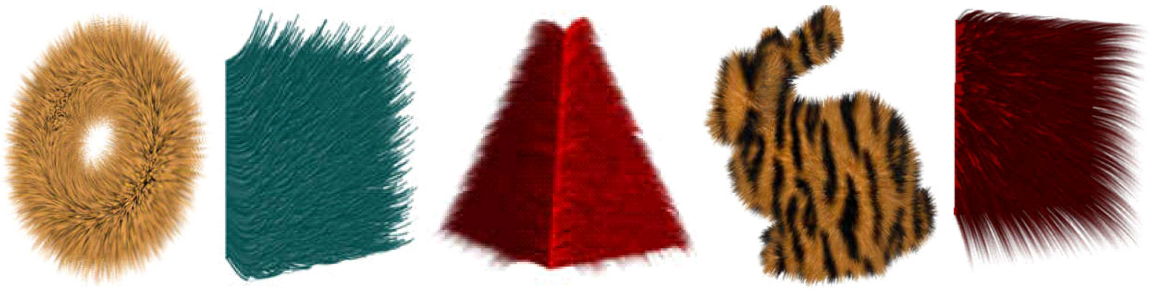
A Practical Guide to Generating Real-Time Dynamic Fur and Hair using Shells

Teddy Bears, Fuzzy Beards, Long Wavy Hair, and Furry Rabbits



Technical Course Notes

Monday, 10th February 2014

in

Organizer & Lecturer
Benjamin Kenwright
Edinburgh Napier University

Abstract

For natural scenes hair and fur is an essential element and plays an important role in multiple disciplines, such as virtual reality, computer games and cinematic special effects. Sadly, it is still difficult to render and animate hair and fur at interactive frame rates due to the huge number of strands in a typical real-world scene (e.g., a rabbit). Generating and simulating *realistic interactive and dynamic* hair and fur effects in real-time is one of the most challenging topics in computer graphics. In this course, we explain how shells provide an uncomplicated, computationally fast, and flexible method for creating life-like 3D fur and hair effects in *real-time* for interactive environments, such as games. We begin by providing a practical introduction to generating realistic-looking, fur and hair (e.g., different hair types with lighting and shadowing) using shells. We then move on to explain and demonstrate how simple low-dimensional physics-based models can be incorporated to produce dynamic and responsive hair movement. This allows our hair and fur method to be manipulated and controlled by the user through forces and texture animations. We show how Perlin noise in conjunction with artist created textures can create natural-looking *controlled* results. In conclusion, the fundamental contribution of this course demonstrates how an enhanced shell-based approach (i.e., shells with physics) offers an option for simulating aesthetically life-like dynamic fur and hair on-the-fly and in real-time.

About the Lecturer

Benjamin Kenwright
Edinburgh Napier University
b.kenwright@napier.ac.uk

Benjamin studied semiconductor engineering and system design at Liverpool University before moving on to work in the game industry for a number of years. Later, Benjamin returned to education to pursue his PhD at Newcastle University in the area of interactive animation. Currently, Benjamin is a lecturer/programme leader at the School of Computing Science at Edinburgh Napier University in the software engineering and game technology group. His research interests include synthetic systems, cognitive solutions, massively parallel architectures, autonomous models, physics-based simulations and procedural animations.

Course Overview

1 minute: introduction and welcome

Benjamin Kenwright

Overview of the course and motivation for attending it.

20 minutes: visualizing 3D objects with shells

Benjamin Kenwright

This section will cover the basics of shells and how they are used to visualize complex 3D geometry at a fraction of the computational overhead (i.e., bandwidth, memory, and computational speed).

40 minutes: hair and fur practical implementation details

Benjamin Kenwright

This section of the course will cover hair and fur specific details, including:

- thickness
- color, texture
- waviness
- ...

We also demonstrate and show real-time examples of fur/hair simulations for characters and terrain (e.g., rabbit fur, human head, and grass).

35 minutes: movement and interaction

Benjamin Kenwright

This section will cover the details of how we can extend the static hair/fur models to include movement through simplified physics-based models (e.g., verlet-springs).
Several examples will be shown to illustrate the various aspects of the combination of physics and shells for dynamic interactive fur/hair results.

Contents

Chapter 1

Introduction

1.1 Motivation

The simulation of furry surfaces (i.e., hair and fur) is indispensable in various fields, such as graphics and animation. For example, the representation of realistic virtual objects, such as teddy-bears and tigers. Yet even with the tremendous growth in computational power over the past decade (e.g., graphics cards with hundreds of thousands of number crunching cores), you would think realistic hair was common place in virtual environments, such as games? Unfortunately not, since within real-time worlds, such as games, hair is only a small part of the picture and must share the system resources with other components. For example, the rendering of the realistic terrain, the physics, the artificial intelligence, the game-play, and the animation all want a piece of the resource pie (i.e., computational time and memory).

This course addresses a number of crucial questions, such as:

- What methods are there for creating realistic fur and hair?
- Why do we use shells to create fur and hair effects?
- How can we create different types of fur (e.g., wavy, tangled, and wooly)?
- How do we make the fur look more realistic? (e.g., add dynamic lighting and shadows)?
- How can we make the fur and hair move realistically and respond to user interaction in real-time?
- What are the limitations of fur shells?

This course focuses on real-time fur and hair effects for interactive environments, such as games and training simulations. We introduce the reader to the basic concept and go step-by-step using a question and answer approach to extend and accommodate techniques for viable real-world solutions. We demonstrate user created hair and fur simulations (e.g., with textures and pre-defined data) before moving onto **procedural techniques dynamic effects, such as hair with different characteristics (i.e., wavy or straight) moving in real-time while growing longer and thicker in on-the-fly.**

1.1.1 Computational Power

While an average GPU can render millions of triangles, we want more than a single 'Tech Demo'. We do not want to dedicate the entire system's resources to hair and fur. In practice, we would like the hair and fur in our implementation to consume less than 10 % of the total frame-rate. Furthermore, we want the fur and hair to possess 'dynamic' physical properties that the user can interact with to produce realistic movement. For example, responding to external force disturbances by swaying and swooshing, such as wind or active motion (see Figure 1.1).

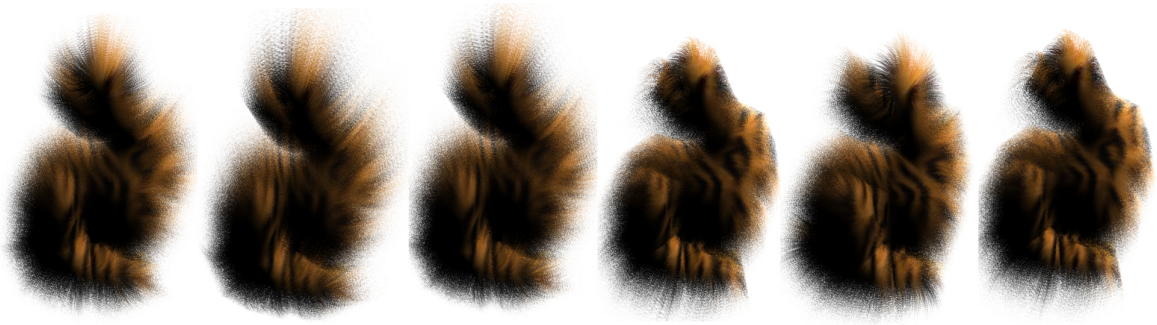


Figure 1.1: **Real-Time Interactive Dynamic Hair** - The user can drag the rabbit model around using the mouse while the hair effect responds accordingly (i.e., swaying and swooshing around).

1.1.2 Food For Thought

An average human has approximately 100,000 strands of hair on their head. But what if our simulation has 20 people? Each moving uniquely. Does their hair all move the same? Does it all look the same?

Technically, a high-end graphics card processor can crunch away on millions of triangles created from splines with lighting that move, but in real-time? Not yet.

For example, let us say we use an uncomplicated brute force approach. Sub-dividing a surface into a 100,000 splines that we subdivide into rough sections and trianglize, as shown in Figure 1.2. We can render 1.8 million triangle easily with today's graphics cards, however, it leaves little room for much else and doesn't allow us to render whole body fur or update it so it moves in real-time.

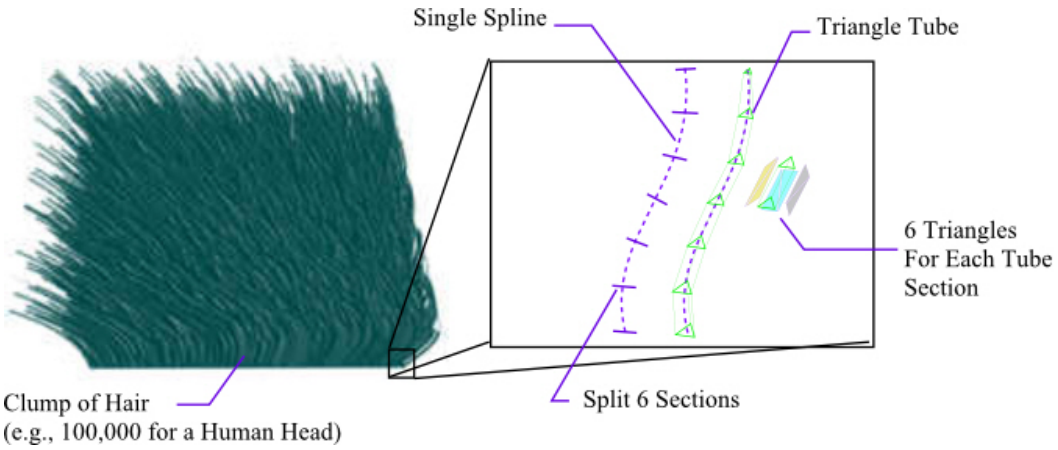


Figure 1.2: **Brute Force Hair-by-Hair Approach** - We can iteratively generate each hair by triangulating the surface of a spline. For example, a simple brute force approach might cost in terms of computational power - 100,000 splines, 6 sub-sections, and 6 triangles for each sub-section. Hence, we get a total triangle count of 100,000x6x3=1.8 million triangles (also see Figure 1.3).

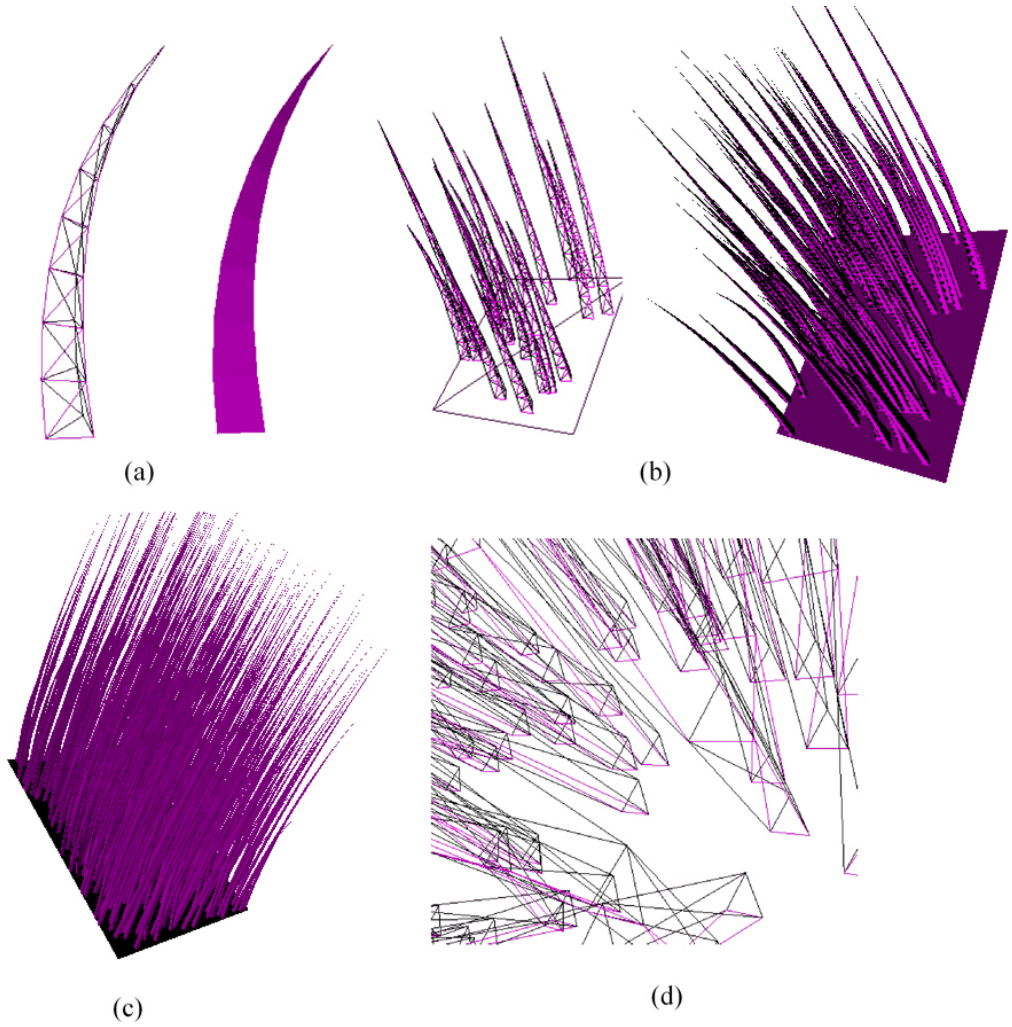


Figure 1.3: **Modelling Individual Hairs** - A simple simulation showing: (a) single hair strand (i.e., six segments), (b) 20 strands (random placement and length), and (c) 100 strands composed of over 14,000 vertices. As the number of hairs get more dense visual artefacts, such as aliasing. Furthermore, the computational overhead of rendering dense numbers hairs, not to mention the memory and bandwidth overhead grows exponentially. While the tessellation of the spline-hairs could be pushed onto completely onto the GPU it is still not a viable real-time solution. However, it should be noted, the method does provide an enormous amount of flexibility and control (e.g., length and thickness) with the ability to zoom in extremely close and get highly realistic individual hair characteristics.

The advantages of modelling individual hair is:

- Highly realistic hair (film quality)
- Flexible and highly detailed
- Can create any imaginable type of hair

The disadvantages include:

- It is primarily for offline processing (not practical for real-time environments, such as games)

- Non-interactive and static
- While it could run in real-time, not for high-fidelity dense amount of hair and fur (crowds of rabbits or bears)

1.2 What Do We Mean By Shells (or Onion Layers)?

A simple analogy to help understand the principle is to imagine a stack of flat transparent glass plates. Then if we draw an identical circle on each glass plate if we look at the stack of plates it produces a visual trick that we are seeing a solid tube, as shown in Figure 1.4. The transparent layers are often referred to as shells and are built up like layers on an onion to produce the visual illusion of a solid object. The shell-based method allows us to visually display very high detail geometry using less system resources (e.g., fewer polygons and memory bandwidth). This method demonstrates excellent real-time results. A transparent texture with noise pixels are scattered across the image and then wrapped across the shells allows us to create a hair like effect. Of course, varying the number of shells and layers shown in Figure 2.1 demonstrates varying realism using a 'fixed' image. Following on from a constant 'fixed' shell, we alter the shell so that the number of 'hair' pixels is reduced as you move further out towards the outer layers. Furthermore, we can start to extend the principle and combine the technique with texture decal information to create textured fur effects (i.e., fur patterns).

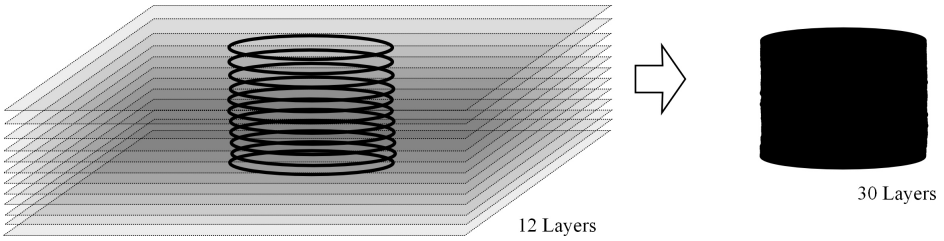


Figure 1.4: **Optical Effect** - Transparent layers allow us to synthesize high detail geometry using low-poly meshes.

1.3 Overview Of Course Material

The first part of the course describes the basics of shells for rendering high detail models, why we use shells their advantages and limitations. We describe how to render simple fur and hair and performance with rendering samples and quality discussion. We discuss how we can create different quality of images and hair types. We then explore making the hair more realistic by including dynamic lighting and shadowing.

The second part moves onto how we can make the hair move in a realistic way so that it is less rigid and static. How can we make the hair move in a natural life-like way and computationally efficiently?

The last part provides details for a number of practical engineering tricks and tips for the reader to explore to make the fur and hair implementation run faster.

1.4 More Information

The inspiration behind this course comes from an early passion for creating highly life-like simulations for interactive virtual worlds with limited resource, such as games. For more information about hair and fur modelling using shells see the reading list at the end of the course notes, or email me with any questions or updates at:

Benjamin Kenwright
b.kenwright@napier.ac.uk

Chapter 2

A Practical Guide to Generating Real-Time Dynamic Fur and Hair using Shells

This course explains how low-poly slices can represent high detailed dynamic geometry. The complex geometry in question is fur and hair and is created using layers also called shells. Textures are mapped onto these shells to represent the cross section of that slice. These textured quads are rendered at relative positions to where the slice was taken. The more slices gives a more detailed visual representation of the model. This method enables us to create fur effects that run in real-time with high visual detail. In this course, we show various enhancements to the basic shell method to generate more exotic, realistic, and dynamic fur effects.

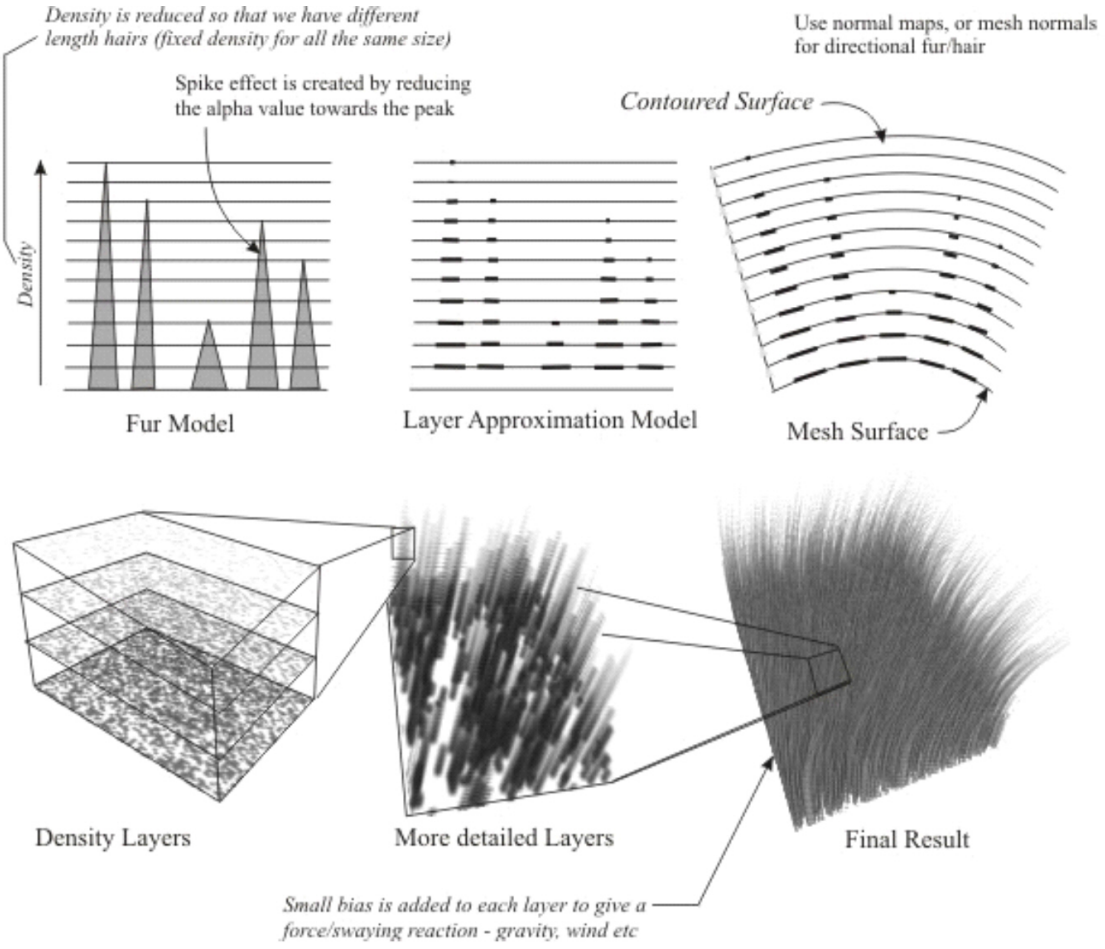


Figure 2.1: **Shells and Hair** - Illustration to show how shells can produce a highly dense looking fur effect.

2.1 Background

2.1.1 CT or CAT-Scan

Shells have long been a popular technique in other fields, such as medicine and data-visualization [10], for representing complex geometry. A popular place many people might have come across shells is the image scanning hardware that takes high resolution pictures of your body using a slicing scheme, as shown in Figure 2.2.

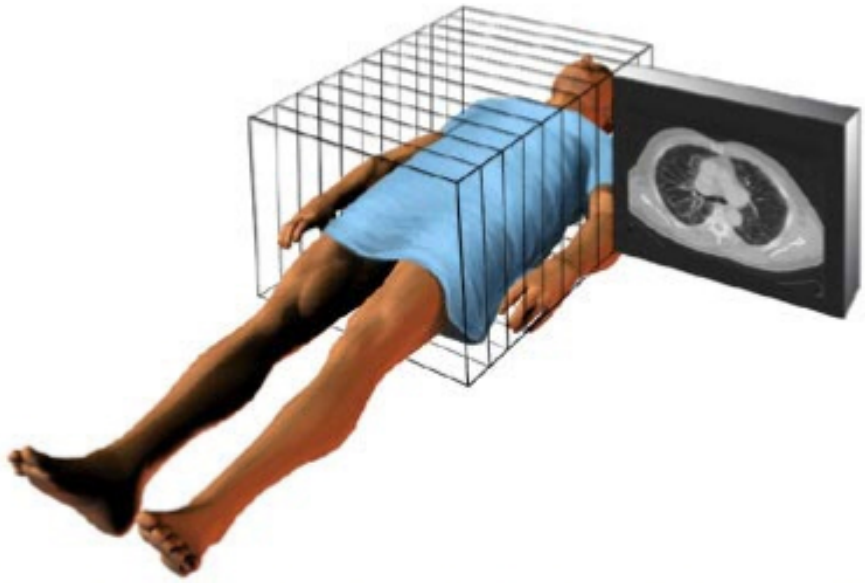


Figure 2.2: **CAT-Scan** - Medical CAT-scan takes multiple slice images to construct a 3D visual model of the internal body.

2.1.2 3D Model Visualization

Procedural algorithms can reproduce infinite resolution 3D models. Rather than reconstructing these models from triangle meshes they can be constructed in shell images.

2.1.3 Grass and Vegetation

A popular method for creating virtual vegetation, such as grass [1], is billboarding. However, when more detail is required, shells offer an alternative solution [4].

2.1.4 Crowds of Standing Characters

Crowds of characters from a distance can also be rendered from a distance. For example, it can be costly to render millions of animated characters, even if each character is a billboard. However, an animated texture slice of millions of characters is cheap, fast, and visually aesthetically pleasing, for large numbers of watching crowds, such as in football matches and racing games.

Figure 2.3: **High Density Stand Crowds** - Even with low-poly 'bill-boarding' techniques it can be expensive and impractical to render huge crowd numbers in virtual games, such as in racing and football games. However, animated textures stretched over shells produces an acceptable life-like solution that is aesthetically pleasing for the gamer.

2.1.5 Other Hair/Fur Engineering Tricks

Hair and fur has been created in games for a long time using billboarding and low-poly meshes. Since triangles are relatively cheap. We can map textures of hair onto the character's low-poly triangle hair mesh. A simple and effect solution. In most cases this is acceptable in games. Furthermore, effects such as bump mapping can add lighting effects to the hair and interconnected triangles by springs can produce slight hair movements to catch the players eye and make the hair seem more dynamic and real.

2.2 Basic Principle

The textures for each shell that make up our fur/hair effect are created using random noise function. The shell quads are created and projected along the normal of the shapes surface.

A few problems exist with the basic method of fixed shells. If too few layers or viewed from side angles the visual effect breaks down. To keep the number of shells at a minimum while producing the most visual realistic effect possible

In Figure 2.8 we vary how the density of random points is reduced as you approach the outer shells. This enables us to gain varying fur types (e.g., clumpy, messy, fine, and thick).

2.3 Basic Implementation (Shaders)

Starting with a very simple demo (i.e., just two files, main.cpp and a fur.fx shader), we create a dynamic texture on-the-fly to generate the fur/hair effect. The source code is available so you can play with it - such as, adjusting the number of layers, fur length, density, bias the tips with force effects and zoom, and so on. A screenshot on the right shows the code running.

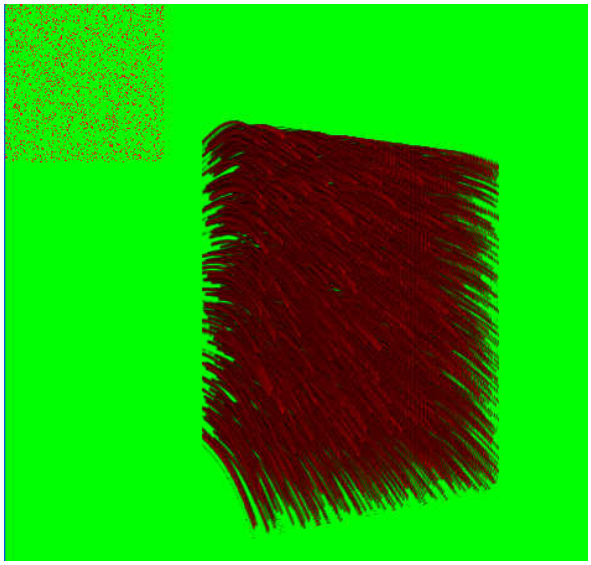


Figure 2.4: **Screen Capture** - From Listing 2.1. The top left of the image shows the generated texture that is mapped onto the shells.

The top left hand corner of the screen is used to show the texture - nothing much as you can see - just a plain texture with noise blobs plotted across the surface. Of course all the surface except the blobs of noise has an alpha value of 0, so its see through.

The demo with practically no effort at all could be expanded to produce a field of grass swaying in the wind! Wow eh? And it doesn't use that much computational power - so you could create a massive field of grass, that looks almost picture perfect in your game and wouldn't cost you an arm and leg in CPU/GPU power.

This is still only a simple model, it's at its bare basics here - as we've not added varying density so that the fur is thicker or thinner as it approaches the tips, also spikes...the hairs are the same thickness all the way up, we can fix that. Then of course, there is texture decals, so your hair is the same color as your textured surface.

And of course there's lighting! The simple demo uses basic diffuse (directional) lighting model...only a few extra lines...which of course you can comment out and see it with and without - but it does not really show the hair shadows. We can add this later by using UV offsets for each layer to create a per hair lighting, sweet stuff eh!

The essential elements of the program source code are given below in Listing 2.1. However, you can download, compile, and experiment with the source code (Download Source Code (11kb)).

Lets take a look at what the main parts of code look like, and see if I can explain what's happening. Below shows the HLSL (High Level Shader Language) Shader which is used to produce the fur effect. We call the shader for each fur layer, and pass it along a parameter FurLength, which is used to determine which layer where at. The single line of code which is most important in the code is:

```
1  float3 P = IN.position.xyz + (IN.normal * FurLength);
```

Where we have our IN.position.xyz input vertices information, and of course IN.normal which is the vertices normal and finally FurLength which is a value from 0 to some max length. So for example, when FurLength is 0, our output vertices data is just the object surface. Then we increment FurLength by some amount for the next layer, e.g. to 1.0, then our next value for P is projected out by some amount producing the next shell or layer...then we increment our layer again and FurLength increases again and we render another shell/layer...eventually producing a fur effect.

Listing 2.1: File: fur.fx. Uncomplicated fur shader

```
1
2  /*********************************************************************************/
3  /*                                                                               */
4  /* File: fur.fx                                                                  */
5  /* b.kenwright@napier.ac.uk                                                      */
6  /*                                                                               */
7  /*********************************************************************************/
8  /*
9     Very basic fur/hair demo showing how to generate realistic looking fur/hair
10    using Shaders.
11 */
12 /*********************************************************************************/
13
14 float FurLength = 0;
15 float UVScale   = 1.0f;
16 float Layer     = 0; // 0 to 1 for the level
17
18 float3 vGravity = float3(0,-2.0,0);
19
20 float4 vecLightDir = float4(0.8,0.8,1,0);
21
22 //-----------------------------------
23 texture FurTexture;
24 sampler TextureSampler = sampler_state;
25 //-----------------------------------
26
27
28 // transformations
29 float4x4 worldViewProj : WORLDVIEWPROJ;
30 float4x4 matWorld : WORLD;
31
32 //-----------------------------------
33 struct vertexInput {
34     float3 position                 : POSITION;
35     float3 normal                   : NORMAL;
36     float4 texCoordDiffuse          : TEXCOORD0;
37 };
38
39 struct vertexOutput {
40     float4 HPOS     : POSITION;
41     float4 T0       : TEXCOORD0; // fur alpha
42     float3 normal   : TEXCOORD1;
43 };
44
45
46 //----------------------------------- ( vs 1.1 )
47 vertexOutput VS_TransformAndTexture(vertexInput IN)
48 {
49     vertexOutput OUT = (vertexOutput)0;
50
51       //** MAIN LINE ** MAIN LINE ** MAIN LINE ** MAIN LINE ** MAIN LINE **//
52       //** MAIN LINE ** MAIN LINE ** MAIN LINE ** MAIN LINE ** MAIN LINE **//
53       //This single line is responsible for creating the layers!  This is it! Nothing
54       //more nothing less!
55       float3 P = IN.position.xyz + (IN.normal * FurLength);
56
57       //Modify our normal so it faces the correct direction for lighting if we
58       //want any lighting
59       float3 normal = normalize(mul(IN.normal, matWorld));
60
61       // Couple of lines to give a swaying effect!
62       // Additional Gravit/Force Code
```

```
63        vGravity = mul(vGravity, matWorld);
64        float k =  pow(Layer, 3);  // We use the pow function, so that only the tips of the hairs bend
65                                   // As layer goes from 0 to 1, so by using pow(..) function is still
66                                   // goes form 0 to 1, but it increases faster! exponentially
67        P = P + vGravity*k;
68        // End Gravity Force Addit Code
69
70
71        OUT.T0 = IN.texCoordDiffuse * UVScale; // Pass long texture data
72        // UVScale??  Well we scale the fur texture alpha coords so this effects the fur thickness
73        // thinness, sort of stretches or shrinks the fur over the object!
74
75        OUT.HPOS = mul(float4(P, 1.0f), worldViewProj); // Output Vertice Position Data
76        OUT.normal = normal; // Output Normal
77      return OUT;
78  }
79
80
81  //---------------------------------- ( ps 1.3 )
82  float4 PS_Textured( vertexOutput IN): COLOR
83  {
84        float4 FurColour = tex2D( TextureSampler,  IN.T0 ); // Fur Texture - alpha is VERY IMPORTANT!
85        float4 FinalColour = FurColour;
86
87        //--------------------------
88
89        //Basic Directional Lighting
90        float4 ambient = {0.3, 0.3, 0.3, 0.0};
91        ambient = ambient * FinalColour;
92        float4 diffuse = FinalColour;
93        FinalColour = ambient + diffuse * dot(vecLightDir, IN.normal);
94        //End Basic Lighting Code
95
96        //--------------------------
97
98        FinalColour.a = FurColour.a;
99        //return FinalColour;      // fur colour only!
100       return FinalColour;       // Use texture colour
101       //return float4(0,0,0,0); // Use for totally invisible!  Can't see
102 }
```

The code should work with even the simplest shaders - and will run with vertex shaders vs1.1 and ps1.3 - but with later shader versions, you can implement loops within the script where by you can move all the fur details to the effect file.

2.4 Initial Results

Runs at real-time frame-rates and is ideal for games. Produce excellent visual effects. Can vary the fur types, e.g., fine, thick, clumpy, and messy. Simple and intuitive, can be combined with textures to create textured fur.

Further work is to add fins and springs between the various layers to give an interactive fur effect. Create grass, tree effects. Alternative noise algorithms to generate shell images compare and investigate.

2.4.1 Onion Shells

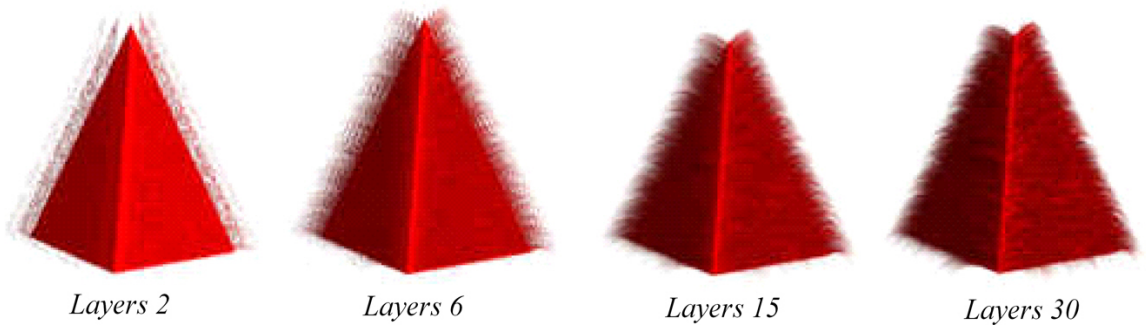


Figure 2.5: **Shell Numbers** - Varying the number of shells.

The shells-based approach has two important features, the direction of the layers (i.e., the normal), and the masking transparent texture on each layer (i.e., the alpha).

2.4.1.1 Normal

Typically, 3D mesh models are exported with normal information. Alternatively, for simple triangle meshes, we can also calculate the normal vector direction using the cross-product of each triangles edge. The vertex normal is used to position each shell (i.e., projecting the vertex in the direction of the normal). The displacement can be linear or non-linear depending upon the hair and fur effect. Of course, the visual quality, highly depends upon the number of shells. A greater number of shells gives a thicker volumetric effect, but at the cost of more computations. Typically, for the simulations in this course use shell numbers in the range of 20 to 60 for the screen captures

2.4.1.2 Alpha

The alpha (i.e., the transparency) of each layer allows us to make hair and fur more realistic. Hair and fur isn't solid. If you pull one of your hairs out of your head and hold it up to the light, you'll see that it allows a small amount of light. We can mimic this effect by giving different hairs varying transparency to more closely emulate real-world hair.

2.5 Random Noise

So how can we create the hair textures? We want to be able to vary the individual hair thickness, varying hair length, sparsity, and style automatically and on-the-fly. The most uncomplicated and simplest approach is to use a random number generator.

2.5.1 Repeatable

Naively, a person might randomly generate solid pixels for each texture layer in turn. However, this would be chaotically disorganized and messy, and resemble nothing that at like hair or fur or anything in fact. Why? Because each of the texture layers would not connect to the next if we did it this way. For example, when we explained the principle shells in Section 1.2, we used the a duplicate circle place at the same location on each layer. However, if the circles were randomly placed at different locations on each layer, we would not see a solid-tube but instead see a gibbering mess.

Alternatively, if we generated random pixel noise for a texture and this same texture on all the layers, we would get perfectly straight hair effect.

2.5.2 Perlin (Coherent Noise)

How can we vary the noise across the layers without it appearing random and disorganized? That is by using the same starting 'seed'. There are different techniques for generating procedural textures, e.g., fractals, Voronoi diagrams, and Worley noise. Each method would give different solutions. However, Perlin noise is probably one of the most popular. The beauty about Perlin noise is that it is not random but turbulent, so where we have a non-deterministic phenomenon it can be applied to give more 'natural' results. Alternatively, there is also the option of loading pre-created textures (e.g., that have been created by an artist) to give the total control over the visual aspect if necessary.

2.5.3 Trigonometric, Exponential, Power

Up until this point we have been projecting the hair along a parallel straight line with the previous layer. We can offset the pixels using a pattern (e.g., sine wave or exponential power) to create more life-like looking hair styles. For example, we know the layer we are on (e.g., 0 to 60 from the base to the tip). We can scale the layer number and input it into a sine function so the hair pixels are wavy. Furthermore, mixing in random component based on the pixel position we can give each hair a seemingly random feel (e.g., direction, length, and color).

2.6 Density

The spacing between the shells does not have to be linear. However, what advantage does it give to modify the distribution of the shells between the base and the tip? This depends on where we want to project the level of detail and the angle of the camera. Essentially, for objects covered in hair, it makes sense to have the outer shells closer together, since they will be the thinnest. For example, the distribution can be scaled algorithmically:

- Linear
- Non-Linear

- Exponential

2.6.1 Texture Resolution

Typically, for the simulations, we use 20 textures of size 128x128. However, over or under sizing the texture produces adverse effects. For example, in Figure 2.6, we show the effects of using a low or high resolution texture. For low resolution textures that are stretched across the model the hairs effect will look like thick poles. Then again, for high resolution textures, the texture is squashed causing adverse visual effects, since the density extremely high and each hair is smaller than a pixel.

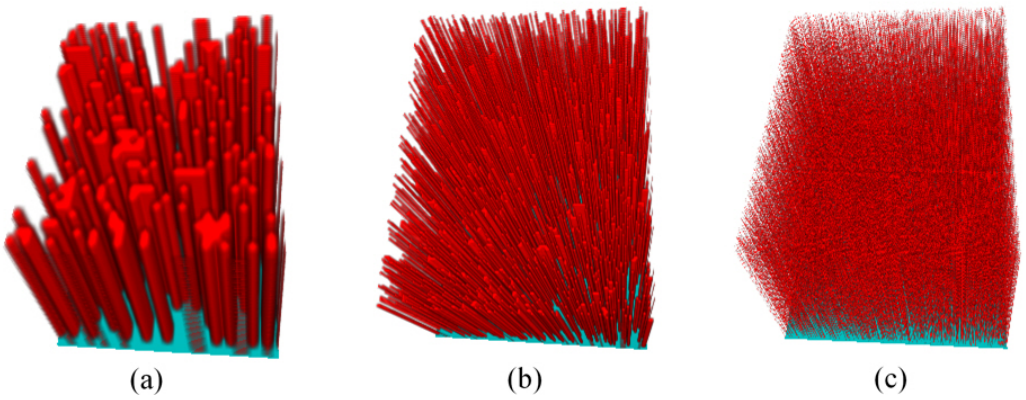


Figure 2.6: **Texture Resolution** - 20 textures of resolution (a) 32x32, (b) 128x128, and (c) 512x512, spread across 60 shells.

2.7 Thickness with Length

Each individual hair is rarely the same length or thickness. Furthermore, each hair does not go in a perfectly straight line. Each hair possesses small random path deviations. There are even the occasions when the hair is just mad. For example, we have all had bad hair days. With hair everywhere. Hence, emulating this random uniqueness makes is important for realism. But how can we do it?

Generating variable thickness and length is a walk in the park on the GPU. We know the position of the shell in relation to the base. We also know the place the pixel will be drawn. Then the radius and length of each individual hair pixel can be mixed with a scaling function (e.g., fixed, linear, ro exponential calculation).

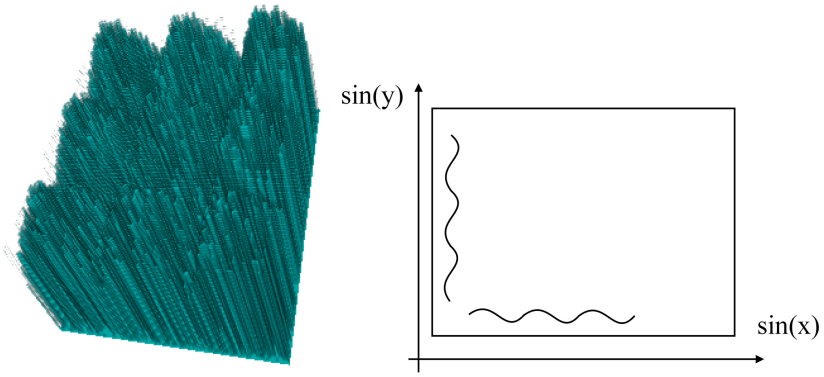


Figure 2.7: **Hair Length Control** - For artistic effects we can create custom hair lengths (e.g., sinusoidal mountains). In the majority of realistic cases we blend the hair length randomly, however, the option of custom control is there.

2.8 Hair Type

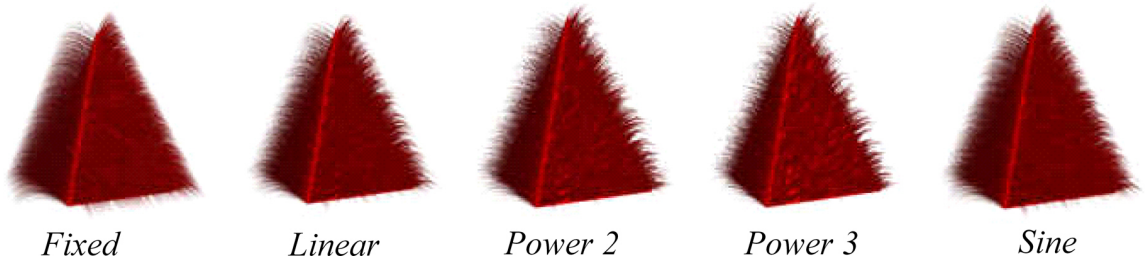


Figure 2.8: **Shell Patters** - Different models for the fur pattern across the layers.

2.8.1 Straight

2.8.2 Clumpy

2.8.3 Wavy

2.8.4 Zigzag (Electric-Shock Hair)

2.9 Color

2.9.1 Fixed Color

Have you ever looked at fur/hair closely? I mean really closely? Then you will notice that all the fur/hairs are not all exactly the same colour. Hence, by introducing an extra 3 lines of code into the basic color shader code, we can generate fur with a slight color randomness. Note that the randomness is only *slight*, but gives an improved life-like realism.

```
// hair color
// Slightly random plus or minus small amount fixed colour
D3DXCOLOR col;
col.r = rnd(0.9, 0.9);  // ie. 0.9, 0.3, 0.3 is brown (rgb)
col.g = rnd(0.3, 0.5);
col.b = rnd(0.3, 0.7);
col.a = 1.0f;
```

The few lines of code above gives each of the hairs their own individuality. Remember though, that each hair must keep its same colour across its different layers, which is the reason why we keep setting the seed value for the random function. The rnd(..) is a random function value, which generates a float random value from min to max value (as shown below):

```
const float INV_RAND_MAX = 1.0 / (RAND_MAX + 1);
inline float rnd(float max=1.0) { return max * INV_RAND_MAX * rand(); }
inline float rnd(float min, float max) { return min + (max - min) * INV_RAND_MAX * rand();↩
     }
```

2.9.1.1 White Hairs

On occasion, we might want to inject the odd white hair to show age. For example, when simulating an old man or woman's head of hair.

2.9.2 Texture

As shown in Figure 2.9 we can create more nature and colorful looking hair effects by mixing the transparent layers with decal information from textures.

Figure 2.9: **Mixing Transparent Layers with Texture Decals** - We can create more life-like looking fur and hair effects by mixing in color information from texture files.

2.10 Shadows

Have you ever noticed that each hair has a shadow? Maybe you have never thought about it. However, inter-hair shadows tare essential for details and realism. While we can vary the alpha value for each hair from base to tip, so individual overlapping hairs stand. However, a constant set of hairs with a fixed alpha produces a sort of blob (i.e., the hair effect becomes a constant blur of color and does not look like hair). Shadows really add emphasis to the individual hairs (see Figure 2.10).

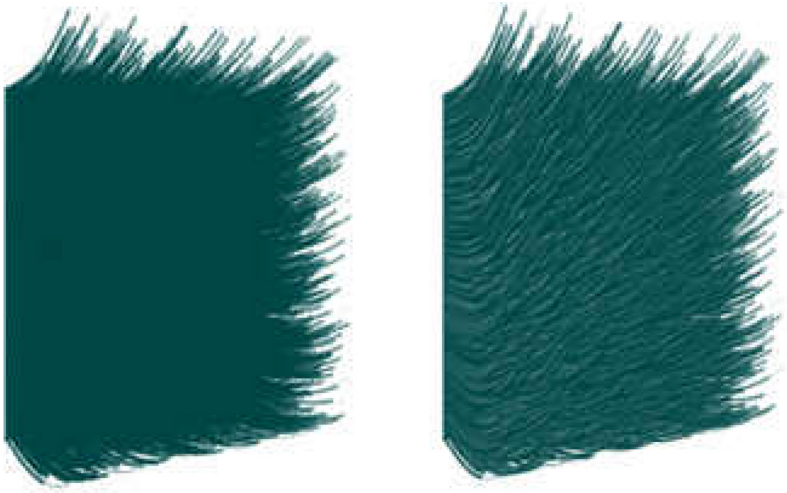


Figure 2.10: **Do we need inter-fur shadowing?** - From the figure, it is easy to see why inter-fur shadowing is important. Firstly, it allows us to see individual hairs and secondly in scenes with dynamic lighting the hair shadows move as would expect and fits in with the surroundings.

2.10.1 Inter-Fur Shadowing

So how can we add shadows to hair when all we have is shells? We achieve this inter-fur shadowing using an uncomplicated trick. The principle is simple, but at the cost of having to render each fur layer an additional time. It works by taking each layer and offsetting the uv coordinates very slightly, using the surface normal as a bias, so that each layers dots or hairs are offset. Furthermore, rather than just render the same color throughout - as we are trying to generate a shadow effect, we convert the rgb offset value to a grey shadow. This grey offset is rendered underneath each layer, so that as the layers are built up as the original layers for the fur, we get a shadowing effect, which makes the individual hairs stand out more.

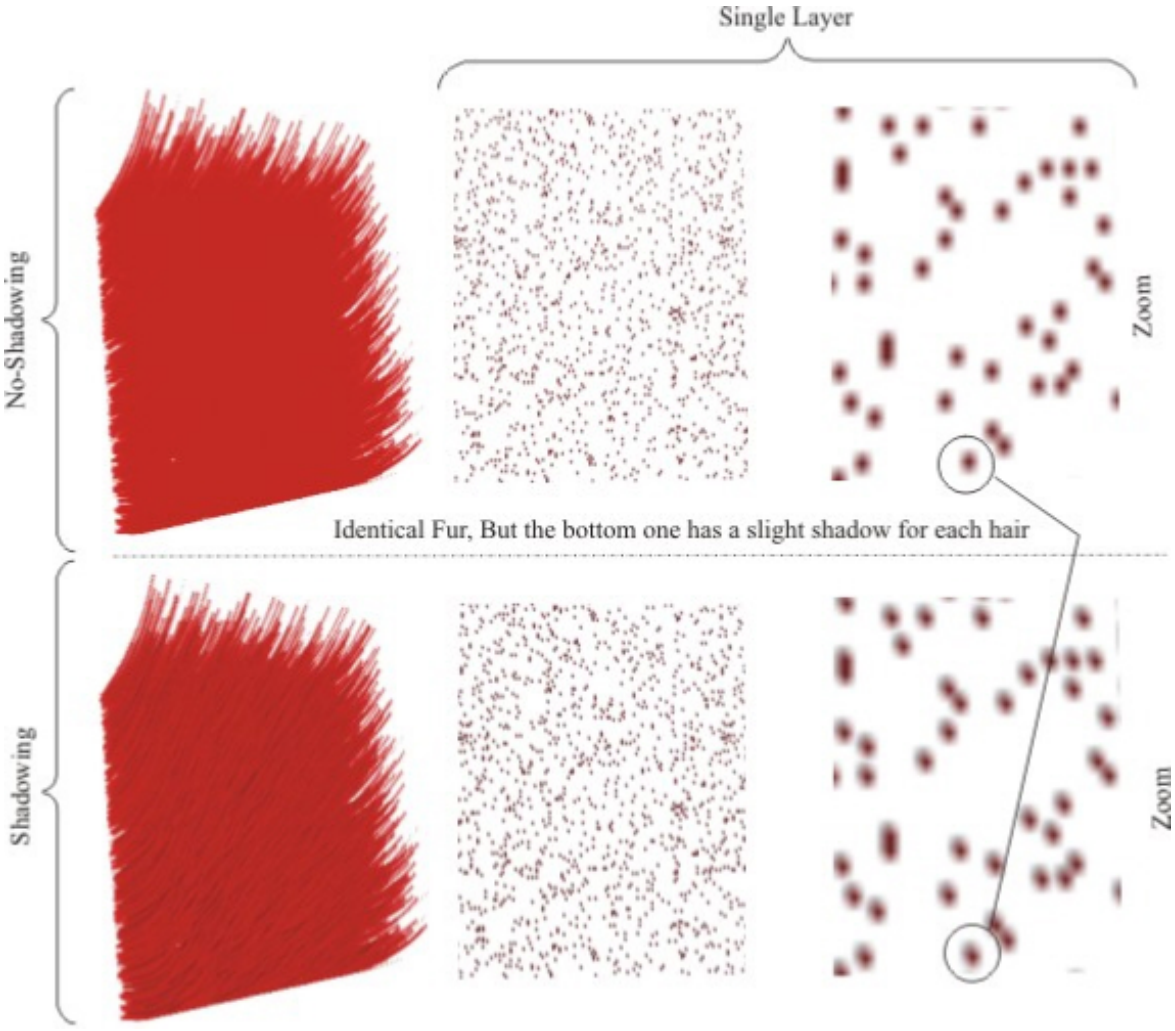


Figure 2.11: **How shadowing looks on each layer?** - Adding a small bias shadow offset to each hair shell allows us to quickly and easily add a inter-fur effect.

We illustrate the effect of inter-fur shadowing in Figure 2.11. While the individual shell textures do little to emphasis the effect, it is more apparent when you look at the overall hair effect. While a grey value is used for the shadow effect, you an make it more dramatic by replacing the shadow color with pure black (i.e. float4(1,1,1, fcolor.a)) in the last line of the shader file. This gives the hairs a really strong shadow outline. Then again, this makes the shadow effect look less realistic, but you can experiment with different methods to create the visual effect you desire.

Listing 2.2: Inter-Fur Shadow Effect

```
Inter-Fur Shadows Effect
....

vertexOutput VS_Shadow_TransformAndTexture(vertexInput IN)
{
    vertexOutput OUT = (vertexOutput)0;

      float3 P = IN.position.xyz + (IN.normal * FurLength);
      float4 normal = mul(IN.normal, matWorld);

```

```
11      //---Additional Gravit/Force Code----
12      vGravity = mul(vGravity, matWorld);
13      float k =  pow(Layer, 3);  // We use the pow function, so that only the tips of the ↩
    hairs bend
14                                 // As layer goes from 0 to 1, so by using pow(..) ↩
    function is still
15                                 // goes form 0 to 1, but it increases faster! ↩
    exponentially
16      P = P + vGravity*k;
17      //---End Gravity Force Addit Code----
18
19      // We want the fur in the center of the object, as where doing offsets!  So if we ↩
    just
20      // use the normal, when it's facing in the z direction, the center of the object ↩
    mesh,
21      // then it will be the same when we scale it larger.  So we modify the normal so we
22      // have larger values in the center and less on the edge for use in our offset inter
23      // fur shading!  If thats makes any sense?...not the best explanation I've ever done↩
     :(
24      float4 znormal = 1 - dot(normal, float4(0,0,1,0));
25      // Works pretty well though, even if you just do this below, and just use the normal↩
    !
26      //float4 znormal = normal;
27
28
29      OUT.T0 = IN.texCoordDiffuse * UVScale;
30      OUT.T1 = IN.texCoordDiffuse * UVScale + znormal * 0.0011;
31      // UVScale??  We only multiply the UVScale by the T0 & T1 as this is our Fur Alpha ↩
    value!, hence
32      // scaling this value scales the fur across your object...so reducing it makes the ↩
    fur thicker,
33      // increasing it makes it thinner.  We don't do it to our T2 output as this is our ↩
    texture
34      // coords for our texture...and we don't want to effect this
35
36      OUT.HPOS = mul(float4(P, 1.0f), worldViewProj);
37    OUT.normal = normal;
38    return OUT;
39  }
40
41  //-------------------------------------------------------------------------------//
42
43  float4 PS_Shadow_Textured( vertexOutput IN ): COLOR
44  {
45      float4 furcolr        =  tex2D( TextureSampler, IN.T0 );
46      float4 furcolr_offset =  tex2D( TextureSampler, IN.T1 );
47
48      //??We use a simple offset trick to give individual hair shadows.  Works by using
49      //the normal  - furcolor_offset in the direction of the normal of the triangle
50      //face.
51      //Of course we scale this by a value so our offset is only small, but just
52      //enough to give some individual hair lighting
53    //--------------------------
54      float4 color = furcolr_offset - furcolr;
55
56      float4 fcolor = color;
57      fcolor.a = color.a;
58      //--------------------------
59
60      //??We have our offset colour - but of course our fur colour could be a single
61      // colour, red or just green!  So we want this as a grey, as we are concerned
62      // with the fur shadows!
63      //--------------------------
64      // From RGB to YUV
65      // Y = 0.299R + 0.587G + 0.114B
66      // U = 0.492 (B-Y)
67
68      // From YUV to RGB
69      // R = Y + 1.140V
70      // G = Y - 0.395U - 0.581V
71      // B = Y + 2.032U
72
73      // Y is the luma, and contains most of the information of the image
74      float4 Y    = float4(0.299, 0.587, 0.114, 0.0f);
75      fcolor  = dot(Y, fcolor); // grey output
76      //--------------------------
77
78      return fcolor;
79
```

```
80          //return float4(1.0f, 1.0f, 1.0f, 0.3f); //rrggbbaa
81  }
82
83  ....
```

2.10.2 Dynamic Shadows

The shadow offset for each hair can be calculated based on a global light source (e.g., distant sun). Hence, as the light source moves, such as travelling across the sky, the shadows would automatically correct themselves. A simple example, would be to place the hair shadow in the direction from the hair to the light.

2.11 Fins

A serious limitation of shell based methods is that when you look at them from the side (e.g., as you get closer to 90 degrees), the illusion is broken. You can see that the fur and hair is composed of shells. We can reduce this by adding *fins*. Fins are essentially quads that stick up along the triangle edges to mask the illusion. Then there is the question of a *single* fin per edge or *multiple* fins per edge. Why would a multiple fins on each layer? For static non-moving shells a single fixed fin spread across all the shells provides an adequate solution. However, as we show later, as we start to squash and distort the projected shells, we also need to squash and distort the fins. If performance is a real-issue, we could sacrifice partial visual quality a single fin and the outer most layer vertices to scale move the fin ends. However, we can also create multiple fins - essentially, a single fin that is sliced in correlation with the layers, so each layer distorts a portion of the fin.

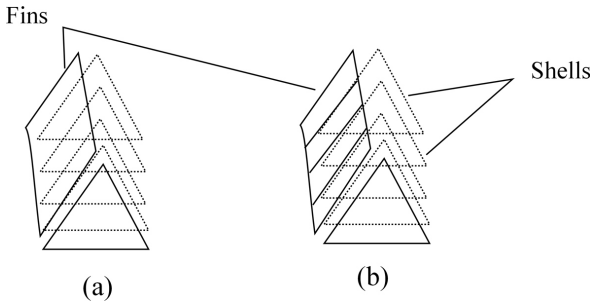


Figure 2.12: **Fins** - (a) single fin spread across all the shells, and (b) a single fin spread across all the shells but with multiple parts that correlate to the shell layer.

2.12 Embedding Information in Textures

Since the actual fur and hair comes from the texture we can embed additional information within the texture. For example, which parts of the model have hair and which do-not, with different parts of the model having different types of fur or hair.

2.13 Movement

While the majority of papers and research focus on making the hair and fur appear life-like and realistic, we go a bit further by also making the hair and fur move in a realistic life-like manner. We explore techniques that will not sacrifice the inherent advantages of the shell based approach (i.e., its computational speed and simplicity)

2.13.1 Procedural

There are a variety of ways to create movement. For example, this includes:

- Trigonometric - uncomplicated trigonometric functions, such as sin and cos, produce smooth coherent solutions. For example, the rhythmic motion for swaying in the swing
- Physics-Based - (i.e., force-controlled) - giving each layer a physical quantity allows us to synthesize unpredictable external forces from the user (e.g., swaying and tilting due to air forces)

2.14 Physics

Incorporating a simple physics-based model into the fur and hair simulation allows us to create a flexible *interactive* solution. For example, *invisible* external forces, such as wind, can shown in hair movement.

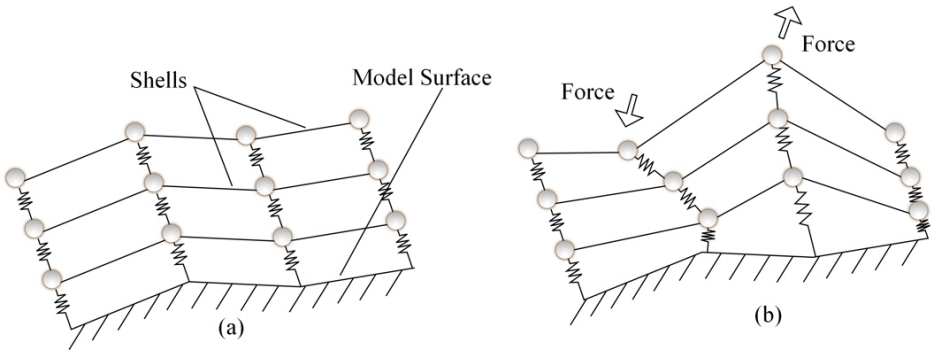


Figure 2.13: **Shell Spring Topology** - Interconnecting the shells using a mass-spring system allows us to create dynamic and responsive fur and hair effects.

2.14.1 Verlet-Integration

Forces, such as gravity and wind, take the static fur to an interactive and dynamic level. Movement can really bring a scene to life. While there are different constraint-based methods for enforcing the shell distances, we exploit the verlet-based integration scheme because it is simple, computational fast, and numerical stable.

With a simple biasing, Newton laws for force ($F = ma$) and, of course, Hook's spring law ($F = -kx$), we can add dynamic motion to the visual fur effect. For example, if we drag our simple hairy bunny model around, the fur with sway and bounce. While the effect is not perfect, the results is visually life-like and responsive. Go crazy, and swishing the bunny back and forth so you can really see the dynamic fur effect.

The biasing of the fur is done by adding to the position of each fur layer. To make the effect non-linear, we add a slight adjustment (i.e., power of 3 bias), so the outer layers are effected more than the inner layers. Giving a curving effect on the hair tips.

You can increase and decrease the fur length by pressing ¡Up¿ and ¡Down¿ keys, so that the forces are really notable. Of course, the code could do with a bit more tweaking so it is a bit more bouncy, but it works and is aesthetically pleasing.

Below shows the code snippet for the shader that adds the dynamic aspect to the fur (i.e., forces and movement). For practice, you can comment the shader code out with comments (i.e., /* */) and it will still work (but not move). For example, you might like to put the fur force code into a separate function later on, so your implementation is more structured. However, for the example, it is more readable and easier to follow if the few extra lines are kept together.

```
1  Code Snippet From Fur.fx
2  ...
3        // Additional Gravit/Force Code
4        vGravity = mul(vGravity, matWorld);
5        //float pp = 3.14 * 0.5 * Layer; // depth paramete
6        //float A = dot(normal, vGravity);  //A is the angle between surface normal and ↩
       gravity vector
7        float k =   pow(Layer, 3); // We use the pow function, so that only the tips of the ↩
       hairs bend
8                                   // As layer goes from 0 to 1, so by using pow(..) ↩
       function is still
9                                   // goes form 0 to 1, but it increases faster! ↩
       exponentially
10       P = P + vGravity*k;
11       // End Gravity Force Addit Code
12 ...
```

Remember, the shader code is repeated in two parts, the fur rendering part and, of course, the shadow part. The code is repeated so our inter-hair shadows correlate with the fur (i.e., they are in the same place).

The solution of spring-mass shells does not mean the end. We also have to consider:

- Multiple Inter-Shell Springs (i.e., the configuration, such as criss-crossing)
- Links
  - Hierarchical Links (i.e., level of details, smaller and smaller)
- Collisions
  - Human-Head
  - Sphere-Sphere
  - Hair Dryer Demo
- Long Hair with Links
- Dynamic Growing (i.e., getting longer with time) - imaging a game character having their hair

grow as the game progressed
- Fury Cube Falling Demo - Furry and hairy objects are just more interesting and fun
- Inter-fur Collisions

2.15 Performance

2.15.1 Number of Textured Triangles

High detail geometry meshes can be unsuitable for the onion shells, since we need to render the geometry multiple times to build up the surface hair volume. Hence, it might be more suitable to use a lower dimensional mesh (e.g, the physics-mesh) for the fur and hair effect.

One major factor is bandwidth. For each frame of the animation, we perform a simple vertex position update calculation on the shader. However, for large numbers of shells with unique textures the texture memory overhead can become expensive. Although, we can use lower-resolution textures and scale them upwards, furthermore, we can share similar textures across multiple shells (i.e., less unique textures).

In some cases it s tempting to push all the dynamic fur operations onto the shader (e.g., blank textures and randomly generate the fur and hair effect), but doing this is not always optimal in terms of kernal overhead. For instance, in the majority of the time the textures are fixed and do not need updating, hence generating them on-the-fly purely on the GPU would less memory expensive but computationally more expensive.

2.15.2 Shells Occluded or Out-of-View

We did not consider any traditional geometry rendering improvement techniques (i.e., do not draw what you cannot see). For example, solid objects, such as characters and planets, have a large majority of shells occluded at any time (e.g., hidden around the back of the solid object). Hence, while it might be extra work keeping track which triangle are visible, it can dramatically enhance performance since you are updating less geometry.

2.16 Limitations

The shell-based approach presented in this course has a number of limitations. The motion is applied across layers and not individual hairs. Furthermore, we only focused on local inter-fur shadowing, and not global shadows (e.g., when you have a valley of hair the overhang hair casts a shadow). However, the approach is ideal for interactive environments, and runs at better than real-time frame-rates while providing an approximate high-fidelity visual hair effect.

2.17 Improvements / Further Work

In this course, we hope to have demonstrated that shell-based fur and hair animation techniques are a valuable tool for interactive environments, such as games. We have provided the basic building blocks needed to start developing a practical implementation. However, this is by no means the end of the adventure for hair and fur effects. For example, some areas of exploration are:

- Hybrid Shells - Mixing mesh tubes with shells - so you get additional control (i.e., splitting the shells into smaller individual movable parts)
- Fur texture is averaged over the current mesh, using the original fur color texture - calculate each triangles size and exact tu/tv values for the fur texture individually.
- Optimise animation loop (exploit GPU)
- Organise the shader code more to use functions
- Do some performance work triangles vs FPS.
- Camera angle
- Multiple orientated shells (e.g., shells that face towards the camera)

Further Reading

Fur and hair is an important effect for real-time environments, hence, there are large number of interesting articles on-line that we would recommend to the reader. For example, a few recommendations are:

- Online Article (xbdev) [3].
- Fur Shading and Modification based on Cone
- Step Mapping [5].
- NVIDA's white paper [7].
- Simulating Weathering of Fur [2].
- Curling and Clumping of Fur [9].
- Real-Time Fur Rendering [6].
- Lecture notes on Real-Time Rendering of Fur from Gary Sheppard [8].

Bibliography

[1] Brook Bakay, Paul Lalonde, and Wolfgang Heidrich. Real-time animated grass. In *Eurographics 2002*, 2002.

[2] Shaohui Jiao and Enhua Wu. Simulation of weathering fur. In *Proceedings of the 8th International Conference on Virtual Reality Continuum and its Applications in Industry*, pages 35–40. ACM, 2009.

[3] Ben Kenwright. Online article: Generating fur in directx or opengl easily. *http://www.xbdev.net/directx3dx/ specialX/Fur/index.php*, 2002.

[4] Ben Kenwright, Rich Davison, and Graham Morgan. Real-time deformable soft-body simulation using distributed mass-spring approximations. In *CONTENT 2011, The Third International Conference on Creative Content Technologies*, pages 29–33, 2011.

[5] Tom Kühnert and Guido Brunnett. Fur shading and modification based on cone step mapping. In *Computer Graphics Forum*, volume 31, pages 2011–2018. Wiley Online Library, 2012.

[6] Jun Lee, DongKyum Kim, HyungSeok Kim, Carloa Henzel, Jee-In Kim, and MinGyu Lim. Real-time fur simulation and rendering. *Computer Animation and Virtual Worlds*, 21(3-4):311–320, 2010.

[7] NVIDA. White paper, fur (using shells and fins). *White Paper*, Feb 2007.

[8] Gary Sheppard. Real-time rendering of fur. *Lecture Notes*, 2004.

[9] Paulo Silva, Yosuke Bando, Bing-Yu Chen, and Tomoyuki Nishita. Curling and clumping fur represented by texture layers. *The Visual Computer*, 26(6-8):659–667, 2010.

[10] KangKang Yin, Kevin Loken, and Michiel van de Panne. Simbicon: simple biped locomotion control. In *Proceedings of the ACM Transactions on Graphics (TOG)*, volume 26:3, Article 105, pages 1–10, 2007.