

Daniel Hulme

Fur Synthesis for Real-World Ray-Traced Graphics

Computer Science Tripos

Pembroke College

2005

Daniel Hulme, Pembroke College

Fur Synthesis for Real-World Ray-Traced Graphics

Computer Science Tripos, 2005

10080 words

Originator: Daniel Hulme

Supervisor: Dr. N. Dodgson

Aims: Fur synthesis is now widespread in film effects and, increasingly, video games, but most useful systems to date have been based on polygon scan-conversion despite the well-known disadvantages of this approach. This dissertation presents a novel method for ray-tracing fur which combines previous research into microsurface representation and soft hypertexture. An open-source implementation of this technique, in the form of an extensive modification to an existing, popular ray-tracer, is described.

Work completed: I have devised such an algorithm and have created a working realisation of same.

Special difficulties: None.

I Daniel Hulme of Pembroke College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date: May 16, 2005

Contents

1	Introduction	5
1.1	Previous Research	5
1.2	Long hair	6
1.3	Terminology	6
2	Preparation	7
2.1	About POV-Ray	7
2.2	Animation	8
2.3	Requirements specification	8
2.3.1	SDL modification	9
2.3.2	Data structures	10
2.3.3	Pelt intersection	10
2.3.4	Transformation into texture space	10
2.3.5	Density model	11
2.3.6	Lighting model	12
2.4	Project planning	12
2.4.1	Software engineering strategy	12
2.4.2	Choice of tools	12
2.4.3	Source control and backup provision	13
3	Implementation	14
3.1	Parsing	14
3.2	Bounding	15
3.3	Projection	15
3.4	Pelt intersection	16
3.5	Sampling	18
3.5.1	Initialization	18
3.5.2	Sample size	19
3.5.3	Iteration	19
3.5.4	Projection into texture space	19
3.5.5	Normal perturbation	20
3.5.6	Density and shading	20
3.5.7	Exit	20
3.6	Density	21

3.7	Textures and Lighting	21
3.8	Modularity and Other Hypertexture Phenomena	21
4	Evaluation	23
4.1	Testing	23
4.1.1	Specification	23
4.1.2	Test Suite	23
4.1.3	Regression testing	24
4.2	Comparative Evaluation	24
4.2.1	Media fur patch	24
4.2.2	Blender	25
5	Conclusions	27
5.1	Further Work	27
5.2	Conclusion	28
A	Pseudocode listings	29
A.1	Simulate_Hyper	29
A.2	Simulate_Hyper_Shadow	31

Chapter 1

Introduction

In this project I implement a novel method for ray-tracing fur. Writing a fast and feature-packed ray-tracer is involved enough to stand alone as a project, so I have chosen to use an existing ray-tracer as the basis of my project, to allow me to focus my efforts on fur in particular.

The Persistence of Vision ray-tracer (POV-Ray) is popular, mature, and free. It offers support for photon mapping, radiosity, and participating media, and uses a human-readable scene-description language (SDL). It lacks support for fur synthesis. I have produced an extensive modification to POV-Ray, which demonstrates a novel combination of texels and procedural hypertextures, representing fur as procedural microspheres extending beyond the surface of the object. I name the resulting program KEV-Ray, after my cat Kevin.

1.1 Previous Research

The two most important papers on fur synthesis are [PH89] and [KK89].

In [PH89], Perlin and Hoffert describe an extension to the idea of an object's surface, that is, a hard boundary separating the inside from the outside, instead adding a 'soft region': a region of space where the density of the object varies smoothly from one to zero. This soft region is procedurally generated from a mathematical description, then treated as volumetric data and rendered with the marching cubes algorithm. As an application, the paper describes a formula for generating soft regions which look like fur.

In [KK89], Kajiya and Kay describe an extension to the idea of texture mapping by using a 'texel': a cube which is mapped into world space with its base attached to the surface of an object. Instead of volumetric density, the texel contains a model for the density of microspheres, and a bi-directional reflectance distribution function (BRDF), which together describe how light hitting the volume is reflected, the same way that a surface texture describes how light hitting the area is reflected. These texels are tiled over the desired object, and rendered by tracing rays through each texel individually. As an

application, the paper describes a method for generating texel data for a furry object.

These two approaches have been regarded as alternatives, and papers since then have tended to cite both but disregard one ([Bid95] is a good example of this). More recently, research has focused on fur for scan-conversion, this being the dominant technique for consumer graphics and computer-generated film and TV effects. This work has become available to artists in the form of plugins for popular graphics packages like Maya, Lightwave, and Blender, and has enabled fur to be used to great effect in films like *Monsters Inc.*, *Shrek*, and *Garfield*. These systems have tended to represent fur as a combination of 2D shading and explicit geometry, with a complex visibility and level-of-detail system deciding when each strand of fur needs to be represented as polygons.

This research effort has culminated so far in [LPFH01], which describes a system for sampling slices of fur into partially transparent images. These are tiled in a series of concentric ‘shells’ around and ‘fins’ perpendicular to the surface of an object and rendered in the same way as other polygons. This means that a scan-conversion renderer can render fur in real-time on consumer graphics hardware; this system has gone into the Microsoft Xbox.

However, ray-tracing is experiencing a revival at the moment, as the new effects demanded of computer graphics are increasingly hard to do in scan-conversion engines. I hope my new method allows ray-traced fur to catch up with scan-converted fur.

1.2 Long hair

Long hair, of the kind found growing on human heads, is not covered under fur synthesis. Long hair behaves very differently and none of the assumptions used for short fur hold. Also, it is easier to represent long hair as explicit geometry, because it looks good with far fewer strands than are needed to make fur look at all reasonable.

1.3 Terminology

In this document I will avoid using the words ‘hair’ and ‘fur’ as they confuse matters. A ‘strand’ is a single, short hair, such as might form the fur coat of an animal. When many strands cover the surface of an object, they form a ‘pelt’. I will only use the word ‘fur’ when this distinction is not necessary.

A ‘primitive’ is a type of geometric object (e.g. sphere, height field). I will use ‘object’ for an instance of a primitive in a 3D scene, rather than in the programming sense. I will use ‘intersection’ usually for the point or points at which a ray crosses the surface of an object, rather than in the set-theoretic sense.

I will assume that any vector referred to as a direction, tangent, or normal, is normalized.

Chapter 2

Preparation

In this chapter I will describe POV-Ray, the foundation on which the project is built, and will break down the problem.

2.1 About POV-Ray

Although now written in C++, POV-Ray was originally a C program so makes little use of the enhanced memory management and object-orientation of C++. A primitive is defined by a struct containing pointers to functions to intersect with it, get the normal at a particular point, etc. A primitive also has a function to parse the SDL describing an object.

The first stage in running POV-Ray is parsing the scene file. This builds several data structures in memory describing the scene. After this, any preprocessing stages for gathering radiosity data or photon mapping are performed. These data are optionally saved to another file for use in subsequent runs of the program. Following this, the actual tracing starts.

A function specific to the camera type¹ runs a loop, on each iteration creating a new Ray and tracing it through the scene. This tracing function attempts to find the object the ray intersects with, and on doing so calls `DetermineApparentColour`, giving the `Intersection` as an argument. `DetermineApparentColour` uses the texture of the object and a lighting model to find the colour at the intersection, spawning and tracing shadow, reflection, and refraction rays as appropriate. Reflection and refraction rays are treated the same way as primary rays (rays that have come straight from the camera). However, shadow rays (fired from the intersection to lights to determine whether the intersection is in shadow) are traced by a different function, `TestShadow`. This function uses `filter_shadow_ray` instead of `DetermineApparentColour`.

¹POV-Ray offers a variety of cameras, including the usual perspective and orthographic cameras as well as more exotic ones such as spherical fisheye and one for generating animations suitable for Omnimax cinemas. As each must distribute the primary rays differently, each has its own function to spawn and trace primary rays.

`filter_shadow_ray` determines only transparency rather than colour and does not spawn additional rays. Testing whether the current ray is vision or shadow and branching repeatedly is inefficient. Having two functions avoids this cost. This separation motivates the hypertexture implementation (see section 3.5).

As part of this loop, supersampling is optionally performed by spawning additional primary rays and blending the colours returned. The colour for each pixel is clipped to the range $[0, 1]$ and written to the output file in the appropriate format.

2.2 Animation

POV-Ray offers limited support for animation. Using configuration files or command-line parameters, the user can specify that an animation of the specified length be created. A special ‘clock’ variable is declared and can be referred to from the scene file: its value increases for each frame of the animation. POV-Ray runs the whole rendering loop completely for each frame, incrementing the value of ‘clock’. As output it generates a sequence of image files which can be processed by other software to turn them into a video file.

This simplistic approach to animation is quite powerful and complex animations can be controlled by macros using the ‘clock’ variable. Another advantage is that it makes parallel execution very simple², and each frame of the animation can be rendered by different machines without any communication other than the initial scheduling of frames to machines and the final assembly of results. The advantage from my point of view is that it is quite easy to animate the fur. Any parameter can be made to vary over time without code modification because the data structures are recreated from the scene file for each frame.

There is an important issue for me to consider, though. Anything which uses random number generation or other non-deterministic behaviour will not work because the numbers will be different for each frame, causing static objects to change over time. Thus all functions must be carefully designed so that they can exhibit pseudorandom behaviour where required, but deterministically. Even the SDL is designed to allow the user to have several pseudorandom number generators (PRNGs), each of which is deterministic and always gives the same sequence regardless of other PRNGs. This behaviour is desirable even for still images, as it means that rendering the same file will always generate the same image, even on different machines.³

2.3 Requirements specification

Before coding begins, it is vital to specify what is required of the project.

²POV-Ray itself is single-threaded and does not use application-level parallelism, but there are several modified versions specialised for parallel or distributed operation.

³This obviously fails to hold if any of the file I/O directives are used to read data from a changing file, or if the few truly random features left for backwards-compatibility with very old versions are used.

2.3.1 SDL modification

The most obvious requirement is that the user must be able to use the SDL to specify parameters for a pelt attached to any particular object. It is desirable to make this easy and consistent with existing syntax. In POV-Ray, each object has a 'material'. The material consists of an 'interior' and some number of 'texture' layers. Using multiple texture layers can be used for effects such as having a metal texture layer with a patchy rust layer over the top. The interior describes any participating media the object may contain, and parameters like the index of refraction. Each texture consists of a 'pigment', which describes the colour of the object, a 'normal', used for bump-mapping, and a 'finish', which describes surface properties like specular highlights, iridescence, and reflection. Both the SDL and the internal data structures reflect this hierarchy, but it is possible to specify properties without reference to the full hierarchy. So a green box might be described by the code:

```
box {
  <-1,-1,-1>, <1,1,1> // the box's corners
  pigment {
    colour <0,1,0> // standard red, green, blue order
  }
}
```

To be consistent with this and hence be user-friendly, I decided that the new hypertexture block should look like this:

```
box {
  <-1,-1,-1>, <1,1,1>
  pigment {
    colour <0,1,0>
  }
  hyper {
    1.0 // gives the length of the fur
    pigment {
      colour <1,0,0>
    }
  }
}
```

In the texture hierarchy, the hypertexture stands on the same level as the texture and interior. This means that each object can have only one hypertexture that extends over the whole object. This is not a major limitation and greatly simplifies the processing. The hypertexture can contain a single texture, allowing the user to specify parameters modifying its appearance. This texture is not limited to a single colour, but can contain a pattern (which is effectively a 3D scalar field) which can produce an effect of stripes, spots, or anything the user can imagine.

2.3.2 Data structures

The parameters must be stored in memory somewhere and associated with the object to which the pelt is attached. As the hypertexture is described entirely by the parameters and there are no other data to store, there is no need for complex data structures in this project.

2.3.3 Pelt intersection

When the scene is being traced, the function for finding the object a ray intersects must also be able to detect intersections with a pelt and decide when it is necessary to call the fur simulation function. There are several ways to do this.

The first way I thought of is to create a ‘point of closest approach’ function for each primitive. Given a ray and an object, this function would determine how close to the object’s surface the ray passes, like a generalised intersection function. If this value is less than the extent of the pelt, it would trigger fur simulation.

Another way is to just use the object’s bounding box (bbox). Every object has a bbox, and before the true object intersection is performed, the ray is intersected with the bbox. If there is no bbox intersection, there cannot be an object intersection. The naïve approach would be to simulate fur throughout the whole box and to ensure that the density model behaves sensibly where there should be no fur. Although easy to code, this would be incredibly slow. Note that the bbox itself has to be enlarged by twice the hypertexture extent to ensure it contains the hypertexture.

The method I later came up with and eventually settled on is to create a ‘hyperbounding’ object. This is an object, usually of the same primitive as the real object, that bounds the fur. The distance between the hyperbound and the object is always at least the extent of the fur, and the hyperbound is generated at parse time. For example, the hyperbound of a sphere centred on the origin of radius 3.0 with fur of extent 1.0 is a sphere centred on the origin of radius 4.0. When an object is intersection-tested, it is checked to see if it has a hypertexture: if so, the hyperbound is intersection-tested and the `Intersection` returned has a flag set to show it was a hypertexture intersection. Thus even primitives for which it is difficult to get the point of closest approach can quickly be tested.

2.3.4 Transformation into texture space

In traditional 3D rendering it is usual to transform co-ordinates on an object’s surface into texture space. Texture space is 2D and lies entirely on the surface of an object. Texture-space co-ordinates are used to map a flat texture onto an object. They are often called ‘UV’ co-ordinates.

In this project, the hypertexture is 3D, so the texture space also has to be 3D. The third dimension gives distance from the surface. I adopt the usual convention to call 3D texture-space co-ordinates ‘UVW’, where \mathbf{u} and \mathbf{v} take their usual meanings as vectors on the surface of the object, and \mathbf{w} is everywhere

normal to the surface. The method to perform this transformation is geometry-dependent and again there are several techniques to choose from.

The easiest, and by far the most commonly used, is to make the user specify this somehow. Most systems simulate the fur in a geometry-independent unit volume in UVW space and require the user to tile this over the desired surface using a modelling program. Although quite fast at runtime and relatively easy to program, it is unsuited to text-based scene description and conflicts with my goal of allowing the user to merely say that he wishes a particular object to be furry.

Another method would be to have a geometry-dependent function that takes an object and a ray segment in world-space, and returns a list of points in UVW space. It would then suffice to linearly interpolate between those points while travelling along the ray. This method appears fast, and would be ideal for a scan-conversion renderer where all surfaces are composed of polygons (i.e. piecewise flat). However, in a system like POV-Ray where curved surfaces really are curved, it would be necessary for the function to be able to return arbitrary curves and for these to be evaluated at each point. This increases the complexity, both in programming and execution time, so I decided not to implement this scheme.

Instead I decided to have a geometry-dependent projection function. This takes a point in world-space and returns a point on the surface of the object such that the normal at the returned point passes through the original point. (It projects the point normally onto the surface of the object.) This is a slightly simpler scheme and avoids the problems with curves, but has problems of its own, as we shall see later (section 3.3).

2.3.5 Density model

To perform simulation, the system must have some model of where in space the strands are and how much they affect the current ray. Previous techniques have tended to use a volume in texture space that is pre-filled with anti-aliased fur (e.g. by tracing the path of a particle system) and to use one of several methods to tile this volume over the surface.

[PH89], however, uses a procedural technique. Perlin and Hoffert use the \mathbf{u} and \mathbf{v} co-ordinates to evaluate Perlin noise [Per02] over the surface of the object. Perlin noise is a function⁴ from points in space to real numbers in the range $[0, 1]$, such that nearby points give similar results but distant points give uncorrelated results. This noise is then modulated to give a pattern whose value is one at several points and tails off rapidly to zero moving away from those points. The modulation is parametrised to allow the user to specify the frequency and falloff.

This technique, being procedural, is closest to the effect I want to achieve as well as to the design philosophy of POV-Ray.

⁴Actually, there are several functions referred to as Perlin noise, not all of which were originated by Perlin, but they all give similar results. In POV-Ray the user can choose from three noise generators in the scene description file.

2.3.6 Lighting model

Fur scatters light differently to traditional Lambertian and reflective surfaces, so the lighting equations used elsewhere in POV-Ray cannot be used here. I use an *ad hoc* lighting model derived in [KK89]. I will not repeat the derivation here, but present the results.

The diffuse (Lambertian) component is that which is independent of viewing direction, the general, dull scattering that looks like rubber. It is given by

$$2k_d|\mathbf{t} \times \mathbf{l}|$$

where k_d is the diffuse coefficient of the surface (set from the texture), \mathbf{t} is the tangent to the strand, and \mathbf{l} is the direction of the shadow ray. Note that any strands pointing towards the light are dark.

The specular component is the shiny highlight where the angle between the viewing direction and the normal is similar to that between the lighting direction and the normal, like the highlight visible on metal objects. It is given by

$$k_s(\mathbf{t} \cdot \mathbf{l} \mathbf{t} \cdot \mathbf{e} + |\mathbf{t} \times \mathbf{l}| |\mathbf{t} \times \mathbf{e}|)^p$$

where k_s is the specular coefficient and p is the Phong shininess exponent (both set from the texture), and \mathbf{e} is the direction of the eye ray. Note that in a pelt specular highlights appear even if the strand is *between* the eye and the light. Experimentation with a desk lamp and a thin, shiny cylinder such as a pen will convince the skeptical reader that this is realistic.

2.4 Project planning

This section covers how I carried out the project work and the decisions that had to be made as to how to go about this.

2.4.1 Software engineering strategy

The development process is well reflected in the order of this document, but was far more incremental. After each section was coded it was merged into the source and unit-tested extensively using my test suite (see section 4.1.2). Only when I was satisfied with its correctness did I label it complete and move on to another problem area. In this way my development process was similar to the evolutionary model, with the stages in the model being problem areas and features ('normal perturbation', for example) rather than development stages (like 'coding').

2.4.2 Choice of tools

Much of the tool choice was taken out of my hands by using an existing project. POV-Ray uses the GNU autotools and make for compilation management, and its own documentation system. No rapid prototyping or GUI design was necessary, so Vim sufficed as a development environment.

2.4.3 Source control and backup provision

CVS was the obvious option for source control as it provides all the source control features I need and I was already familiar with its use and administration. One particularly useful feature was the ability to maintain a separate ‘vendor branch’, allowing changes in the upstream POV-Ray sources to be easily merged into KEV-Ray. For speed of access and ease of management the CVS repository was kept on my personal computer.

The backup system was built on this, consisting of a cron job to archive and compress the entire repository daily and send this to the backup server, Pelican. For extra peace of mind I kept a second periodic backup on a removable drive. This system was tested at the start of Lent term when my computer suffered a critical error on the filesystem containing the repository. The backup was restored within 24 hours and no work was lost.

Chapter 3

Implementation

Once the design decisions had been made, the work to be done consisted of writing code to perform fur simulation and integrating this new code with existing POV-Ray code, most of which is uncommented and dates back ten years.

3.1 Parsing

Reading the POV-Ray parser source code gives me a sense of how contemporary readers of Newton’s work on the differential calculus must have felt. The topic is complicated enough on its own, but he wrote in Latin to make sure that only properly educated people could understand it. Even then, readers had to understand a new, poorly explained notation invented specifically for the purpose and since made obsolete by better notation invented elsewhere (by Leibniz).

Rather than use a popular parser/tokenizer-generator like `lex` and `yacc`, POV-Ray uses its own system, which basically consists of an ‘expect’ script written in C preprocessor statements. This is mainly for performance reasons — the POV-Ray parser is highly optimised for SDL and faster than today’s automatically generated parsers, let alone those of fifteen years ago — but also for other operational reasons, like being able to debug it without knowing the internal workings of the parser generator. I will not go into further details here except to the extent that they are relevant to the changes that had to be made.

Tokens to be recognised in SDL are stored in a statically created data structure. This contains for each token an ID, the text of the token, and a string describing its meaning, used for generating error messages. To add a new token, a preprocessor constant must be added to the enum of token IDs, and this constant must be associated with a string in the token data structure. The order of token IDs influences the binding precedence of operators (amongst other things) so must be chosen carefully.

Parsing is performed by creating a `Parse_Hyper` function. This must accept a ‘{’ character from the token stream, then possibly an identifier for an existing

(previously declared) ‘hyper’ block. If such an identifier is the first token, the data structure corresponding to the named ‘hyper’ must be copied to a new structure on the heap and subsequent modifiers applied to this copy. Otherwise a new structure must be allocated.

Following this other tokens are expected, each of which sets some property of the hypertexture and may cause other functions to be called to parse e.g. a ‘texture’. After a ‘}’ is read the function returns.

This function is called by the existing function `Parse_Object_Mods`, which is responsible for anything that modifies an object, for example a ‘texture’, ‘transform’, or the new ‘hyper’. I added a few lines to recognise the new ‘hyper’ token and in such a case call `Parse_Hyper` and make the object point to the newly generated structure.

As in several places in this project the actual code written was quite simple but finding where to write it was hard work.

3.2 Bounding

I add to each primitive a hyperbound function, which takes an object and a float indicating the extent of the hypertexture. It produces another object which completely contains the hypertexture (a hyperbound). This is performed at parse time by the `Post_Hyper` function.

The hyperbound function takes a pointer to an `Object` and a double representing the extent of the hypertexture. It needs to know nothing more of the hypertexture. It uses the primitive’s existing functions to create a new object larger by the extent than the original object, and sets the `Hyper->Bound` field to point to this new object. `Post_Hyper` does all other necessary operations, like copying the transformations applied to the original object to its hyperbound. There is also a `Default_Hyperbound` function, which is used if the primitive in question does not support automatic hyperbounding (i.e. those primitives for which I have not written a hyperbounding function). This function causes an error to be raised.

It is possible for the user to manually specify a hyperbound using the ‘bounded_by’ keyword when defining the hypertexture.

Also the object’s `bbox` is enlarged in each direction by the extent so that the `bbox` encloses the hypertexture as well as the object. This is necessary to ensure that the existing intersection-testing functions will return all objects whose hypertextures intersect the ray.

3.3 Projection

At the heart of my algorithm is the projection functions. For a primitive to support hypertexture it must define such a function, which takes a pointer to the `Object` in question, a `Vector` representing the point to be projected onto the surface, and a pointer to an `Istack` to return its results. This last struct is

a stack of `Intersection`s and is the same data structure used by ray-primitive intersection functions. Using this as the interface between `Simulate_Hyper` and projection functions is quite useful for several reasons. The most pressing is that all the other primitive functions that need to be called, e.g. to get UV co-ordinates or to evaluate a normal perturbation function, would usually be called after intersecting the object with a ray, so take an `Intersection`. The `Intersection` contains fields for all the parameters we need to pass. The `Istack` itself is a highly optimised data structure, and are allocated from a pool to avoid calls to `malloc`.

There is a `Default_Project` function which raises an error, for primitives for which I have not written a projection function. Primitives using this function cannot be given hypertextures.

3.4 Pelt intersection

Pelt intersection is about finding the ray segment along which hypertexture simulation must be performed, and if it is necessary, performing the simulation in an efficient manner and blending the colour with other contributions to the ray.

During the trace, one of the tracing functions is called. There are several tracing functions, and each uses a different technique to find objects the ray *might* intersect (the exact details of which are unimportant), but eventually it will call `Intersection` on a candidate object to check whether it intersects the object. If the candidate object has no hypertexture (its `Hyper` field is null) the intersection test proceeds as normal using the existing code; otherwise, the hyperbound is tested for intersections as well as the object itself. The closest intersection is returned: if it is with the hyperbound, the `Hyper` flag of the `Intersection` is set.

The `Ray` struct already contains an array of pointers to `Interiors`, which represents the hollow, media-containing objects the ray is in and is used to determine whether to sample a ray segment. An array is used instead of a list because iterating over an array is much faster than over a list and the overhead associated with rearranging pointers in the array is amortized over many accesses. It does have the disadvantage of putting a hard upper limit on the number of media the ray can be in at once, but this is set high enough to not cause inconvenience, and can be changed at compile-time.

Guided by this existing implementation, I added to `Ray` an array of pointers to `Objects` with hypertextures. This is used in the same way. In the tracing functions, if the ray is infinite and is in a hypertexture, `Simulate_Hyper` is called. If the ray is non-infinite (i.e. it intersects an object), `Determine_Apparent_Colour` is called as usual.

The existing participating media code, I feel, does things in an inefficient order. `Determine_Apparent_Colour` first fires a shadow ray to each light, then does the usual shading calculations, then spawns new rays for reflections and refractions. After all this, it checks whether it needs to call the media code and

does so. This order of operations makes blending colours (for partially transparent objects) easier, but means that sometimes a lot of computation is done and then discarded because the medium is opaque. Because complete opacity is a common case for fur, I wanted to avoid this waste so do things in a different order. I renamed `Determine_Apparent_Colour` to `Determine_Intersection_Colour`, and made `Determine_Apparent_Colour` a wrapper function that checks whether hypertexture sampling is necessary and later calls `Determine_Intersection_Colour` if appropriate. The logic to determine whether hypertexture sampling is necessary is quite complicated as there are many special cases to consider.

If the ray starts outside all hypertextures and the intersection has the `Hyper` flag clear, this is the normal (hypertexture-free) case, and `Determine_Intersection_Colour` is called as normal. If the flag is set, the ray is entering a hypertexture. The ray is copied and its origin set slightly further along than the intersection point, to avoid errors caused by floating-point inaccuracy. `Trace` is called on this new ray.

If the ray starts inside a hypertexture, the ray segment between the ray's origin (or eye point) and the intersection is simulated. If the intersection has the `Hyper` flag set, a new ray is spawned and traced as above and the result blended with the simulation result. If the flag is clear, `Determine_Intersection_Colour` is called to determine the colour at the intersection and the result blended with the simulation result. This may cause shadow, reflection, and refraction rays to be spawned as usual. There is, however, an optimisation: if the transparency of the hypertexture (determined by the simulation) is lower than a certain value then the simulation result alone is used, avoiding spawning a new ray or calling `Determine_Intersection_Colour`. This 'bailout' value was already used to avoid ray-spawning in POV-Ray's adaptive depth control mechanism, and can be set by the user. By default its value is $1/255$: the smallest change visible in a 24-bit-per-pixel image.

This order of events is efficient, particularly in the common case where no hypertexture is involved. It correctly handles all special cases, including where there are multiple overlapping hypertextures which are left by the ray in a different order to that in which they are entered. It also behaves sensibly when a foreign object lies in the pelt, although there is no method for communicating this to the simulation function (e.g. to allow the strands to bend to avoid hitting the object).

The only problem with this approach is that it co-operates poorly with the existing participating media: if a ray segment passes through both hypertexture and media, only the hypertexture will be simulated. Thus the region of space containing hypertexture cannot contain media. This may result in a visible boundary at the surface of the bounding object if it is within media or if atmospheric media (extending across the whole scene) are used. However I see no way to accurately and efficiently allow both parts of the code to simulate the same ray segment without unfeasibly large changes to the media code. It would require information about the opacity accumulated along the ray to be passed back and forth, and would necessitate large changes to the sampling methods

of the media code to make them compatible with my code. Such changes would seriously reduce the efficiency of both features and are somewhat outside the scope of this project.

3.5 Sampling

After hypertexture has been created and detected, it remains to actually sample ray segments throughout the volume. I consider that part of the code in this section. Volume sampling is a relatively old technique for discrete integration: here we are integrating light intensity (i.e. colour) along a ray by summing the contribution at each small segment of the ray. Each sample is influenced by the density of strands in that region and how much light they reflect.

For efficiency there are two different functions for simulating hypertexture: `Simulate_Hyper` for normal rays, and `Simulate_Hyper_Shadow` for shadow rays. Because shadow rays do not require lighting calculations, this allows me to optimise the shadow function more, and runs roughly five times faster than the single-function implementation, for a small code size increase. However, the two functions are very similar so I will cover them both together. A pseudocode outline of each is given in appendix A and it may be helpful to follow that while reading this section.

3.5.1 Initialization

Each function takes three pointer arguments: the `Ray` to sample along, an `Intersection`, and a `Colour` to store the result. Like the participating media code, they sample from the origin of the ray to the depth given by the `Intersection`. Both functions use the array in the `Ray` for the hypertextures to simulate: if this is empty correct behaviour will result, but the optimisation of testing for this case and exiting early is not performed. I do not expect that this function will be called in this case, and the logic described in the previous section avoids doing so, and saves the cost of a comparison for each ray (also for each shadow ray, which is a large cost).

`Simulate_Hyper` starts by getting an array of the scene's global lights that affect the given ray segment (i.e. excluding spotlights and cone lights in other parts of the scene), and an array of the local lights for each object.¹ These arrays are stored on the heap, but to avoid calls to `malloc` a single pool is allocated at startup and dynamically reallocated as necessary. This gives a 1% speedup but sacrifices re-entrancy, which is not needed for this function. The pools are freed by `Deinitialize_Hyper_Code` on shutdown. The light lists are supplied by some code previously used for media, which I have broken out into separate functions called from both here and the media code.

¹POV-Ray allows the user to associate lights with a particular object, so that the light only illuminates that object. This is useful for fill-in lights.

3.5.2 Sample size

Once the light lists have been generated, in both functions it is necessary to determine the sampling size. Intuitively, one might think that this can be determined automatically using Nyquist's theorem², because the frequency of the fur is known, but unfortunately this frequency is given in texture space, and the relationship between texture space and world space is not generally known.

Thus the user can specify a sample size for each hypertexture: the minimum is taken out of all hypertextures being sampled. This size is adjusted in the cases of extremely long (or infinite) and extremely short ray segments: in the first case to ensure the algorithm terminates, and in the latter to speed up sampling at the silhouette.

3.5.3 Iteration

Then each function enters a for-loop parametrised on the depth. For strict accuracy, the final sample should have its size adjusted to fit the distance between the penultimate sample and the exit point. Most other systems do this. However, for sample sizes small enough that accuracy is important, the change is small enough that the inaccuracy is unnoticeable, especially because the last sample is the one that has least contribution to the final result (because it is obscured by all the other samples), so I avoid this small extra cost. There is an early-exit test: if the hypertexture sampled so far has reached complete opacity, the function can return because the remaining hypertexture cannot affect the result and need not be sampled.

In `Simulate_Hyper`, `Test_Shadow` is called to determine the lighting intensity from each light. As noted earlier, it calls `filter_shadow_ray` instead of `Determine_Apparent_Colour`. The hypertexture logic is the same in each case, but the former calls `Simulate_Hyper_Shadow` instead of `Simulate_Hyper`. Because no additional rays are spawned, `filter_shadow_ray` does its usual operations *before* processing hypertexture, because in this case it is likely that hypertexture processing can be avoided if the object intersected is opaque.

Then, a nested for-loop is entered. This iterates over all hypertextures being sampled. The `Colour_SampleResult` has been initialised to black and accumulates the contribution of each hypertexture to the current sample.

3.5.4 Projection into texture space

The object's projection function is called, projecting the sample point down to the surface. Another nested loop iterates over the projected points but in most cases there is only one such point. The depth of the point (distance between the original point and its projection) is compared with the hypertexture's extent for another quick bail-out. If the object has a UV mapping, this is performed on

²Nyquist's theorem is an important result in sampling theory which says that to recover a signal of maximum frequency f you need to sample it with frequency $2f$.

the projected point; otherwise, UV co-ordinates are faked by the simple formula

$$\begin{aligned}\mathbf{u} &= \mathbf{x} + \mathbf{y} \\ \mathbf{v} &= \mathbf{z}.\end{aligned}$$

This formula gives quite bad results for distributing strands over the surface, but without UV co-ordinates I cannot determine co-ordinates corresponding to two vectors always tangent to the surface. I have considered various other schemes to work around this, but none is satisfactory, and this at least is cheap. Whichever method is used, the depth then becomes the \mathbf{w} co-ordinate, giving the original point three-dimensional texture space co-ordinates.

3.5.5 Normal perturbation

Normal perturbation (or bump-mapping) is an old technique in computer graphics to give flat surfaces the illusion of bumpiness. When the object is intersected, before the shading calculations are performed, the surface normal is perturbed by some amount. This affects the shading calculation such that the part of the surface intersected with appears to be at a different angle to the lights and the viewer. The amount of perturbation varies over the surface to produce an effect of curvature. In this project, I use the same idea in a new way to allow the user to make the pelt curve as if a force acts on it. Bear in mind that each strand is effectively a visible surface normal.

If normal perturbation is applied to the hypertexture's texture block, the existing perturbation function is called here to determine the perturbation at the projection point. This is used to calculate the vector difference between the original and perturbed normals, which vector is added to the texture-space co-ordinates. I call this "wibbling", and it effectively warps the mapping between world and texture space, allowing the hair to bend as if a force is acting upon it. This will give interesting results if the normal perturbation (which is, mathematically speaking, a scalar field) is discontinuous.

3.5.6 Density and shading

Following this, `hyper_density` is called to determine the surface density at the sample point. This is added to an accumulator and is used to scale the contribution to `SampleResult`. The existing function `Compute_Pigment` is called to evaluate the colour of the object's pigment at the sample point (in texture space). The ambient shading is added, and if the sample point is lit, another for-loop calls `scattering` for each light source and adds the scaled shading contributions to `SampleResult`.

3.5.7 Exit

The per-projection and per-hypertexture loops end, and `SampleResult` is scaled by the transparency of previous samples. This transparency value is updated

with the surface density accumulator, and the per-sample loop ends.

3.6 Density

In section 2.3.5 I covered the density model, which determines where in texture space the strands are. Here I will go into a little more detail about its implementation.

The `hyper_density` function takes a point in texture space and a non-null `Hyper`, and returns a floating-point value representing the density of microsurfaces for the given hypertexture at the given point.

I divide the `u-v` plane into squares of the hypertexture frequency, each containing one strand. The texture-space co-ordinates are split into two integers giving the co-ordinates of the square the point is in and two floats giving the offset within this square. Pseudo-random numbers are produced quickly by hashing the two integers and using this to index into a short array of statically computed random numbers. Two such numbers are used as offsets to jitter each strand's placement within its square, making the placement less regular. Another is used to vary the length slightly to avoid a brush-like pelt.

The float co-ordinates within the square are processed to give a single value describing the point's Euclidean distance from the strand's centre (which rises vertically in the `w` direction). After deriving and testing several types of fall-off of density with distance, I have adopted the function that simply returns a density of 1.0 for points inside the strand and 0.0 for points outside (i.e. further away than its radius, which is user-specified and defaults to 0.3 of the square size). This function gives the most realistic results out of all those I have tested.

3.7 Textures and Lighting

As I mentioned in section 2.3.6, the usual lighting calculations do not work for fur. The `scattering` function implements the modified calculations. It takes a point in texture space, the tangent to the strand (which is always the perturbed normal to the surface), the light colour and direction, the eye (viewing) direction, the pigment colour computed above and the `Hyper`. It is also passed a pointer to `SampleResult` to return its results. The lighting equations are computed and the function returns.

3.8 Modularity and Other Hypertexture Phenomena

Reviewing the above sections, you may notice that the only areas of code that are specific to fur are the two functions `hyper_density` and `scattering`. This is no accident. Although this project is concerned particularly with fur, previous

research has tended to propose general techniques for hypertexture phenomena and then apply these to fur as a specific application.

It would be possible to not have calls to these two functions hard-coded into `Simulate_Hyper` and `Simulate_Hyper_Shadow`; instead, pointers to them would be placed into the `Hyper` structure.³ Alternative density and scattering functions representing other hypertextures could be written. The SDL parser could insert the appropriate function pointers according to a keyword in the hyper block. In this way, or using more modern object-oriented techniques, a more general hypertexture system could be written for little additional programming complexity and execution time. This would be able to simulate trees on a far-off hillside, fire, aura, or feathers (which could have been useful for the recent film *Valiant*). Although KEV-Ray already does a good impression of a carpet, more specialised fur-like density functions could handle carpets of various types, grass (with daisies), moss, or sheep fleeces (which are too curly to be handled correctly by my technique).

³This would incur a small overhead as these small functions are currently inlined by the compiler.

Chapter 4

Evaluation

This chapter covers the correctness of the project in two senses: the testing to ensure that it behaves as expected in all cases, and the comparison with other projects to ensure that expectations are not too narrow.

4.1 Testing

4.1.1 Specification

Because of the size of the project, I found careful specification of each function useful in avoiding bugs. By considering the special cases of each operation in an abstract manner even before designing or implementing algorithms, I was able to ensure that the uncertainty and confusion that often causes bugs was absent. This included reasoning about when pointers are allowed to be null and when memory is allocated and freed. Some of this specification found its way into the previous chapter in the description of each function, but again, the greatest body of work is in the derivation of specifications for the existing functions I had to call and/or modify, in order that I could do so correctly.

4.1.2 Test Suite

Formal reasoning is not a panacea, so production of a test suite was essential. Having an extra person write the test suite in parallel with the coding would have ensured that the tests (thus the bugs they would detect) were not limited by the world-view of the coder (thus the bugs he would write), but as this was not the case I had to do my best to ensure that the testing was still effective.

Using a consistent set of test scenes was useful because it gave a good overview of the progress of the project, as more and more features started to work properly; also because subtle inaccuracies that might have gone unnoticed on a first viewing were obvious on an image of a scene I had seen on many previous occasions. Because of the aesthetic nature of the project, automated testing (beyond simply testing compilation and lack of error messages) was not

possible, making the testing and debugging effort more considerable than for a program whose output is machine-checkable.

4.1.3 Regression testing

Because of the huge variety of features POV-Ray offers and the possible interactions between these features, I found thorough regression testing quite unfeasible. However, I used KEV-Ray as I would normally use POV-Ray throughout the development process, and the range of SDL files I rendered gave all the existing features a significant ‘field test’ to ensure they still worked as expected.

4.2 Comparative Evaluation

Evaluating this project is quite difficult. Normally one would compare the new implementation against existing programs, but this is an implementation of a new algorithm. Normally one would compare this algorithm to others designed for the same task, but I have been able to find only one implementation of ray-traced fur that runs on modern systems. It is another POV-Ray patch, developed independently of mine and released while I was working on this project. Normally there is a quantitative accuracy that can be assigned to the output of programs, but in this case the reader will have to make his own aesthetic judgement.

I present three programs: POV-Ray with the media fur patch, KEV-Ray, and Blender. I also present some sample art produced by my system, to demonstrate the range of effects achievable with hypertexture fur and the utility of the features I have provided.

4.2.1 Media fur patch

This patch treats fur as being a clever type of participating media. Rather than change the whole intersection system to allow the pelt to be outside the object, it uses the existing media support and has the pelt inside the object. The major limitation of this is that the transparency of the fur is not handled automatically: any absorption of light must be specified by the user and does not have the structure of fur. The fur cannot cast spiky shadows or shadow itself realistically. This separate treatment of scattering and absorption causes the fading of colour towards the horizon in the sample pictures.

The user must specify the container object manually: it is not automatically generated. The container object is the equivalent to my hyperbound, but it must be a strict container as fur will appear in all space inside the container. In KEV-Ray, if the hyperbound is larger than the region in which fur would appear, rendering is slower than it would usually be, but the result is the same. In order to give the strands direction, the user must also give the shape of the object the pelt is attached to. This cannot be any shape: it must be one of a small list of

primitives. The documentation suggests that having the pelt emerge spherically from the centre of the object is a good enough approximation for most uses.

The patch does not allow arbitrary force fields to comb the fur, but the user may give a single vector which will be treated as a constant force over the whole object. It also does not allow UV-mapped pigments to be applied to the fur, nor fur colour to vary over the length of a strand. Although it implements diffuse and specular reflection calculations like KEV-Ray, it does not allow ambient shading to be used.

The patch does have one feature not offered by KEV-Ray: it allows the fur to be wavy. A single parameter ‘waves’ controls the amount of waviness. I had planned to make this feature available in KEV-Ray, and it would be quite easy to implement using the existing functions for producing turbulence in patterns, but I did not have enough time to include every feature.

For comparison, I include pictures rendered with KEV-Ray and the media fur patch. The reflective sphere over a checkered plane is somewhat of a tradition in ray-tracing. The first image is KEV-Ray, with a fur length of 2.0 units. The second is also from KEV-Ray, but with a fur length of 4.0 units. The other two are the same, but rendered with the media fur patch.

The final picture demonstrates using normal perturbation to simulate gravity. Each strand fades to grey towards the inside. This picture cannot be replicated using media fur.

Each of these scenes is rendered with full, adaptive oversampling (anti-aliasing) on my lightly-loaded Athlon 2800+ computer running GNU/Linux with a 2.6.11 kernel. To prevent unfairness from compiler optimisations, both programs were compiled on my computer with the same optimisation flags. The oversampling is a large cost for fur: times without oversampling are on the order of five minutes.

Picture	Renderer	Memory usage / bytes	Time taken / s
plane 2	KEV-Ray	677602	5292
plane 4	KEV-Ray	677602	9353
plane 2	media fur	653573	5214
plane 4	media fur	653573	9826
ball	KEV-Ray	3816288	3215

4.2.2 Blender

Blender [TR04] is an open-source 3D modelling and animation program with a scan-conversion renderer. It has a young but stable fur rendering system based on a particle system. This means that the limitations and features of its particle system apply also to fur: perhaps the most significant limitation being that memory usage scales linearly with the number of strands in the scene (whereas in KEV-Ray memory usage is independent of this). Only polygon meshes can be given fur, but as Blender only has a few primitives and many other features work only on polygon meshes, this is a small limitation. Blender is far from state of the art in scan-conversion: I mention it merely to compare my results with an existing open-source system of comparable popularity to POV-Ray.

Unfortunately, the main disadvantage of the system is that it is difficult to use. The user needs to manually create a texture to apply to the particles to make them look like fur, and must adjust the particle system parameters to make them geometrically like fur. In three days of trying, with reference to the tutorial given in [TR04], I was unable to produce an image that looked anything like fur. For this reason the final image I include is a standard image supplied with Blender to advertise the feature. It is rendered at low resolution and I do not have statistics for it.

Chapter 5

Conclusions

My project was a success. It met all of the core requirements outlined in the proposal and most of the extensions. It successfully integrates a novel fur synthesis method into an open-source ray-tracer.

5.1 Further Work

As with many Part II projects, I am left with the feeling that I could have done some really interesting things given another year. Though my core project works well and I have completed many of the extensions suggested in my original proposal, the time I have spent thinking about fur and ray-tracing has made me realise how many possibilities remain to be explored.

POV-Ray is a program with many features and options, and the implementation took long enough that there was not time at the end for me to fully test and experiment with how my new features interact with the existing techniques. In particular, image-mapped textures would have been interesting to try and produce novel effects. Instead, I will leave such explorations to users of my patch. Also, it would have been both entertaining and useful to make the projection and automatic bounding functions work for some of POV-Ray's more exotic primitives, such as Julia fractals and heightfields¹

In my original research, I found much literature (e.g. [PC01], [SF92]) on physical simulation of several hypertextures (mainly grass and fur) for animation, to cover the behaviour of these objects under special circumstances (e.g. in the wind, or when wet). A user would be able to use these algorithms in external programs to control the SDL and thus the fur, but it would have been interesting to implement some of these internally as part of KEV-Ray, so a user would be able to just specify, for example, wind strength and direction and have the turbulence automatically created in the animation. There is at least another six months' work in this, though, and it could form the basis of a future project.

¹A heightfield takes an image, maps it onto a segment of the x-y plane, and takes the height as the image intensity at that point. It is often used to create landscapes.

Another feature useful for animation would be the ability to turn the procedural fur into explicit geometry. Though the geometry would render more slowly, it is much more flexible. Animators could use this to allow fur to interact with other objects, to be tied in knots, or to be shed.

5.2 Conclusion

This project has, in many ways, been the bane of my life over the past year. The initial search for inspiration for a proposal and supervisor gave way to a feeling of premature commitment to a do-or-die idea. The initial excitement at inventing a new approach to an old problem was replaced by a dread of basing my project on a technique that had never been tried before. Each time I felt pride at having overcome what last week had been insurmountable difficulties, the prospect of the problems yet to face gave me renewed reason to worry. The security I once felt at having an established program to work with was supplanted by the horror of trying to peel away years of old hacks in an attempt to understand old, uncommented code. Even as I write this, the pleasant anticipation of the consummation of these months of effort is overshadowed by a hollowness at how much is left undone. However, it was worth it. The project has allowed me to experience both the thrill of research and the feeling of accomplishment of working on useful, widely-used software; both the demands of careful specification and the compromises of efficiency; both the defeat of untraceable bugs and the victory of seeing expected output. From before the beginning to after the end, KEV-Ray has filled my thoughts, stimulated my creativity, and shaped my development as a computer scientist.

Appendix A

Pseudocode listings

A.1 Simulate_Hyper

```
void Simulate_Hyper(RAY* Ray, INTERSECTION* Exit, COLOUR Result)
    DBL blocking, Density, coeff;
    int samples;
    VECTOR P,Q, Delta, Wibble;
    UV_VECT Uv;
    OBJECT *Obj;
    INTERSECTION *Proj;
    COLOUR SampleResult;
    RAY Light_Ray;
    COLOUR Pattern_Colour, Light_Colour;
    DBL sample_size = MAX_DBL;

    get number of lights;
    resize lighting pools;
    get global light list;

    foreach Obj in Ray->Hypers
        get local light list(Obj)
        if (Obj->Hyper->Sample_Size < sample_size)
            sample_size = Obj->Hyper->Sample_Size;

    blocking = 0.0;
    coeff = 1.0;

    if (Exit->Depth > 2000*sample_size)
        sample_size = Exit->Depth / 2000;
    else if (Exit->Depth < 2 * sample_size)
        sample_size = Exit->Depth;
```

```

Delta = Ray->Direction * sample_size;
P = Ray->Initial - (sample_size/2.0 * Ray->Direction);
for (samples = ceil(Exit->Depth / sample_size);
    samples > 0 && blocking < 1.0;
    samples--)
    P += Delta;
    SampleResult = <0,0,0>;
    foreach global_light
        global_light.shadowed = Test_Shadow(global_light.Light,
&depth, global_light.Light_Ray,
Ray, P, global_light.Colour);

    foreach Obj in Ray->Hypers
        Pattern_Colour = <0,0,0>;
        foreach Proj in Project(Obj,P)
if Proj->Depth <= Obj->Hyper->Extent
    Proj->PNormal = Proj->INormal;
    if (Test_Flag(Obj, UV_FLAG))
        UVCoord(Q, Obj, Proj);
    else
        Q[U] = Proj->IPoint[X] + Proj->IPoint[Y];
        Q[V] = Proj->IPoint[Z];
        Q[W] = Proj->Depth/Obj->Hyper->Extent;

    if (Obj->Hyper->Texture->Tnormal != NULL)
        Perturb_Normal(Proj->PNormal,
Obj->Hyper->Texture->Tnormal, Proj->IPoint, Proj);
        Wibble = Proj->INormal - Proj->PNormal;
        Wibble *= Q[W];
        Q += Wibble;

    Density = hyper_density(P, Q, Obj->Hyper) * sample_size;
    Compute_Pigment(Pattern_Colour,
        Obj->Hyper->Texture->Pigment, Q, NULL);
    if (Density > 0.0)
        blocking += Density;
        SampleResult += Obj->Hyper->Texture->Finish->Ambient
            * Density * Pattern_Colour * Frame.Ambient_Light;
        foreach global_light
            if (!global_light.shadowed)
scattering(Proj->PNormal, Ray->Direction,
            global_light.Light_Ray.Direction,
            global_light.Colour, Obj, Pattern_Colour,
            Density, SampleResult);

```

```

    foreach local_light in local_light_list(Obj)
        if (!Test_Shadow(local_light.Light, &depth, &Light_Ray,
            Ray, P, Light_Colour))
            scattering(Proj->PNormal, Ray->Direction,
                Light_Ray.Direction, Light_Colour, Obj,
                Pattern_Colour, Density, SampleResult);

    Result += SampleResult * coeff;
    coeff = 1.0 - blocking;
    if (coeff < 0.0) coeff = 0.0;

    free local light lists;
    Result->transmit = coeff;

```

A.2 Simulate_Hyper_Shadow

```

void Simulate_Hyper_Shadow(RAY *Ray, INTERSECTION *Exit,
    COLOUR Result)
    DBL blocking = 0.0;
    int samples;
    VECTOR P,Q, Delta, Wibble;
    UV_VECT Uv;
    OBJECT *Obj;
    INTERSECTION *Proj;
    DBL sample_size;

    foreach Obj in Ray->Hypers
        if (Obj->Hyper->Sample_Size < sample_size)
            sample_size = Obj->Hyper->Sample_Size;

    if (Exit->Depth > 1000*sample_size)
        sample_size = Exit->Depth / 1000;
    else if (Exit->Depth < 4 * sample_size)
        sample_size = Exit->Depth;

    Delta = Ray->Direction * sample_size;
    P = Ray->Initial -(sample_size/2.0 * Ray->Direction);
    for (samples = ceil(Exit->Depth / sample_size);
        samples > 0;
        samples--)
        P += Delta;
        foreach Obj in Ray->Hypers
            foreach Proj in Project(Obj,P)
if Proj->Depth <= Obj->Hyper->Extent
    Proj->PNormal = Proj->INormal;

```

```

if (Test_Flag(Obj, UV_FLAG))
    UVCoord(Q, Obj, Proj);
else
    Q[U] = Proj->IPoint[X] + Proj->IPoint[Y];
    Q[V] = Proj->IPoint[Z];
    Q[W] = Proj->Depth/Obj->Hyper->Extent;

if (Obj->Hyper->Texture->Tnormal != NULL)
    Perturb_Normal(Proj->PNormal,
Obj->Hyper->Texture->Tnormal, Proj->IPoint, Proj);
    Wibble = Proj->INormal - Proj->PNormal;
    Wibble *= Q[W];
    Q += Wibble;

blocking += hyper_density(P, Q, Obj->Hyper) * sample_size;
Result *= exp(-blocking);

```

Bibliography

- [Bid95] H. B. Bidasaria. A new method for modeling of hair-grass type textures. In *CSC '95: Proceedings of the 1995 ACM 23rd annual conference on Computer science*, pages 109–113, New York, NY, USA, 1995. ACM Press.
- [KK89] J. T. Kajiya and T. L. Kay. Rendering fur with three dimensional textures. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 271–280, New York, NY, USA, 1989. ACM Press.
- [LPPFH01] Jerome Lengyel, Emil Praun, Adam Finkelstein, and Hugues Hoppe. Real-time fur over arbitrary surfaces. In *SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 227–232, New York, NY, USA, 2001. ACM Press.
- [PC01] Frank Perbet and Maric-Paule Cani. Animating prairies in real-time. In *SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 103–110, New York, NY, USA, 2001. ACM Press.
- [Per02] Ken Perlin. Improving noise. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 681–682, New York, NY, USA, 2002. ACM Press.
- [PH89] K. Perlin and E. M. Hoffert. Hypertexture. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 253–262, New York, NY, USA, 1989. ACM Press.
- [SF92] Mikio Shinya and Alain Fournier. Stochastic motion — motion under the influence of wind. In *Computer Graphics Forum (Eurographics '92)*, pages 119–128, September 1992.
- [TR04] et al. Ton Rosendall, Steanfo Selleri. *The Official Blender 2.3 Guide*. No Starch Press, 555 De Haro Street, Suite 250, San Francisco, CA 94107, 2004.

Index

- adaptive depth control, 17
- ADC, *see* adaptive depth control
- animation
 - motion generation, 27
- anti-aliasing, *see* supersampling
- ‘bailout’, 17
- bbox, *see* bounding box
- Blender, 25
- ‘bounded_by’, 15
- bounding box, 10
- bugs
 - avoidance of, 23
 - noticeability of, 23
 - untraceability of, 28
- camera
 - types of, 7
- ‘clock’, 8
- concurrency, 8
- Default_Hyperbound, 15
- Default_Project, 16
- Deinitialize_Hyper_Code, 18
- density fall-off, 21
- Determine_Apparent_Colour, 7
- Determine_Intersection_Colour, 17
- diffuse scattering, 12
- explicit geometry
 - creation from procedural fur, 28
 - level-of-detail decisions, 6
 - use for long hair, 6
- filter_shadow_ray, 8
- fin texture, 6
- ‘finish’, 9
- hair, 6
- heightfields, 27
- ‘hyper’, 9
- Hyper flag in Intersection, 10
- hyperbound, 10
 - automatic generation, 15
- image mapping, 27
- index of refraction, 9
- ‘interior’, 9
- intersection, 6
- Istack, 15
- Julia fractals, 27
- Lambertian scattering, *see* diffuse scattering
- lex, 14
- lighting model
 - implementation, 21
 - theoretical basis, 12
- lights
 - listing, 18
 - local, 18
- ‘material’, 9
- media
 - conflict with hypertexture, 17
 - detecting, 16
 - order of operations, 16
- media fur patch, 24
- noise, 11
- ‘normal’, 9
- normal perturbation, 20
- Nyquist, 19
- object, 6

parallelism, 8
Parse_Hyper, 14
Parse_Object_Mods, 15
 parser
 execution of, 7
 unreadability of, 14
 participating media, *see* media
 particle system
 use in Blender, 25
 use to generate density model, 11
 pelt, 6
 Perlin noise, *see* noise
 Persistence of Vision, *see* POV-Ray
 physical simulation, 27
 ‘pigment’, 9
 point of closest approach, 10
Post_Hyper, 15
 POV-Ray, 5
 difficulty of modification, 15
 features of, 5
 horror at reading code of, 28
 lack of comments in, 14
 popularity of, 25
 primitive, 6
 PRNG, *see* random numbers
 projection function, 11

 random numbers, 8
 fast generation, 21
Ray, 16
 ray
 primary, 7
 shadow, 7
 use of origin to indicate start of sampling region, 18
 regression testing, 24, 27

 sample size, 19
 sampling, 18
scattering, 21
 scene-description language, 5
 generation by third-party programs, 27
 inextensibility of, 14
 SDL, *see* scene description language

 shell texture, 6
Simulate_Hyper, 18, 29
Simulate_Hyper_Shadow, 18, 31
 soft region, 5
 specular highlights, 12
 strand, 6
 jittered placement, 21
 length variation, 21
 supersampling, 8

Test_Shadow, 7
 texel, 5
 ‘texture’, 9
 texture hierarchy, 9
 texture space, 10
 threads, 8
 tokens, 14

 UV co-ordinates, 10
 faking, 20

 volumetric media, *see* media

 XBox, 6

 yacc, 14