values is handled by the `CGControl` class. The light colour and position are constant throughout the program and are set at start-up when the shaders are compiled. Shell displacement and the projection matrix are set by calls to `cgSetDisplacement` and `cgSetMatrix` respectively and can be seen in the render code in figure 5.7. The density factor is specified every time the shader is activated as a change in shader program (from fin to shell rendering) always occurs when a new shell is started. The value this is to take is stored in the `fTexSet` structure that is passed to `cgActiveShaders`.

The first job of the vertex shader is to calculate the world position of the vertex. This is done in two parts. First, the vertex position is offset in local space in regards to the displacement and wind vector values using the equations from Section 3.4.3. This resulting position must then be multiplied by the model-view-projection matrix to give the final screen space location. This process is shown in Figure 5.8

```

//Get bend values
float3 localBend = IN.wind.xyz - (dot(IN.normal, IN.wind.xyz) * IN.normal);
float force = length(localBend);
if(force > 1.0) {
    localBend = normalize(localBend);
    force = 1.0;
}

//Calculate bend factors
float d2 = displacement * displacement;
float f2 = force * force;
float wf = (0.861986 * force) - (0.176676 * f2);
float hf = 1.0 - (0.04743 * force) - (0.36726 * f2);
hf = (hf * d2) + (displacement * (1 - displacement));
wf = wf * d2;

//Set displacement
float4 nww;
nww.xyz = (IN.normal * hf * IN.wind.w) + (localBend * wf * IN.wind.w);
nww.w = 0.0;

//Set position, and texture co-ordinates
float4 PS = mul(modelViewProj, (IN.position + nww));
OUT.oPos = PS;

```

Figure 5.8 – The positional vertex shader code

The other purpose of the vertex shader is to calculate the lighting. This is performed using the equations specified in Section 3.2.4. During the development of this process, it was found that pure anisotropic lighting produced unrealistic results where hairs pointing directly towards the light were rendered too dark. Therefore a small Lambertian diffuse component was added to the lighting function to simulate the reflective light these hairs would receive from those around them.

Unfortunately, as has already been stated, per-pixel lighting was not completed within the deadline due to complications passing the normal data between programs and processors. If this was to be implemented, the ambient, specular and Lambertian diffuse components would still be calculated in the vertex shader, with just the anisotropic diffuse component being calculated in the fragment shader. This would be the best approach as it would greatly reduce the number of calculations performed, compared to that required for the full lighting model, and as the diffuse component is the strongest factor, the reduction in quality would be barely noticeable.

Once the lighting has been calculated, the shadowing factor is derived as indicated in equation 3.4. A second shadowing function, representing the object-over fur shadowing component is also calculated using equation 5.1, shown below. Recall that the shadowing function is multiplied by the light intensity, so smaller shadow factors produce the greatest shadow.

$$\text{If } L \cdot N + 0.5 < 0.5 \text{ then } 1 \text{ otherwise } 0 \quad (\text{Eq. 5.1})$$

Where L is the light vector and N is the hair direction

The minimum value of the two shadowing factors is then taken to be the true shadow factor and the overall lighting value is calculated as shown in Figure 5.9. The position and lighting values, along with the texture co-ordinates that are un-used by the vertex program, are then passed out as TEXCOORD* values for interpolation between the polygon fragments by OpenGL.

```

//Shadow Components
float FoFSP = (1 - max(dot(IN.normal, L), 0));
float FoFS = clamp(((4 * (displacement / density)) - (3 * FoFSP)) / FoFSP, 0.0 +
    (1.0 - density), 1.0);
float Shadow = min(FoFS, saturate(dot(L, IN.normal) + 0.5));

OUT.oLight.w = Shadow;
OUT.oCol = ambient + (((diffuse * 0.35) + (ddiffuse * 0.15) + (specular * 0.3))
    * lightColor * Shadow);

```

Figure 5.9 Calculating the resultant lighting value

5.3.2 Shell Fragment Shaders

Once the vertex program has defined the position and properties of the polygons, it is the fragment shaders job to render them. The input values to fragment shader are:

Float4 texCoord : TEXCOORD0	x/y elements represent the texture co-ordinates of the fur type, z element is the existence value of fur at that point in range [0-1], and the w element is the current displacement in range [0-1]. All values specified by vertex shader
Float2 colCoord : TEXCOORD1	The texture co-ordinate for this fragment on the colour map. Specified by VS
Float3 color : TEXCOORD2	The resultant light value for the taking into consideration light colour and shadowing. Specified by the vertex shader
Float4 lightDir : TEXCOORD3	This represents the light vector being passed to the fragment shader. This was included for per-pixel lighting but was never fully implemented. The fourth element was to store the shadow factor for application to the final diffuse component.
Uniform sampler2D shellTex	This is a pointer to the fur set texture that is to be used.
Uniform sampler2D shellCol	This is a pointer to the fur colour texture that is to be used.
Uniform sampler2D shellNorm	This is a pointer to the fur normal map that would have been used.

Table 5.2 – Input parameters to the shell fragment program

The texture samplers are specified in `CGControl` when the shader is activated. When the call to `cgActivateShaders` is made, the `fTexSet` of the current fur type is passed as a parameter. As `CGControl` already knows what displacement level is currently set, from an earlier call to `cgsetDisplacement`, it can select and bind the appropriate textures from the texture set as it activates the shader program itself.

Although the fragment shader performs numerous techniques from the review and analysis stages, the final process is remarkably simple. First of all, the textures are referenced to obtain the dominance, variance, alpha and colour values of the current fragment. It would logically follow that the fragment is checked for existence at this point, but because of the architecture of current GPU's this is not the most efficient approach.

This is down to two reasons. Firstly, a fragment cannot be discarded until the program has finished running, so where you check for existence makes no difference to performance. Secondly, true branching is not supported, so all `if` statements are inlined. For example, the code...

```

If (a) {B, C} else {D, E}
...is converted to...
If (a) then B
If (a) then C
If (!a) then D
If (!a) then E

```

...for implementation on the current GPU architectures. What this means in this case is that checking for existence at the beginning of the shader and only performing the calculations if a hair is present would result in the existence check being performed for every instruction in the program. Due to the length of the program at this point in time, the efficiency difference is only about 10%, but for a further enhanced version the improvements would be huge.

Therefore, the shader first calculates the colour of the fragment under the presumption that a hair exists. This is achieved by taking the referenced colour from the input map and multiplying it by the variance and lighting values. The existence value is then calculated as defined in Eq. 3.2.

Finally, the existence check is performed. Note that the alpha value is set to zero before the check to avoid an else branch from being inlined. If the fragment is part of a hair this alpha value is then changed to that in the fur set, which may or may not be an opaque value. The code for this shader is shown in figure 5.10

```

struct appdata {
    float4 texCoord : TEXCOORD0;
    float2 colCoord : TEXCOORD1;
    float3 color     : TEXCOORD2;
    float4 lightDir  : TEXCOORD3;
};

void main(appdata IN,
    uniform sampler2D shellTex,
    uniform sampler2D shellCol,
    uniform sampler2D shellNorm,
    out float4 Color : COLOR)
{
    float4 DomRef = tex2D(shellTex, IN.texCoord.xy);
    float4 ColRef = tex2D(shellCol, IN.colCoord);
    Color.xyz = ColRef.xyz * IN.color * DomRef.x;
    Color.w = 0.0;
    float exist = max(2.0 - (2.0 * ((1 - IN.texCoord.z) / DomRef.y)),
        0);
    if(IN.texCoord.w <= exist) Color.w = DomRef.z;
}

```

Figure 5.10 – The shell fragment program

5.3.3 Other Shaders

As the different components of this project utilise the same underlying techniques, the resulting shaders are very similar to the one just described. This section indicates the primary differences between them.

Fins

The fins shaders share the same placement, lighting and shadowing components as are found in the shells. There are however some differences to support the varied alignment of the polygons.

In the vertex shader, only the x co-ordinates for the fur texture are provided. The y co-ordinate is derived from the current displacement value, and a vertical offset passed with the texture data. This offset is required to indicate the top and bottom of the current displacement, as unlike with the shells approach, the layers present with the fins have depth.

The vertex shader also calculates the alpha value of the fin to eliminate the popping affect described earlier. This alpha value is then passed to the fragment shader in place of the displacement value. This can be done as the y texture co-ordinate will be a more accurate representation of the offset at this point than the uniform displacement value, so the displacement value would not be used if it was still sent.

On the fragment side, the only differences present in the code are the multiplication of the final result by the alpha value generated by the vertex shader, and the use of the y texture co-ordinate as a height reference, rather than the displacement value passed across in the shells program.

Whiskers

The important thing to note about whiskers is that they are singular entities, drawn with infinitesimally thin lines. As they have no volumetric area, they do not need a fragment shader as all the lighting and colouration is done in the vertex shader.

5.3.4 Testing

Testing is a difficult operation with Cg programs as there is extremely limited feedback making it impossible to print out check variables. Therefore the majority of the testing was done by perception, and occasionally by outputting check variables via the colour channels of the pixel. This channelling approach proved extremely useful, but has limitations in that only an approximation of a given value can be obtained by viewing an on screen colour.

To test the dynamics, force lines were rendered on the model showing the direction and magnitude of force. These are still visible in Edit Mode of the final product. This proved considerably useful, particularly when problems were encountered in swapping between spaces for the movement based calculations.

5.4 User Interfaces

FurGLE has 3 different user interfaces, each of which implements the general Interface class shown in Figure 5.11. Each of these interfaces are described below, and the screen layouts are shown in Figure 5.12

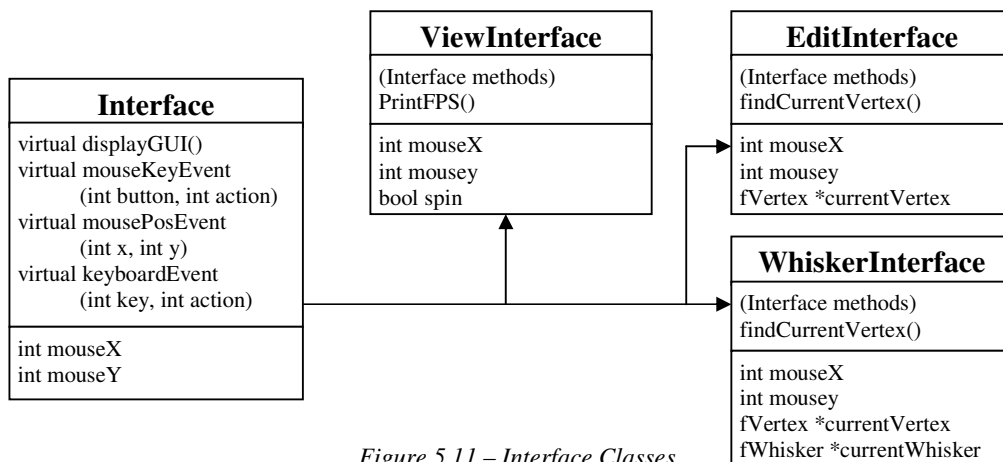


Figure 5.11 – Interface Classes

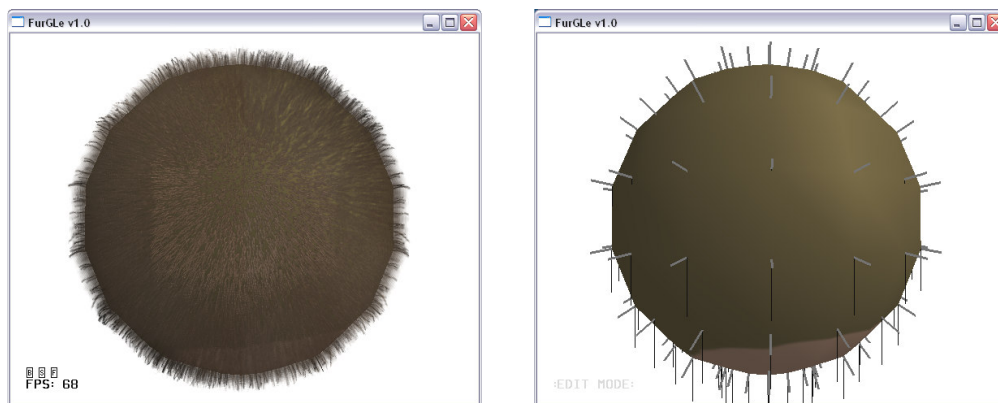


Figure 5.12 – Interface screenshots

5.4.1 View Mode

This is the default user interface which is used to view the model. The control method is:

- LMB + Drag - Rotates the model for alternate viewing angles
- RMB + Drag - Zooms in and out of the model
- LMB + LCTRL + Drag - Spin the model around the Z axis (Yaw)
- B, S, F - Pressing these keys toggles visibility of the base model, shells and fins respectively.

Boolean values are used to store the status of the mouse buttons, and movement is detected by comparing the current mouse position with the previous mouse position. Using these values the corresponding Camera class methods are called to reposition the viewing position.

The separate visibility settings are achieved by altering the `renderFur`, `renderFins` and `renderSkin` flags in the Main program when the corresponding key press is detected.

5.4.2 Edit Mode

If the user holds down the E key, they will enter edit mode. This allows the length, direction and type of the fur to be altered at each vertex. The control system is:

- LMB – Select vertex
- LMB + Drag – Change fur direction
- RMB + Drag – Vary length of fur
- 1, 2, 3, 4 – Set's fur type at current vertex.

This mode depends heavily on being able to select a vertex in 3D space using selection on a 2D plane. To do this, `gluUnProject` is used to obtain 3D positions on the near and far planes of the view volume relative to the position on the screen. Note that to do this the screen y co-ordinate value must be inverted, as OpenGL's y axis is opposite to that in Windows.

Once the two points have been identified, a ray indicating the points in 3D space that could possibly be selected can be constructed by Eq. 5.2. By giving each vertex a spherical selection volume, potential vertices can be found via a ray-sphere intersection test via equation. 5.3 [23]. The closest successful vertex from this test will then be the selected vertex. At this point, a pointer to the vertex is obtained, making the alteration of the fur properties a trivial process.

$$\text{nearPos} + i(\text{farPos} - \text{nearPos}) \quad (\text{Eq. 5.2})$$

$$a = i^2 + j^2 + k^2$$

$$b = 2i(x_1 - l) + 2j(y_1 - m) + 2k(z_1 - n)$$

$$c = l^2 + m^2 + n^2 + x_1^2 + y_1^2 + z_1^2 + 2(-lx_1 - my_1 - nz_1) - r^2$$

If $b^2 - 4ac \geq 0$ then the ray and sphere intersect (Eq. 5.3)

Where the ray starts at (x_1, y_1, z_1) with direction (i, j, k)
and the sphere has center at (l, m, n) with radius r


```

extern fObject *furryObj;
float CHECK_RADIUS = 0.1f;

double px,py,pz;
double modelMatrix[16]; glGetDoublev(GL_MODELVIEW_MATRIX, modelMatrix);
double projMatrix[16]; glGetDoublev(GL_PROJECTION_MATRIX, projMatrix);
int viewport[16]; glGetIntegerv(GL_VIEWPORT, viewport);
int wx, wy;
glfwGetWindowSize(&wx, &wy);
int windowY = wy - mouseY;
gluUnProject(mouseX, windowY, 0.0, modelMatrix, projMatrix, viewport, &px, &py, &pz);
Vector3d nearPos = Vector3d((float)px, (float)py, (float)pz);
gluUnProject(mouseX, windowY, 1.0, modelMatrix, projMatrix, viewport, &px, &py, &pz);
Vector3d farPos = Vector3d((float)px, (float)py, (float)pz);
Vector3d ijk = farPos - nearPos;
float vertexDist = 10000.0f;

float a = (ijk.x * ijk.x) + (ijk.y * ijk.y) + (ijk.z * ijk.z);
float cf = (nearPos.x * nearPos.x) + (nearPos.y * nearPos.y) + (nearPos.z * nearPos.z);

vector<fVertex>::iterator vc = furryObj->vertexList.begin();
while(vc != furryObj->vertexList.end()) {
    float b = (2 * ijk.x * (nearPos.x - vc->position.x)) +
              (2 * ijk.y * (nearPos.y - vc->position.y)) +
              (2 * ijk.z * (nearPos.z - vc->position.z));
    float c = (vc->position.x * vc->position.x) + (vc->position.y * vc->position.y) +
              (vc->position.z * vc->position.z) + cf + (2 * (0.0f - (vc->position.x *
              nearPos.x) - (vc->position.y * nearPos.y) - (vc->position.z * nearPos.z))) -
              (CHECK_RADIUS * CHECK_RADIUS);
    float det = (b*b) - (4 * a * c);
    if(det >= 0.0f) {
        float newVertexDist = (vc->position - nearPos).magnitude();
        if (newVertexDist < vertexDist) {
            currentVertex = &(*vc);
            vertexDist = newVertexDist;
        }
    }
    vc++;
}

```

Figure 5.13 – Code for selecting a vertex.

5.4.3 Whiskers Mode

Whiskers mode is initiated when the W key is pressed. The appearance is the same as edit mode, but the controls are tailored towards the editing of whiskers. The controls are as follows:

- LMB – Create/Remove whisker at current vertex
- RMB + Drag – Increase/Decrease length of the whisker.

The same method is used to identify the selected vertex as with the previous interface, when creating a whisker. Once the vertex is obtained, the list of whiskers currently in the system is checked to see if an entry exists that is linked to that vertex. If a whisker is found then it is removed from the whisker list, otherwise a new `fWhisker` instance is created relative to the selected vertex and added to the system.

5.4.4 Testing

The features of each interface were tested and tweaked accordingly during the development process. All functionality of the interfaces are now correct and easy to use. However, during more frequent use, whilst generating models for the next chapter, it was found that the program occasionally terminates abruptly during the vertex picking algorithm shown in Figure 5.11. It is unclear why this happens and all debugging attempts have failed to find any fault with the code. This problem is not a frequent occurrence, so is therefore not a major problem, but should be resolved if the system was to be extended for industrial or public use.

Chapter 6: Results and Evaluation

This chapter considers the success of the project as a whole, and considers its potential for future development. This is done in several stages. First, Section 6.1 considers the individual requirements of the project, and the degree to which they have been achieved. Section 6.2 then evaluates the quality of textures generated for use in the system, comparing the types generated to their real-life counterparts, before Section 6.3 evaluates the system as a whole. Finally, the future potential for this technique is discussed in Section 6.4, before a summary of future work in Section 6.5.

6.1 Fur Rendering

At the beginning of the analysis section, a number of objectives were set. This section considers the degree of success the individual parts of the system in regards to these objectives.

6.1.1 Fur Features

Shells

The implementation of shells in this system is of a high standard, with minimal faults identified. The only negative point that is immediately apparent is the lack of support for LOD. Unfortunately, there is no obvious solution to this problem due to the manner in which the shell textures are saved in a static-sized array.

The program itself uses 16 shells for rendering fur. This was judged as the best balance to produce reasonably efficient frame rates whilst maintaining a coherent image. Further experimentation has shown that the shell quantity can be dropped to 12 before the image begins to noticeably break apart. Therefore, it is feasible to suggest this as the ideal quantity for use in any marketable product that adopts the shells method.

Overall, the shells approach has proven to be a very good base method for rendering fur. It is versatile enough to handle a range of situations and has huge potential for continual expansion.

Fins

The inclusion of fins has a considerable affect on the quality of silhouettes, as seen in Figure 6.1. Unfortunately, due to some unidentifiable errors in system the fin selection routine contains some margin of error, as is clear when the shells are deactivated. Some fins that should be rendered are absent from the final image, and some silhouettes near-parallel with the viewing direction are rendered, producing dark 'spots' as seen below. However, it is felt this is due to errors in the implementation, and not the method itself, so fins remain a worthy extension to the shells method.

The main fin-based finding in this paper is that the alpha fins method is necessary to prevent popping, but need only be implemented over a small angle around the silhouettes.

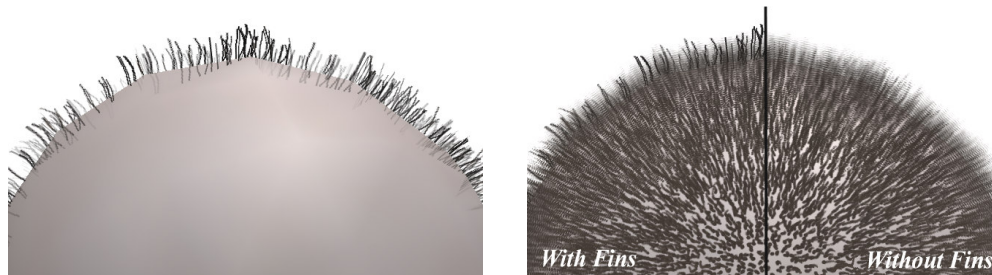


Figure 6.1 – Illustrations of fins in the final product

Length and combing

Although both these methods have a rather limited viable range, they add considerable realism to the overall result. There are no problems found with these functions in the system, nor were there any new findings.

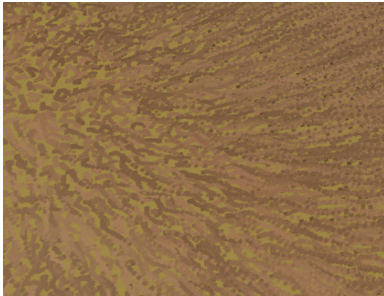
Lighting

Even though per-pixel lighting was never fully implemented, the lighting model is very effective. During the course of the project, it was found that introducing a traditional diffuse component to the lighting equation improved the overall realism of the rendered image. This extension is necessary as it models the global reflection light that will be obtained from the surrounding hairs.

Shadowing is implemented using an extension on Bank's Approximation method. The results are of good quality, and are highly efficient. Admittedly, they are not as realistic as implementing the shadow mapping approach would have been, but the effect is still believable and considerably faster.

Colouration

Due to complications involved in swapping between texture domains, the ATi offset approach was excluded from this project. It was deemed too expensive to warrant slight visual quality increase, which tests performed thus far support.



The concept of a variance map was also introduced to make slight variations to hair colour. The results from this prove to be very beneficial, as they make each individual hair easier to distinguish. Without this function, the fur looked very plain and flat, but the colour variations help to promote the 3D nature of the fur.

Figure 6.2 – Hair colour variance

6.1.2 Interpolation

The interpolation method described in this paper works remarkably well. Several revisions were made to the classification algorithm (Eq. 3.2) before the results were legible, but the final implementation provided a very suitable model for this feature. There are problems where hairs of different types pass through each other, but as the Z-buffer is disabled whilst the shells are rendered, this does not produce any detrimental graphic effects.

At the moment, a major limitation with this method is that each vertex can only be of one fur type. This means any transitions must occur over a single polygon from non-existence to full existence. This is a major limitation, as models will have to be specifically tailored to allow for realistic transitions. In some cases this may not even be possible. For example, there is no possible way under the current system to have an interpolation occurring over a curved surface.

A future extension would be to relax this constraint allowing fur to change more gradually across large areas of the model. This would be a fairly easy task as the shaders would already be able to support this extension due to the multi-pass nature in which it is rendered.



Figure 6.3 – Fur type interpolation

6.1.3 Dynamics

The concept of hair dynamics has also been successfully introduced in this paper. The wind vector approach adopted for bending the hairs, has, after some modification for current GPU architectures, proven to be a suitable and reasonably efficient method for displacing the shell locations correctly. Figure 5.5 below proves this by showing the status of a furry sphere under different physical conditions.

The current implementation does have one major downfall, in that physics model used to calculate the wind vectors is rather primitive. The task of calculating physics accurately, particularly for structures as complex as fur, is a very difficult process and was never intended to be the focus of this extension. However, now that the method is proven as valid, building a scientifically correct implementation of physical modelling is a logical extension, and would bring a great increase to the realism of the dynamics.

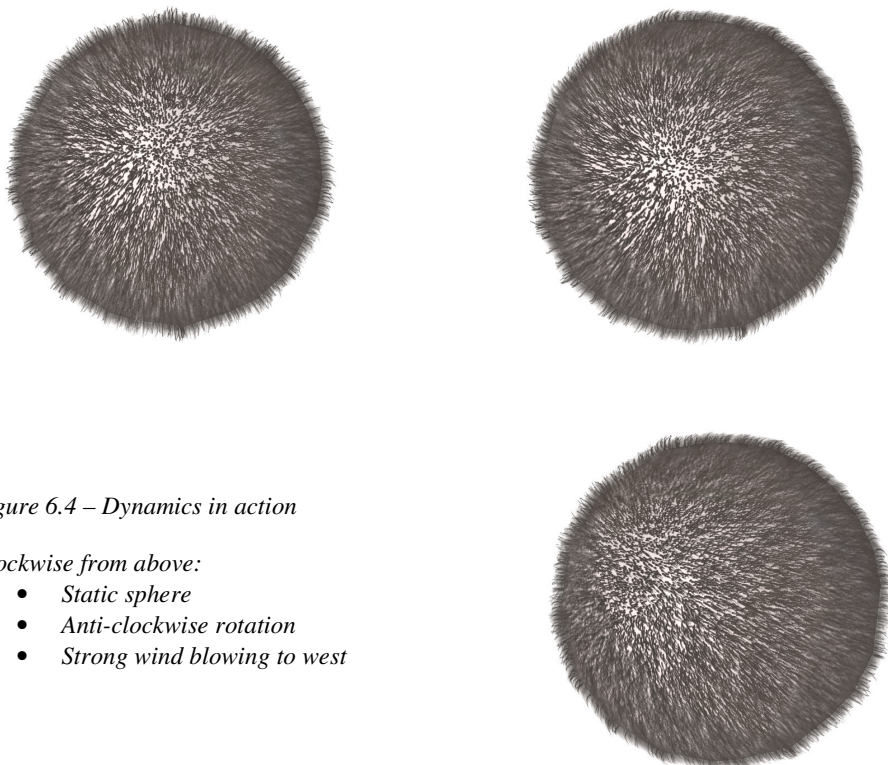


Figure 6.4 – Dynamics in action



Clockwise from above:


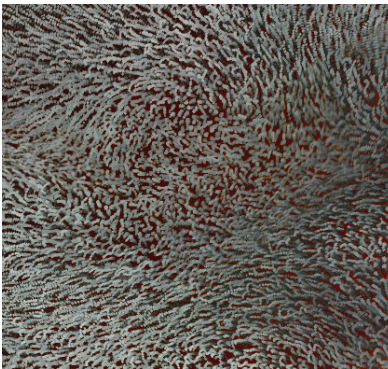
- Static sphere
- Anti-clockwise rotation
- Strong wind blowing to west

6.2 Fur Generator

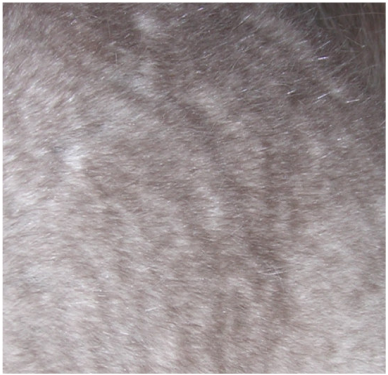
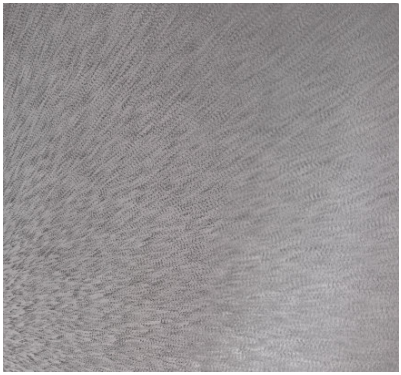
In this section, the ability to generate different fur types is considered. A number of real fur images have been obtained and attempts have been made to replicate the fur in the system. The results are shown below.

6.2.1 Results

<i>Fur Type Generation Test 1 – Rabbit Fur</i>		
	Thickness:0.45	
	Density:0.75	
	Curliness:0.4	
<p>As illustrated above, rabbit fur can be quite accurately generated by FurMak. The only fault with this representation is that the full hair length cannot be replicated with just 16 shells. If the hairs are extended in length the correlation between the hair segments breaks up. However, unless the results are being directly referenced as they are here, the length difference would not be noticeable.</p>		


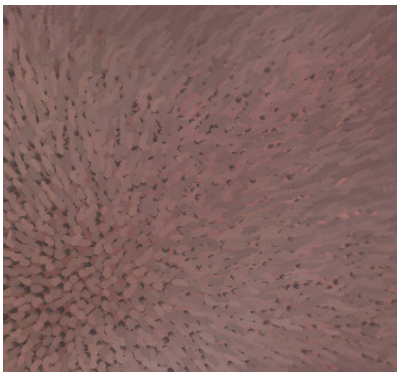
<i>Fur Type Generation Test 2 – Border Collie (Sheep-dog) Fur</i>		
	Thickness:0.45	
	Density:1.0	
	Curliness:1.0	
<p>The generation of curly dog fur was less successful. In the real fur, large groups of hairs curl in a very similar direction producing well defined ‘rolls’ in the fur. FurMak does not support such hair groups, but the result is mimicked by changing the normals of the furry model. The main problem, however, that cannot be cheated is that the real fur curls back on itself. This cannot be done in this system, but may be possible by allowing the generation particles full freedom of movement along all 3 axis.</p>		

Fur Type Generation Text 3 – Chinchilla Fur


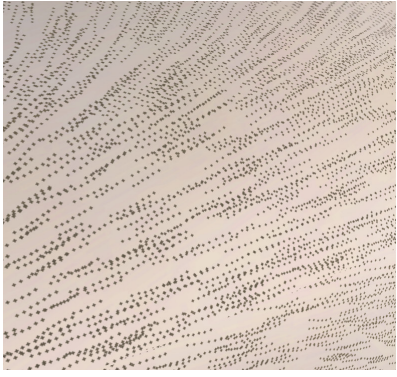
	Thickness:0.3	
	Density:1.0	
	Curliness:0.1	

A Chinchilla's coat is possibly the perfect type of fur to show off the shells method. The extremely high quantity of hairs is easily modelled and can be displayed at equally fast speeds to any lower frequency fur, unlike the naïve or polygon strip approaches. The texture frequency on the model had to be increased to get the hair fine enough, but the overall result is extremely good.

Fur Type Generation Text 4 – Synthetic Fur from Novelty Slippers

	Thickness: 0.8	
	Density:1.0	
	Curliness:0.45	

This thick nylon 'fur' is very difficult to replicate. A suitable representation has been achieved by using thick hairs, but in reality the fur type should consist of many smaller hairs woven together. This could be resolved by introducing a grouping property to future revisions of the fur generator. There are also a number of these thin threads that do not get bound to a group, adding to the fuzziness of the fur type which must also be represented.

<i>Fur Type Generation Text 5 – Human Forearm</i>		
	Thickness:0.28	
	Density:0.15	
	Curliness:0.8	
<p>The fine, spare hair is quite easy to model in this system, but unfortunately the combing required breaks the hair segments apart resulting in a dotted image than only barely resembles the intended fur type. There is nothing wrong with the generated textures, just a strong limitation on the combing abilities.</p> <p>If the 3-dimensional freedom of movement was implemented for the particles it could be possible to add another constraint to increase the lateral range of the particle so that a line, rather than a dot, could be made on the texture as the particle runs along the shell plane. This would increase the visual quality of long short hairs, but would add a directional element to the fur type, limiting the beneficial ‘use-anywhere’ property of the fur textures generated in this project.</p>		

6.2.2 Evaluation

As can be seen, the fur generator can reproduce most common fur types to a reasonable degree. There are a few instances where the fur type must be generalised to a simpler representation, but the results still appear valid unless studied closely. This shows that particle based generation is a suitable method for the creation of fur types, and is evidently far superior than the results that would have been obtained via the noise model.

There are two downfalls of the fur generator that have been identified, that present the opportunity for further extension. Firstly, the particles could be extended to have 3-dimensional freedom of movement to allow for more elaborate curls. Secondly, a grouping function could be included to clump individual fur together. This would make it possible to model a wider range of synthetic furs and open the opportunity for non-standard fur types such as a sheep’s fleece and wet or matted fur.

It is also evident from these images, and from figure 6.5 on the following page, that the intermediate cube surface approach to mapping the model with a secondary texture domain is a success. The fur patches are the same size throughout, and it is very difficult to see the joins caused by face changes in the texturing process. Both these problems still occur of course, and could be resolved using an applied ABF approach (2.4.3), but the anomalies are not considered evident enough to cause any problems or warrant any further development.

6.3 Overall System

To test the system as a whole, the following model of the Stanford Bunny [24] was created. This incorporates all the features included in the system.

There are 3 hair types present each with its own colour map, as well as a skin texture. The fur length varies across the body, being shorter around the head and longer on the tail, and is combed via the varying of normals to give a more realistic affect.



Figure 6.5 – Stanford Bunny

It is evident from this that all sections of the system work together suitably to produce high quality results. The frame rate for the above model generally varied between 15-25fps when run at a resolution of 1280x960, depending on orientation and screen area of the model. The lowest frame rate attained was 6fps, which was a result of zooming in on the tail hair transition (shown in Figure 6.6) In reality, this view would never be obtained, so the overall efficiency can be viewed as ‘real-time’.

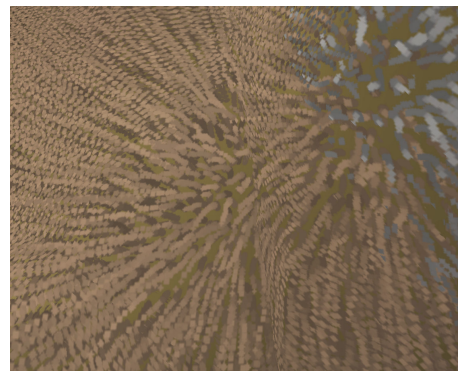


Figure 6.6 – Slowest attainable viewpoint

There are however some problems with the model in the previous section. Firstly, the rabbit does not look entirely realistic, but I feel that this is due to my inexperience with 3d modelling packages and drawing textures, and is not the fault of the system itself. The colouration textures used here are fairly simplistic resulting in a fairly plain looking model, and the mesh itself is not fully set up correctly as can be seen with the incorrect locations of the eye and tail fur transitions. Both of these problems could be resolved by a trained artist.

Secondly, the silhouette features some anomalies, with some sections appearing to have considerably more hair than others. This is caused by the slight faults with the fin rendering indicated in section 6.1.1. In the full model, these are increasingly evident as the combed normals make the likelihood of miscalculation considerably higher.

6.4 Usability

It is clear from the results obtained over the last 7 pages that the shells and fins method produced good quality results. However, to be truly useful, the system must be efficiency enough to achieve 30+ fps.

Due to time restrictions on the project, the program has not been fully optimized, so falls short of this high target. As described in Section 5.2.2 of the last chapter, the model is stored in Display Lists that must be constantly updated. As each list is used at least 16 times this is still considerably faster than immediate mode would be. However, if the data structure was rearranged to work with buffered array lists instead, the transfer of data to the GPU would be noticeably faster.

Another possibility for increasing efficiency is to offload some of the work back to the CPU, such as offsetting the shell vertices. At the moment, the bottleneck of the system is in the vertex shader and moving these calculations could improve speed slightly, although it would result in less processor time available for any system in which the fur would be implemented.

Although the bottleneck is in the vertex shader, the pixel shader is not far behind. Although the program lengths are extremely short for fragment shaders, the number of fragments being calculated can be many times the resolution of the screen (due to the shells overlaying one another). This is obviously a major problem that would need considering when adding extra detail to the system, such as the full implementation of per-pixel lighting.

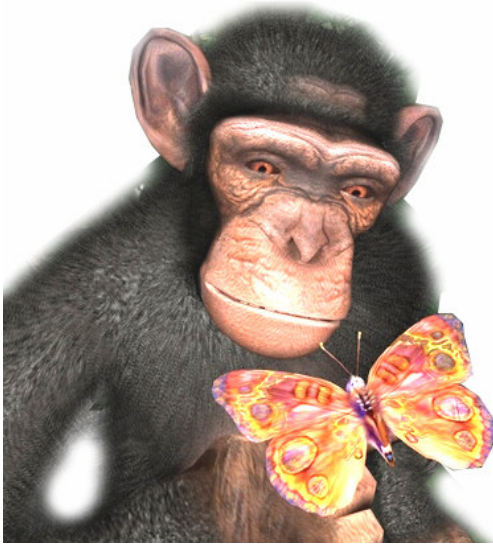
If the vertex-based efficiency issues were resolved, this method of rendering fur would certainly be a viable option for implementing in real systems, such as games. The project is GPU heavy so the extra processing required on the CPU to run a full system would not affect the rendering. Also, furry objects in games would cover a much smaller area than a presentation based project such as this, so the fragment shaders could be more elaborate without causing speed issues.

6.4.1 Comparison to existing methods

Now that the technique has been evaluated, it is compared with other implementations of real-time fur available today.

ATi Chimp

As a demonstration for their 9800 family of cards, ATi developed the Chimp demo [25]. This is the system frequently referred to during this paper [2]. As it is an in house demo it is extremely efficient, which proves the huge potential for the shells method if tailored specifically to a particular chipset. The visual result also shows how good a model can look when engineered by professional artists.



The ATi approach implements per-pixel lighting which is noticeable improvement. The offset approach for colouration is not apparent from the demo, which supports my theory to exclude it.

Features not present on the ATi Chimp are fur dynamics and multiple fur type transitioning. It is also unclear what shadowing method, if any, is adopted.

Overall, this demo proves the potential efficiency of the method, supports my choice to exclude offset mapping, and suggests my extensions of fur type interpolation and dynamics to be worthy advancements.

*Figure 6.7 – ATi Chimp Demo
Permission pending from [25]*

Black & White 2

This upcoming game from Lionhead Studios [26] includes a rather impressive fur renderer using the polygon strips approach that was discarded in the literature review.

The results are extremely good, both by visual quality and efficiency. The strips approach allows for much longer hair than the shells method, and the visual quality around the silhouettes of the model are considerably better. Dynamics are also included, and are of very high quality.

The down-side to the strips approach is the messier, more cartoon-like appearance. This is because the strips have to be arranged rather irregularly to create a wide viewing scope, making combing difficult to achieve. The shorter fur also has a slightly less impressive appearance than with the shells model.

It would be interesting to see if the improved silhouettes of this method could replace the fins present with the shells approach to produce a hybrid technique. This would then have the benefits of both methods.

*Figure 6.8 – Black & White 2 ‘Good Ape’ Model
Used with permission from [??]*



6.4.2 Future Potential

Although this method could be used in modern day systems, the method would need further work, primarily on efficiency issues, to meet the strict graphical needs of the industry. At this point in time programmable pipeline architectures are still quite primitive making further enhancement difficult. However, the power of GPU's are increasing at incredible rates, making an enhanced version of this method a very viable option for any projects in the early stages of development. Here we consider the future of the method in relation to upcoming technological advancements.

This summer will be seeing the introduction of DirectX 9.0c and its implementation of Shader model 3.0. This will introduce full branching, increasing the power of conditional calculations and making recursion possible. There is also an increase in the number of instructions available (in excess of 65,000) and a large increase in available memory registers [27]. These features give potential for more complex features, such as the shadow mapping technique (2.2.4), to be implemented for future hardware.

The throughput of the shaders is also increased with the potential for the multi-texture pixel shaders to be up to twice as fast as they are now. This means that within the next 6 months there will be cards on the market that could run the shells approach at competitive rates, making it increasing viable to be a key method in future interactive products.

The most exciting news however is a vertex shader process called instancing. Details are a little vague on this feature at this point of time, but the concept is that a whole model can be drawn in several places at the same time with small programmable variations. Depending how this is implemented, it may be possible to calculate all the shell vertices on a model in a single pass.

6.5 Future Work

To summarise, the shells method is an extremely promising method for rendering real time fur, but there is still many areas for extension or improvement. The further work that would be of the greatest benefit are considered to be as follows:

- **Improved fur generator** – Support for 3-dimensional freedom of movement for the hair particles, and a clumping factor to better replicate synthetic and matted fur, would greatly increase the range of fur types that could be implemented by this system.
- **Improved efficiency** – The method could be made much more efficient through the use of buffered vertex arrays and enhanced shader programs. With the 3.0 shader model it may even be possible to render all shells in a single pass.
- **Replace Fins with Polygon Strips** – The polygon strip approach in Black & White 2 produces very good results. It would be interesting to experiment mixing the two methods by replacing the faulty fin routine with a more constraint-free polygon strip approach
- **Per-Pixel Lighting** – Due to time constraints, per-pixel lighting was not achieved, but much of the code remains for correction/extension. Attaining this would produce a great increase in visual realism.
- **Improved Dynamics** – The dynamics process in this project proves that hair animation is possible with the fins technique. However, the results are unrealistic and could be improved with a more accurate physics engine.

Chapter 7: Conclusions

The primary project aim was to find a method for rendering short fur in real-time. The shells and fins approach by Lengyel et al [7] was adopted for this purpose, and proved to be very suitable. Several areas were identified for improvement in addition to the original objectives of interpolation and dynamics, and are discussed below.

Pure anisotropic lighting was used for the fur, but it soon became clear that areas of hair pointing towards the camera were being rendered too dark. This was because the reflective lighting that would be cast upon these hairs from those around them was not modelled. To resolve this, a small Lambertian diffuse component was added to the lighting model, and proved to be a great improvement. Another property of lighting that was developed further is fur-over-fur shadowing. The solution is based around the Bank's Approximation method, which darkens the sections of fur closest to the skin. Considerations of light direction and hair density were added to this model to produce an affective shadowing function for short hair.

Another area considered in detail was the colouration of the individual hairs. An offset map approach was initially selected, but as the colouration procedure became more complex it was decided that the benefit of this feature was not worth the computation costs – an assumption which later comparisons with existing systems confirmed. The concept of hair colour variance was also developed to enhance the indication of individual hairs, giving more volume to the resulting image.

A successful approach was also found for the hair interpolation problem indicated at the start of the project. This makes use of a dominance map to indicate how long it takes for each individual hair to disappear as the fur types merge in to one another. A secondary program (FurMak) was also developed to prepare models for use in this system, which also includes a reasonably flexible fur generator for constructing the fur types.

The final area of dynamics was also fully implemented, although the physics behind it are somewhat primitive. However, the underlying physics engine could be easily changed without any need to recode any of the fur 'bending' procedure itself. Another benefit of the dynamics approach is that it is independent of any animation methods used on the creature being modelled. That is, it relies purely on detecting the world space location of points in space, so will continue to work correctly regardless of what methodology is implemented for animating the model itself.

The only slight problem encountered during the project is that of efficiency. The method in which the polygon data is sent to the GPU could easily be improved, but as the bottleneck is in the Shader Programs most of the time it is unclear what degree of improvement this would produce. However, considerations are made for the future of the method, and it can be said, with relevant certainty, that graphics hardware will soon be powerful enough to make this a very worthwhile method for rendering real-time fur.

References

1. M. Olano, J.C. Hart, W. Heidrich, M. McCool (2002) Real-Time Shading, A K Peters Ltd, p. 4-5, ISBN: 1-56881-180-2
2. J. Isidoro, J. L. Mitchell (2002) User Customizable Real-Time Fur, SIGGRAPH 2002 Sketch, San Antonio
3. J. T. Kajiya, T. L. Kay (1989) Rendering Fur With Three Dimensional Textures, Proceedings of SIGGRAPH 89, p. 271-280
4. Meyer, A. & Neyret, F. (1998) Interactive Volume Textures, Eurographics Rendering workshop 1998, p. 157 – 168
5. Neyret, F. (1998) Modelling, Animating and Rendering Complex Scenes using Volumetric Textures. IEEE Transactions on Visualization and Computer Graphics, 4(1), p. 55-70.
6. Lengyel, J. (2000) Real-time Fur. Eurographics Rendering Workshop 2000, p. 243-256.
7. J. Lengyel, E Praun, A Finkelstein, H Hoppe (2001) Real Time Fur over Arbitrary Surfaces, Proceedings of the 2001 symposium on Interactive 3D graphics, p. 243-256
8. F. Perbet, M-P Cani (2001) Animating Prairies in Real-Time, ACM Interactive 3D Graphics.
9. K. Ward, C. Ming (2003) Adaptive Grouping and Subdivision for Simulating Hair Dynamics, Proceedings of Pacific Graphics 2003.
10. D. Stalling, M. Zockler, H-C. Hege (1997) Fast Display of Illuminated Field Lines, IEEE Transactions on Visualization and Computer Graphics, 3(2), p. 118-128.
11. G. S. P. Miller (1988) From Wire-frames to Furry Animals, Proceedings on Graphics Interface '88, p. 138-145
12. G. Papaioannou, (2002) A Simple and Fast Technique for Fur Rendering, Technical Report, Department of Informatics, University of Athens.
13. E. Praun, A. Finkelstein, H. Hoppe (2000) Lapped Textures, SIGGRAPH 2000, New Orleans.
14. A. Sheffer, E. de Sturler (2001) Parameterization of Faceted Surfaces for Meshing using Angle-Based Flattening, Computational Science and Engineering Program, University of Illinois.
15. A. Sheffer, E. de Sturler(2002) Smoothing an Overlay Grid to Minimize Linear Distortion in Texture Mapping, ACM Transactions on Graphics, Vol. 21, No. 4
16. L. V. Ahlfors, L. Sario(1960) Riemann Surfaces, Princeton University Press, Princeton, New Jersey.
17. Fast Furrier Transform, <http://graphics.cs.uiuc.edu/~awu/furrier/index.html>, last visited: 02/12/03
18. B. M. Bakay (2003) Animating and Lighting Grass in Real-Time, Thesis, Department of Computer Science, University of British Columbia.
19. Y. Bando, B-Y. Chen, T. Nishita (2003) Animating Hair with Loosely Connected Particles, EG Computer Graphics Forum, Vol. 22 No. 3, p. 411-418

20. J. Grosjean (1986) Principles of Dynamics, Stanley Thorned, ISBN: 0-85950-295-3
21. R. Fernando, M. J. Kilgard (2003) The Cg Tutorial, Addison Wesley, ISBN: 0-321-19496
22. T. Funkhouser (2000) Ray Casting, Princeton University, COS 426
23. A. Watt, F. Policarpo (2001) 3D Games: Real-time Rendering and Software Technology, Addison Wesley, ISBN: 0201-61921-0
24. <http://www.cmlab.csie.ntu.edu.tw/~robin/courses/cg03/model/index.html>, last visited: 05/05/04
25. ATi 9800 Demos, <http://www.ati.com/developer/demos/r9800.html>, last visited 05/05/04
26. Black & White 2 Official Website, <http://www2.bwgame.com/static/bw2003/bw2.html>, last visited 05/05/04
27. <http://www.anandtech.com/video/showdoc.html?i=2023&p=2> , last visited 05/05/04

Appendix – Shader Programs

Shell Vertex Program

```

struct appdata
{
    float4 position : POSITION;
    float3 normal   : NORMAL;
    float3 texCoord : TEXCOORD0; // X,Y -> (U,V) : Z -> Existance
    float2 colCoord : TEXCOORD1;
    float3 tangent  : TEXCOORD2;
    float4 wind     : TEXCOORD3; // X, Y, Z -> WIND : W -> HAIR LENGTH
    float3 color    : COLOR;
};

struct vfconn
{
    float4 oPos : POSITION;
    float4 oTex : TEXCOORD0; // X,Y -> (U,V) : Z -> Existance : W -> Layer
    float2 oCMT : TEXCOORD1;
    float3 oCol : TEXCOORD2;
    float4 oLight : TEXCOORD3;
};

vfconn main(appdata IN,
            uniform float4x4 modelViewProj,
            uniform float3 eyePosition,
            uniform float3 lightColor,
            uniform float3 lightPosition,
            uniform float displacement,
            uniform float density)
{
    vfconn OUT;

    //Get bend values
    float3 localBend = IN.wind.xyz - (dot(IN.normal, IN.wind.xyz) * IN.normal);
    float force = length(localBend);
    if(force > 1.0) {
        localBend = normalize(localBend);
        force = 1.0;
    }

    //Calculate bend factors
    float d2 = displacement * displacement;
    float f2 = force * force;
    float wf = (0.861986 * force) - (0.176676 * f2);
    float hf = 1.0 - (0.04743 * force) - (0.36726 * f2);
    hf = (hf * d2) + (displacement * (1 - displacement));
    wf = wf * d2;

    //Set displacement
    float4 nww;
    nww.xyz = (IN.normal * hf * IN.wind.w) + (localBend * wf * IN.wind.w);
    nww.w = 0.0;

    //Set position, and texture co-ordinates
    float4 PS = mul(modelViewProj, (IN.position + nww));
    OUT.oPos = PS;
    OUT.oTex.xyz = IN.texCoord.xyz;
    OUT.oTex.w = displacement;
    OUT.oCMT = IN.colCoord;
}

```

```

//Set per-pixel lighting data
float3 L = normalize(lightPosition - IN.position.xyz);
float3 binormal = cross(IN.tangent, IN.normal);
float3x3 rotation = float3x3(IN.tangent, binormal, IN.normal);
OUT.oLight.xyz = mul(rotation, L);

//Lighting Components
float3 ambient = {0.2, 0.2, 0.2};
float diffuse = max(dot(L - (dot(IN.normal, L) * IN.normal), L), 0);
float ddiffuse = max(dot(IN.normal, L), 0);
float3 HW = normalize(L + normalize(eyePosition - IN.position.xyz));
float specular = pow(max(dot(HW - (dot(IN.normal, HW) * IN.normal), HW),
0),50);

//Shadow Components
float FoFSP = (1 - max(dot(IN.normal, L),0));
float FoFS = clamp(((4 * (displacement / density)) - (3 * FoFSP))/ FoFSP,
0.0 + (1.0 - density), 1.0);
float Shadow = min(FoFS, saturate(dot(L, IN.normal) + 0.5));

OUT.oLight.w = Shadow;
OUT.oCol = ambient + (((diffuse * 0.35) + (ddiffuse * 0.2) +
(specular * 0.25)) * lightColor * Shadow);

return OUT;
}

```

Shell Fragment Program

```

struct appdata
{
    float4 texCoord : TEXCOORD0; // X,Y -> (U,V) : Z -> Existance : W -> Layer
    float2 colCoord : TEXCOORD1;
    float3 color : TEXCOORD2;
    float4 lightDir : TEXCOORD3;
};

float3 expand(float3 v) {
    return normalize((v - 0.5) * 2);
}

void main(appdata IN,
    uniform sampler2D shellTex,
    uniform sampler2D shellCol,
    uniform sampler2D shellNorm,
    out float4 Color : COLOR)
{
    // Texture co-ordinates
    float4 DomRef = tex2D(shellTex, IN.texCoord.xy);
    float4 ColRef = tex2D(shellCol, IN.colCoord);

    // Per pixel Lighting
    //float3 light = normalize(IN.lightDir.xyz);
    //float3 normal = expand(tex2D(shellNorm, IN.texCoord.xy).xyz);
    //float LH = dot(normal, light);
    //float3 HN = normalize(light - (LH * normal));
    //float diffuseLight = dot(HN, light);
    //float3 diffuse = 0.5 * diffuseLight * IN.lightDir.w;
    //float3 diffuse = float3(0.5, 0.5, 0.5);

    //Set color, alpha
    Color.xyz = ColRef.xyz * IN.color * DomRef.x;
    Color.w = 0.0;
    float exist = max(2.0 - (2.0 * ((1 - IN.texCoord.z) / DomRef.y)), 0);
    if(IN.texCoord.w <= exist) Color.w = DomRef.z;
}

```


Fin Vertex Program

```

struct appdata
{
    float4 position : POSITION;
    float3 normal   : NORMAL;
    float4 texCoord : TEXCOORD0; // X -> U : Y -> Existance
                                // Z   -> Reverse Displacement (Fins)
    float2 colCoord : TEXCOORD1;
    float3 tangent  : TEXCOORD2;
    float4 wind     : TEXCOORD3; // X, Y, Z -> WIND : W -> HAIR LENGTH
    float3 color    : COLOR;
};

struct vfconn
{
    float4 oPos : POSITION;
    float4 oTex : TEXCOORD0; // X,Y   -> U,V : Z   -> Existance : W   -> Alpha
    float2 oCMT : TEXCOORD1;
    float3 oCol : TEXCOORD2;
    float4 oLight : TEXCOORD3;
};

vfconn main(appdata IN,
            uniform float4x4 modelViewProj,
            uniform float3 eyePosition,
            uniform float3 lightColor,
            uniform float3 lightPosition,
            uniform float displacement,
            uniform float density)
{
    vfconn OUT;

    //Get bend values
    float3 localBend = IN.wind.xyz - (dot(IN.normal, IN.wind.xyz) * IN.normal);
    float force = length(localBend);
    if(force > 1.0) {
        localBend = normalize(localBend);
        force = 1.0;
    }

    float disp2 = displacement - IN.texCoord.z;

    //Calculate bend factors
    float d2 = disp2 * disp2;
    float f2 = force * force;
    float wf = (0.861986 * force) - (0.176676 * f2);
    float hf = 1.0 - (0.04743 * force) - (0.36726 * f2);
    hf = (hf * d2) + (disp2 * (1 - disp2));
    wf = wf * d2;

    //Set displacement
    float4 nww;
    nww.xyz = (IN.normal * hf * IN.wind.w) + (localBend * wf * IN.wind.w);
    nww.w = 0.0;

    //Set position, and texture co-ordinates
    float4 PS = mul(modelViewProj, (IN.position + nww));
    OUT.oPos = PS;
    OUT.oTex.y = disp2;
    OUT.oTex.xz = IN.texCoord.xy;
    OUT.oTex.w = 1.0 - abs(dot(IN.normal, normalize(eyePosition -
                                                    IN.position.xyz))) * 3.0);
    OUT.oCMT = IN.colCoord;
}

```

```

//Set per-pixel lighting data
float3 L = normalize(lightPosition - IN.position.xyz);
//float3 binormal = cross(IN.tangent, IN.normal);
//float3x3 rotation = float3x3(IN.tangent, binormal, IN.normal);
OUT.oLight.xyz = L;

//Lighting Components
float3 ambient = {0.2, 0.2, 0.2};
float diffuse = max(dot(L - (dot(IN.normal, L) * IN.normal), L), 0);
float ddiffuse = max(dot(IN.normal, L), 0);
float3 HW = normalize(L + normalize(eyePosition - IN.position.xyz));
float specular = pow(max(dot(HW - (dot(IN.normal, HW) * IN.normal), HW),
0), 50);

//Shadow Components
float FoFSP = (1 - max(dot(IN.normal, L), 0));
float FoFS = clamp(((4 * (disp2 / density)) - (3 * FoFSP)) / FoFSP, 0.0, 1.0);
float Shadow = min(FoFS, saturate(dot(L, IN.normal) + 0.6));

OUT.oLight.w = Shadow;
OUT.oCol = ambient + (((diffuse * 0.35) + (ddiffuse * 0.2) +
(specular * 0.25)) * lightColor * Shadow);
//OUT.oCol = ambient * 5.0;

return OUT;
}

```

Fin Fragment Program

```

struct appdata
{
    float4 texCoord : TEXCOORD0; // X,Y -> (U,V) : Z -> Existance
                                // W -> Alpha
    float2 colCoord : TEXCOORD1; // X,Y -> (U,V)
    float3 color : TEXCOORD2;
    float4 lightDir : TEXCOORD3;
};

float3 expand(float3 v) {
    return normalize((v - 0.5) * 2);
}

void main(appdata IN,
    uniform sampler2D finTex,
    uniform sampler2D finCol,
    uniform sampler2D finNorm,
    out float4 Color : COLOR)
{
    // Texture co-ordinates
    float4 FinRef = tex2D(finTex, IN.texCoord.xy);
    float4 ColRef = tex2D(finCol, IN.colCoord);

    // Per pixel Lighting
    //float3 light = normalize(IN.lightDir.xyz);
    //float3 normal = expand(tex2D(shellNorm, IN.texCoord.xy).xyz);
    //float LH = dot(normal, light);
    //float3 HN = normalize(light - (LH * normal));
    //float diffuseLight = dot(HN, light);
    //float3 diffuse = 0.4 * diffuseLight;
    //float3 diffuse = float3(0.5, 0.5, 0.5);

    //Set color, alpha
    Color.xyz = ColRef.xyz * IN.color * FinRef.x;
    Color.w = 0.0;
    float exist = max(2.0 - (2.0 * ((1 - IN.texCoord.z) / FinRef.y)), 0);
    if(IN.texCoord.y <= exist) Color.w = FinRef.z * IN.texCoord.w;
}

```

Whiskers Vertex Program

```
struct appdata
{
    float4 position : POSITION;
    float3 normal   : NORMAL;
    float4 wind     : TEXCOORD0; // X, Y, Z -> WIND : W -> HAIR LENGTH
    float3 color    : COLOR;
};

struct vfconn
{
    float4 oPos : POSITION;
    float3 oCol : COLOR;
};

vfconn main(appdata IN,
            uniform float4x4 modelViewProj)
{
    vfconn OUT;

    //Get bend values
    float3 localBend = IN.wind.xyz - (dot(IN.normal, IN.wind.xyz) *
                                     IN.normal);

    float force = length(localBend);
    if(force > 1.0) {
        localBend = normalize(localBend);
        force = 1.0;
    }

    //Calculate bend factors
    float d2 = IN.position.w * IN.position.w;
    float f2 = force * force;
    float wf = (0.861986 * force) - (0.176676 * f2);
    float hf = 1.0 - (0.04743 * force) - (0.36726 * f2);
    hf = (hf * d2) + (IN.position.w * (1 - IN.position.w));
    wf = wf * d2;

    //Set position
    float4 npos;
    npos.xyz = IN.position.xyz + (IN.normal * hf * IN.wind.w) +
                    (localBend * wf * IN.wind.w);
    npos.w = 1;
    float4 PS = mul(modelViewProj, npos);
    OUT.oPos = PS;

    OUT.oCol = IN.color;

    return OUT;
}
```

